

# Recherches de motifs et de similarités en bioinformatique : modélisations, solutions logicielles et matérielles

Mathieu Giraud<sup>1</sup> Laurent Noé<sup>2</sup> Gregory Kucherov<sup>2</sup> Dominique Lavenier<sup>1</sup>

<sup>1</sup>IRISA / CNRS / Université de Rennes 1, Campus de Beaulieu, 35042 Rennes cedex  
{mgiraud, lavenier}@irisa.fr

<sup>2</sup>LORIA / INRIA / Université de Nancy 1, Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy cedex  
{noe, kucherov}@loria.fr

**Résumé :** Ce tutoriel expose certains problèmes fondamentaux en algorithmique du texte pour la bioinformatique, leurs solutions actuelles ainsi que quelques perspectives de recherche.

Après une introduction expliquant pourquoi la bioinformatique s'intéresse aux séquences de caractères et d'où provient le problème de recherche de motifs, nous présentons de façon progressive différentes modélisations des motifs (partie 2). Un motif peut être un simple mot ou un ensemble de mots que l'on recherche de manière exacte ou approchée, par similarités. Plus généralement, on définit un motif comme un langage pouvant se situer à différents niveaux de la hiérarchie de Chomsky et formalisable par des structures telles que des grammaires ou des automates.

Le tutoriel se poursuit par la présentation des méthodes logicielles ou matérielles qui résolvent les recherches de motifs selon la modélisation choisie (partie 3). Ces algorithmes s'effectuent avec ou sans pré-traitements du motif ou de la banque de séquences. Quand les motifs deviennent complexes, la recherche par balayage devient la seule solution possible, que cela soit par heuristique ou de manière exacte. Nous évoquerons aussi les architectures spécialisées destinées à traiter de grandes quantités de données : ces machines doivent équilibrer puissance de calcul et accès aux données.

**Mots-clés :** recherche de motifs, bioinformatique, distance d'édition, langages, automates, programmation dynamique, heuristiques à base de graines, architectures, FPGA.

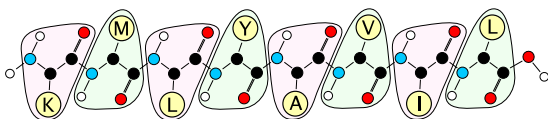


FIG. 1 – Structure des protéines. Une protéine est un polymère peptidique, c'est-à-dire un assemblage de plusieurs acides aminés.

## 1 INTRODUCTION : ADN, PROTÉINES, MODÉLISATION DE FAMILLES DE SÉQUENCES ET BANQUES DE DONNÉES

### 1.1 De l'ADN aux protéines

Le *métabolisme*, ou fonctionnement des cellules, est assuré en grande partie par des molécules appelées *protéines*. Les protéines sont des successions d'*acides aminés* qui, repliées dans une certaine structure 3D, ont une fonction au sein de la cellule (figure 1). Il y a 20 acides aminés :

$$\Sigma_{20} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$$

L'information codant les protéines est conservée dans les *chromosomes* d'une cellule sous forme d'*acide désoxyribonucléique (ADN, figure 2)*. L'ADN est une succession de *nucléotides* pris parmi l'alphabet à 4 lettres :

$$\Sigma_4 = \{A, C, G, T\}$$

Un des rôles de la machinerie cellulaire est de transformer l'ADN en protéines (figure 3). L'ADN est transcrit en *ARN messenger*. Celui-ci est traduit en protéines par les ribosomes selon le *code génétique* qui associe à chaque *codon* (c'est-à-dire à chaque triplet de nucléotides de  $\Sigma_4$ ) un acide aminé de  $\Sigma_{20}$ . Les *codons stop* provoquent l'arrêt de la traduction (tableau 1). La traduction peut se faire selon trois *phases de lecture* différentes par sens de lecture, soit six phases au total (figure 4).

Un *gène* est la portion d'ADN qui code une protéine. L'ADN *codant* peut représenter moins de 1 % des génomes eucaryotes. Les autres régions de l'ADN, *non codantes*, ont d'autres fonctions telles que la régulation. Le tableau 2 donne quelques ordres de grandeurs sur l'ADN et les protéines.

### 1.2 Modèles et motifs pour coder les similarités

La comparaison des protéines entre elles et leur regroupement en *familles fonctionnelles* passent par l'étude des similitudes entre leurs structures 3D ou leurs séquences 1D. La plupart du temps, les protéines d'une même famille proviennent d'un ancêtre commun.

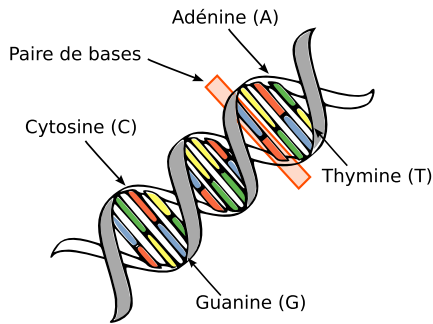


FIG. 2 – Structure en double hélice de l’ADN. Chaque nucléotide est composé d’un groupe phosphate, d’un sucre (le désoxyribose), ainsi que d’une *base azotée* A, C, T ou G. Les bases A et T, ainsi que G et C, sont complémentaires : un pont hydrogène les relie.

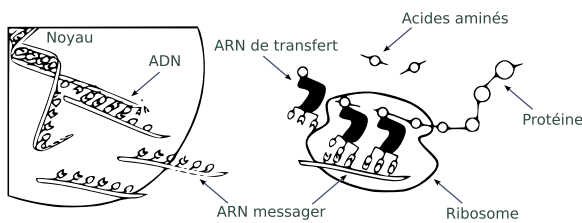


FIG. 3 – La machinerie cellulaire. Dans le noyau, l’ADN des chromosomes est transcrit en ARN messager par la transcriptase. L’ADN messager migre vers le cytoplasme, et les ribosomes le traduisent en protéines.

1	T	C	A	G
2 3	TCAG	TCAG	TCAG	TCAG
T	F L	L	I I I M	V
C	S	P	T	A
A	Y s	H Q	N K	D E
G	C C s W	R	S R	G

TAB. 1 – Le code génétique associe chaque triplet de nucléotides (comme CAT) différent des codons stop (s) à un acide aminé (comme l’histidine H).

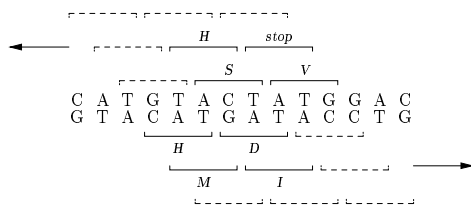


FIG. 4 – La machinerie cellulaire peut traduire les chaînes nucléiques en protéines suivant six phases distinctes selon l’endroit où est initiée la traduction ainsi que selon le sens de lecture.

### 1.2.1 Similarités 3D

La fonction métabolique d’une protéine dépend fortement de sa structure 3D : les *sites actifs* de la protéine interagissent avec d’autres molécules appelées *ligands*. Les structures 3D des protéines sont difficilement accessibles. La *crystallographie* permet de connaître quelques structures complètes de protéines dans certains milieux, mais cette technique reste lourde. La résolution directe des équations physico-chimiques sur de si grandes molécules est combinatoirement hors de portée des moyens de calculs actuels. D’autres recherches sur le *repliement de protéines* tentent de comprendre la structure 3D d’une protéine à partir des séquences 1D [Andonov et al., 2004].

### 1.2.2 Motifs

Il est surtout possible d’évaluer les similarités entre protéines en travaillant directement avec les séquences 1D. En effet, la structure 3D d’une protéine est fonction de la séquence linéaire 1D des acides aminés qui la composent sur  $\Sigma_{20}$ . Les similitudes peuvent être partielles : une similarité de 30% dans les séquences 1D suffit généralement à obtenir une même structure 3D. Pour rechercher la fonction d’une protéine ou d’un gène, on essaie donc de retrouver des similarités entre sa séquence d’acides aminés ou de ses bases d’ADN et les séquences déjà connues. Les protéines présentant des similarités de séquences 1D et de structures 3D sont appelées *homologues*.

Lorsqu’une famille de séquences a été identifiée (par similarité, expériences en laboratoire, ou expertise humaine), il est possible de construire un *modèle* qui la représente que nous appelons *motif*. Le motif peut :

1. représenter un *domaine* de quelques dizaines d’acides aminés, bien conservé au cours de l’évolution dans une même famille. Ces domaines, sont idéalement caractéristiques d’une fonction donnée ou d’un certain site actif ;
2. être un modèle calculé ou appris sur toutes les séquences. Le motif se représente alors par un *langage* qui décrit une certaine structure des séquences. Ces modèles *génèrent* des séquences, souvent avec un score ou une probabilité qui mesure l’adéquation entre le modèle et la séquence.

Un exemple de représentation par motif couramment utilisée par les biologistes est la syntaxe PROSITE [Hulo et al., 2004] comme dans le motif [FYWL]-x-[LIVM]-H-G-L-W-P caractéristique d’une famille de ribonucléases. Les crochets désignent le choix entre plusieurs acides aminés et le x est un *joker*, c’est-à-dire un acide aminé quelconque. Il est possible d’avoir des jokers de longueur variable comme dans le motif C-x(6,8)-[LFY]-x(5)-[FYW]-x-[RK]-x(8,10)-C-x-C-x(6,9)-C où le premier joker a une longueur comprise entre 6 et 8. Des programmes tels que PRATT [Jonassen et al., 1995] sont capables d’inférer de tels motifs à partir d’un jeu de séquences.

La syntaxe PROSITE ne permet de représenter qu’un sous-ensemble des expressions régulières : les motifs peuvent être représentés par des automates (figure 5

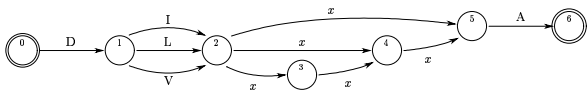


FIG. 5 – Un automate décrivant le motif PROSITE D-[ILV]-x(1,3)-A.

et partie 2.3). Pour gagner en expressivité, il est intéressant d'utiliser la seconde catégorie de modèles, par exemple en allant plus loin dans la hiérarchie de Chomsky ([Chomsky, 1957], en considérant que le modèle est un *language* à représenter d'une manière ou d'une autre (voir partie 2.1.1).

### 1.3 Banques génomiques et puissance de calcul

L'automatisation des outils de séquençage conduit depuis une trentaine d'années à une croissance exponentielle des données biologiques et de leurs données associées (annotations, méta-données). La banque américaine Genbank, la banque japonaise DDBJ et la banque européenne EMBL mettent en commun toutes les séquences nucléiques du domaine public séquencées dans le monde au sein de l'INSDC (*International Nucleotide Sequence Database Collaboration*). La banque EMBL [Stoesser et al., 2003] contient, en mai 2005, 85 milliards de bases réparties en 49 millions d'entrées. Cette quantité double tous les 15 mois (figure 6).

À partir de ces données brutes et faiblement structurées, les biologistes cherchent quotidiennement à obtenir un aperçu des fonctions métaboliques et donc, finalement, une compréhension de certains mécanismes biologiques. Cette compréhension peut être utilisée notamment en pharmacologie (présélection bio-informatique de substances actives sur certaines protéines) ou en génie génétique. Elle passe cependant par des traitements lourds lors d'opérations comme la recherche de similarités, de motifs ou de modèles.

Contrairement aux bases de données usuelles, les banques de séquences sont des collections *non structurées* de données brutes. Certaines applications peuvent indexer les banques (voir partie 3.1.2), mais d'autres nécessitent des balayages complets auquel l'indexation n'apporte aucune aide [Williams et Zobel, 2002]. La croissance des banques devient alors problématique si on la compare à celle du pouvoir de calcul des processeurs qui, lui, double seulement tous les 18 mois selon la loi de Moore [Moore, 1965, ITR, 2004] (figure 6). De nouvelles solutions, logicielles comme matérielles, sont donc indispensables pour exploiter ces masses de données.

## 2 MODÉLISATIONS

Cette partie présente quelques éléments se rapportant à la théorie des langages et montre différentes manières de modéliser les motifs en bioinformatique. Un motif est un *language*, langage pouvant se situer à différents niveaux de la hiérarchie de Chomsky (partie 2.1).

Le langage peut être réduit à un *mot* ou un *ensemble de*

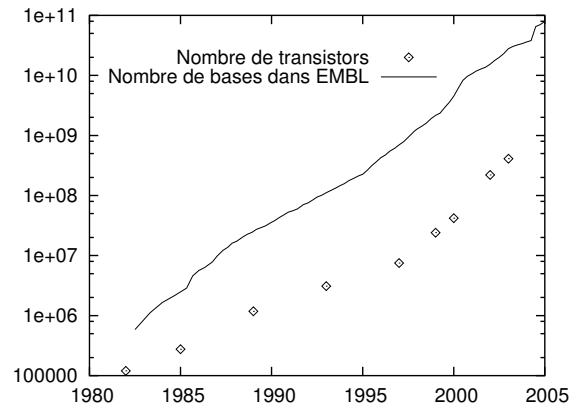


FIG. 6 – Nombre de transistors dans les processeurs (Intel) et taille des banques de données génomiques (EMBL).

*mot*s. Pour faire des recherches approchées, une extension est de considérer tous les mots “proches” du mot initial selon une *distance d'édition* calculable par programmation dynamique (2.2).

Le langage peut aussi être défini par d'autres outils, en particulier par des *automates* finis ou pondérés (2.3 et 2.3). Rechercher un motif équivaut à opérer un *filtrage* sur une banque de données en sélectionnant des séquences supposées pertinentes parmi un ensemble plus large, filtrage dont on peut *mesurer la qualité* (2.4).

### 2.1 Mots et langages

Soit  $\Sigma$  un ensemble fini appelé *alphabet*. Les éléments de  $\Sigma$  sont appelés des *caractères*. Un *mot* est une suite finie de caractères  $w = w_1w_2 \dots w_n \in \Sigma^*$ . Un *sous-mot* de  $w$  est une suite extraite de  $w$ , et un *facteur* de  $w$  une sous-suite  $w_a w_{a+1} \dots w_b$  extraite continuellement. Un *language*  $\mathcal{L} \subset \Sigma^*$  est un ensemble de mots (partie de  $\Sigma^*$ ).

#### 2.1.1 Classes de langages

La hiérarchie de Chomsky définit différentes classes de langages [Chomsky, 1957, Hopcroft et Ullman, 1969] :

- les *langages réguliers* sont les langages reconnus par les automates finis. Ils se représentent par les expressions régulières (figure 7) ;
- les *langages algébriques (context-free)* sont les langages reconnus par les automates à pile ;
- suivent d'autres langages plus complexes, jusqu'aux *langages récursivement énumérables* qui sont tous les langages qu'une machine de Turing peut énumérer. Le problème d'appartenance d'un mot à un langage est alors non décidable dans le cas général.

#### 2.1.2 Intérêt pour la bio-informatique

Les travaux de Searls [Searls, 1992, Searls, 1993a, Searls, 1997] montrent que certaines structures biologiques se formalisent par des langages à différents niveaux de la hiérarchie de Chomsky. Certaines “structures secondaires” doivent être décrites par des motifs algébriques (tableau 3). L'ambiguïté de certains de ces motifs

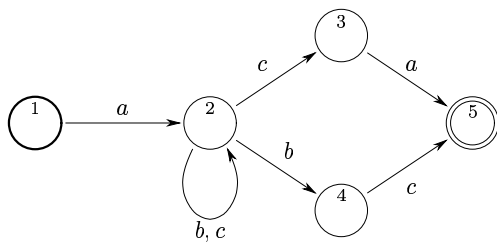


FIG. 7 – Automate fini  $\mathcal{A}_1$ . Un automate fini reconnaît un mot  $w$  si et seulement si il existe un chemin étiqueté par  $w$  conduisant d’un état initial (ici l’état 1) à un état final (ici l’état 5). L’automate fini  $\mathcal{A}_1$  sur l’alphabet  $\Sigma = \{a, b\}$  reconnaît ainsi le langage rationnel  $\mathcal{L}_1 = (a|b)(b|c)^*(ca|bc)$ .

se traduit biologiquement par des repliements distincts. Les répétitions ou les pseudo-nœuds demandent même des grammaires dépendants du contexte. Des travaux plus récents ont introduit d’autres classes de motifs et de langages [Rivas et Eddy, 2000, Leung et al., 2001].

Les langages réguliers ou algébriques ne doivent cependant pas être abandonnés, les différentes modélisations des séquences biologiques visant à remplir deux critères opposés : les motifs choisis doivent posséder une *expressivité* la plus grande possible pour modéliser les propriétés biologiques étudiées, tout en gardant une bonne *efficacité* pour des solutions logicielles et/ou matérielles adaptées aux problèmes posés.

## 2.2 Recherches par similarités

Dans sa forme la plus simple, un motif est un simple mot  $v = \text{AAATTACGT}$ . Cette partie montre comment autoriser des *erreurs d’édition* dans la reconnaissance de motifs, ce qui mène aux méthodes de comparaison de séquences utilisant la programmation dynamique, telles que celles proposées par Needleman et Wunsch [Needleman et Wunsch, 1970] ainsi que Smith et Waterman [Smith et Waterman, 1981], respectivement en 1970 et 1981. La première évalue la similarité entre deux chaînes d’ADN ou entre deux protéines et calcule le meilleur alignement global, tandis que la seconde trouve des paires de sous-séquences hautement similaires en calculant des alignements locaux. Les deux méthodes utilisent un *coût d’alignement* qui peut être vu comme un *score de similarité*. Le motif sera ainsi donné par une *séquence requête*  $v$  ainsi qu’un *seuil* sur le score.

### 2.2.1 Similarités et alignements

Mesurer la similarité entre deux séquences demande à *aligner* ces séquences, c’est-à-dire à mettre en correspondance leurs régions similaires pour estimer le *coût* pour transformer une séquence en l’autre. À chaque position dans l’alignement correspond une des trois situations suivantes (figure 8) :

- un *match*, quand le même caractère  $\alpha$  apparaît dans les deux séquences,
- une *substitution* (ou *mismatch*) lorsqu’il y a deux ca-

ractères différents  $\alpha$  et  $\beta$ ,

- ou un *gap*, c’est-à-dire une *insertion* d’un caractère dans seulement une séquence, ou symétriquement une *délétion* dans une des deux séquences.

Ces opérations, bien qu’elles soient une idéalisation de l’évolution d’un génome, correspondent à un mécanisme réel d’erreur de copie de la machinerie cellulaire : substitutions, insertions et délétions ont une fréquence généralement comprise entre  $10^{-5}$  et  $10^{-8}$  lors de chaque duplication de l’ADN.

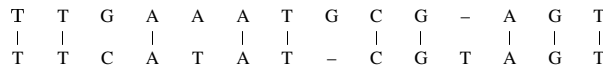


FIG. 8 – Alignement global de deux chaînes d’ADN avec 10 matches, deux mismatches G/C et A/T, et deux gaps G/- et -/T.

Le nombre minimum de substitutions, insertions et délétions pour passer d’une chaîne à l’autre est appelé *distance d’édition* ou *distance de Levenshtein* [Levenshtein, 1966].

La distance d’édition peut se raffiner en utilisant des *scores de substitution*, notamment lorsqu’on traite des séquences protéiques. On dispose alors d’une fonction  $d(\alpha, \beta)$  mesurant la similarité entre  $\alpha$  et  $\beta$ . Les scores de la matrice BLOSUM62 sont ainsi liés aux probabilités de substitution des acides aminés, probabilités estimées sur l’évolution d’organismes sur des milliers de générations [Henikoff et Henikoff, 1992] (figure 9).

De même, on peut fixer une pénalité  $g_{\text{penalty}}$  pour chaque insertion ou délétion et considérer cette pénalité comme un cas particulier de la fonction :  $d(\alpha, \varepsilon) = d(\varepsilon, \beta) = g_{\text{penalty}}$ . On retrouve ainsi la distance de Levenshtein avec la fonction caractéristique  $d(\alpha, \alpha) = 0$  et, si  $\alpha \neq \beta$ ,  $d(\alpha, \beta) = -1$ .

Une autre distance usuelle compte le nombre minimum d’insertions et délétions entre deux chaînes : une substitution est alors vue comme une délétion suivie d’une insertion. Cette distance  $d_{12}$  se calcule par  $d_{12}(\alpha, \alpha) = 0$ ,  $d_{12}(\alpha, \varepsilon) = -1$ ,  $d_{12}(\varepsilon, \alpha) = -1$ , et, et  $d_{12}(\alpha, \beta) = -2$  lorsque  $\alpha \neq \beta$  sont deux caractères différents appartenant à  $\Sigma$ .

$$\begin{cases} g_{\text{penalty}} = -1 \\ \forall \alpha, d_{12}(\alpha, \alpha) = 0 \\ \forall \alpha, \beta, \alpha \neq \beta, d_{12}(\alpha, \beta) = -2 \end{cases}$$

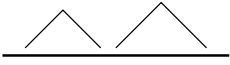
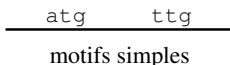

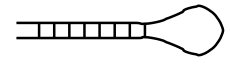

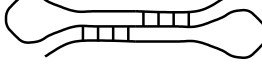
### 2.2.2 Relations de programmation dynamique

Appelons  $X = (x_1, x_2 \dots x_m)$  et  $Y = (y_1, y_2 \dots y_n)$  les deux séquences à comparer, et  $H(i, j)$  le score de similarité maximum entre les deux sous-séquences  $x_0 \dots x_i$  et  $y_0 \dots y_j$ . On peut calculer  $H(i, j)$  par une récursion à deux dimensions en utilisant l’équation de Needleman-Wunsch (NW, [Needleman et Wunsch, 1970]) :

$$\begin{aligned} \forall i : H(i, 0) &= g_{\text{penalty}} \times i \\ \forall j : H(0, j) &= g_{\text{penalty}} \times j \\ \forall i, j, i \neq 0, j \neq 0 : & \end{aligned} \quad (1)$$

	ADN	protéines
éléments de base	nucléotides (bases, pb) $\Sigma_4$ , 4 possibilités	acides aminés (aa) $\Sigma_{20}$ , 20 possibilités
fabrication	auto-réplication	par la machinerie cellulaire à partir de l'ADN
orientation	5' – 3'	NH <sub>2</sub> – COOH
tailles typiques	1 gène : 300 – 3000 pb génomme bactérien : 100 000 bases génomme humain : $3 \times 10^9$ bases	1 protéine : 100 – 1000 aa  30 000 gènes
banques de données (mai 2005)	EMBL : $85 \times 10^9$ bases	TrEMBL : $555 \times 10^6$ aa SwissProt : $65 \times 10^6$ aa

TAB. 2 – ADN et protéines. Les séquences nucléiques (ADN) ou protéiques sont des mots sur un alphabet à 4 ou 20 lettres. La banque protéique UniProt se répartit en deux banques : SwissProt, qui recense les protéines identifiées, et TrEMBL, une traduction automatique de la banque nucléique EMBL.

classe de motifs	reconnaissance		signification biologique	
	par automate	complexité	dépendances	exemples nucléiques
rationnels $((g a)(c t))^*$	automates finis	linéaire	 dépendances locales	 motifs simples
algébriques (context-free) $S \mapsto gSc$	automates à pile	quadratique	 dépendances imbriquées	 structures secondaires (tige-boucle)
dépendant du contexte (context-sensitive) $At \mapsto aA$	automates bornés	NP-complet	 dépendances croisées	 répétitions pseudo-nœuds

TAB. 3 – Les motifs rationnels ne suffisent pas à décrire correctement les structures des séquences nucléiques. Ce tableau, inspiré de [Searls, 1997], montre les relations entre les divers classes de langage de la hiérarchie de Chomsky et les structures biologiques. Une structure comme les tige-boucles correspond à un palindrome complémentaire, comme dans TTATTTGCATCGGGCAAATCG.

	D	E	H	I	L
D	6				
E	2	5			
H	-1	0	8		
I	-3	-3	-3	4	
L	-4	-3	-3	2	4

FIG. 9 – Extrait de la matrice BLOSUM62 qui associe à chaque paire d’acides aminés un score de substitution [Henikoff et Henikoff, 1992].

$$H(i, j) = \max \begin{cases} H(i-1, j-1) + d(x_i, y_j) \\ \text{(match ou substitution)} \\ H(i-1, j) - g_{\text{penalty}} \\ \text{(insertion)} \\ H(i, j-1) - g_{\text{penalty}} \\ \text{(délétion)} \end{cases}$$

La figure 10 donne un exemple de calcul de matrice de programmation dynamique avec l’équation NW. La quantité  $\mathcal{H}(X, Y) = H(m, n)$  obtenue en fin de calcul est la *similarité globale* entre  $X$  et  $Y$ .

Cependant, dans beaucoup d’applications, on préfère rechercher des *similarités locales*, par exemple lorsqu’on compare deux grandes séquences d’ADN. On cherche donc les meilleures sous-séquences similaires entre  $X$  et  $Y$ . Appelons  $H(i, j)$  la similarité de la paire de sous-séquences les plus similaires parmi celles terminant en  $x_i$  et en  $y_j$ . Cette similarité peut être calculée par l’équation de Smith-Waterman (SW, [Smith et Waterman, 1981]) :

$$\begin{aligned} \forall i, j : H(i, 0) = H(0, j) = 0 \\ \forall i, j, ij \neq 0 : \end{aligned} \quad (2)$$

$$H(i, j) = \max \begin{cases} 0 \\ \text{(début d'un alignement local)} \\ H(i-1, j-1) + d(x_i, y_j) \\ \text{(match ou substitution)} \\ H(i-1, j) - g_{\text{penalty}} \\ \text{(insertion)} \\ H(i, j-1) - g_{\text{penalty}} \\ \text{(délétion)} \end{cases}$$

La figure 10b donne un exemple de calcul utilisant SW. La quantité  $\mathcal{H}(X, Y) = \max H(i, j)$  représente la meilleure similarité locale entre  $X$  et  $Y$ . On peut aussi utiliser des relations *globales-locales* qui calculent la similarité entre des sous-séquences de  $X$  et la chaîne  $Y$  entière. Dans tous les cas, on peut fixer un seuil  $h_0$  et rechercher toutes les chaînes  $X$  dans une banque de séquences avec une similarité  $\mathcal{H}(X, Y) \geq h_0$ .

Dans les équations de programmation dynamique, la propagation des scores dans la matrice de programmation dynamique est uniquement locale : chaque cellule intérieure reçoit les scores des trois cellules précédentes et envoie son résultat aux trois cellules suivantes (figure 11, à gauche). La partie 3.2 montre comment calculer effica-

cement ces  $\mathcal{O}(mn)$  cellules par des architectures systoliques.

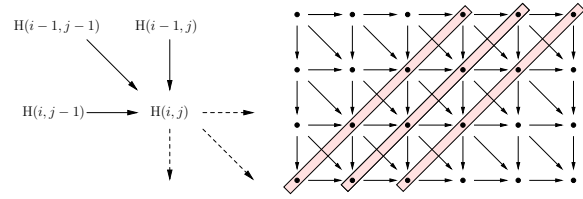


FIG. 11 – Localité du calcul dans une cellule SW/NW (à gauche). Les flèches noires montrent les scores provenant des trois cellules précédentes. La seule autre information dont la cellule a besoin est  $d(x_i, y_j)$ . Les flèches en pointillés montrent la propagation des données vers les trois cellules suivantes. Lorsqu’on cherche à calculer toutes les cellules d’une matrice de programmation dynamique, les cellules le long d’une même anti-diagonale peuvent être calculées simultanément (à droite).

### 2.2.3 Fonctions de gap

Dans les équations (1) et (2), la pénalité  $g_{\text{penalty}}$  est constante. Un gap de  $\ell$  positions successives obtient donc une pénalité globale  $g(\ell) = g_{\text{penalty}} \ell$  : la pénalité peut être vue comme une fonction linéaire. On peut considérer d’autres fonctions de gap plus proches de la réalité biologique : il est “plus coûteux” d’ouvrir un gap que d’en étendre un existant. On utilise ainsi souvent des fonctions affines de type  $g(\ell) = g_{\text{open}} + \ell g_{\text{extend}}$ . Dans ce cas, les relations de programmation dynamique peuvent utiliser plusieurs matrices, comme l’a montré Gotoh en 1982 [Gotoh, 1982] :

$$\forall i, j, ij \neq 0 : \quad (3)$$

$$H(i, j) = \max \begin{cases} 0 \\ \text{(si alignement local SW)} \\ H(i-1, j-1) + d(x_i, y_j) \\ \text{(match ou substitution)} \\ I(i, j) \\ \text{(insertion)} \\ D(i, j) \\ \text{(délétion)} \end{cases}$$

avec :

$$D(i, j) = \max \begin{cases} H(i, j-1) - g_{\text{open}} \\ D(i, j-1) - g_{\text{extend}} \end{cases}$$

$$I(i, j) = \max \begin{cases} H(i-1, j) - g_{\text{open}} \\ I(i-1, j) - g_{\text{extend}} \end{cases}$$

Lorsqu’une fonction affine est utilisée, on peut toujours calculer toutes les cellules en  $\mathcal{O}(mn)$  opérations [Gotoh, 1982].

La programmation dynamique permet donc, en partant d’une séquence, d’une distance et d’un seuil, d’accepter un ensemble de séquences similaires à la séquence initiale.

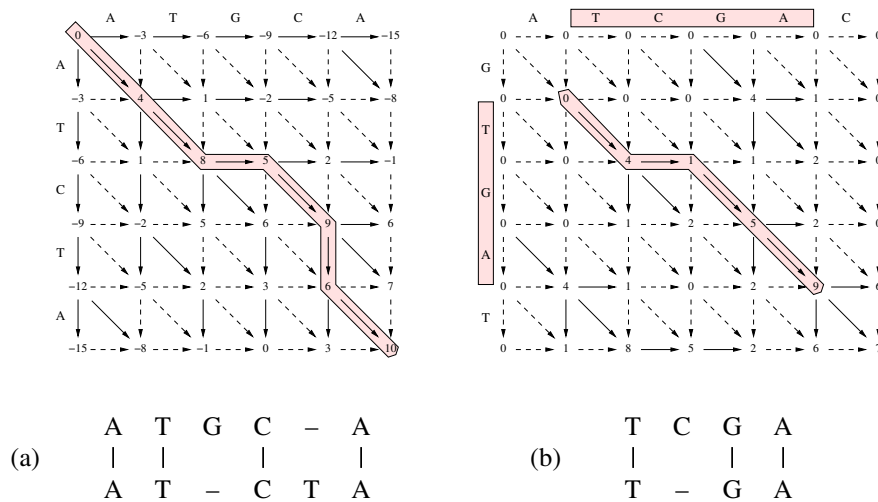


FIG. 10 – Exemple de calcul d’alignement global avec les équations NW (a) et d’alignement local avec les équations SW (b). Les scores sont de +4 pour un match, -2 pour un mismatch et -3 pour un gap. Les flèches continues montrent d’où provient le score qui a maximisé chaque cellule. La succession de ces flèches révèle le meilleur alignement.

### 2.3 Vers d’autres modélisations des motifs

Nous présentons maintenant des modélisations de motifs qui reconnaissent plusieurs séquences, que cela soit par profils ou par modèles plus généraux.

#### 2.3.1 Profils dépendant de la position

Étant donné une famille de séquences similaires, il est possible de construire un *alignement multiple* comme sur la figure 12 [Thompson et al., 1994].

```

FOS_RAT      LVQPTLVSSVAPSQ-----TRAPHPYGLP
FOS_MOUSE   LVQPTLVSSVAPSQ-----TRAPHPYGLP
FOS_CHICK    LVQPTLISVAPSQ-----NRG-HPYGV
FOSB_MOUSE  LVQPTLISMAQSQQPLASQPPAVDPYDMP
FOSB_HUMAN  LVQPTLISMAQSQQPLASQPPVVDYDMP
*****:** * **...  :..  .**:**

```

FIG. 12 – Alignement multiple de cinq protéines avec ClustalW [Thompson et al., 1994]. Sur la ligne du bas, \* indique un match, et : et . des matches plus faibles.

À partir d’un alignement multiple, on établit un *motif consensus* comportant à chaque position l’acide aminé le plus présent. Des choix tels que [PQ] peuvent être employés lorsque plusieurs acides aminés ont le même nombre d’occurrences (figure 13a). De nombreux motifs ont été créés, notamment dans la banque PROSITE [Hulo et al., 2004]. Ils correspondent à un langage  $\mathcal{L}$  comprenant plusieurs mots (dictionnaire), et ne sont qu’une expression régulière très simple (figure 5).

On peut souhaiter conserver toutes les informations statistiques d’un alignement dans un *profil* qui enregistre la distribution des acides aminés à chaque position (figure 13b). L’adéquation  $[\mathcal{P}](w)$  d’une séquence  $w$  à un profil  $\mathcal{P}$  est calculée en multipliant les probabilités à chaque position, les utilisations réelles utilisent des valeurs logarithmiques. On fixe un seuil  $s_0$  au-delà duquel la séquence est reconnue.

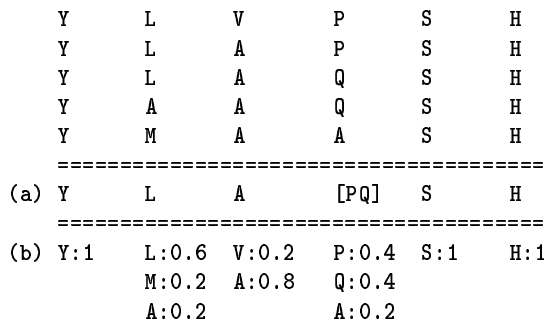


FIG. 13 – Consensus (a) et profil (b) à partir d’un alignement multiple. Le motif consensus YLA [PQ] SH peut être un ancêtre de la famille. Le profil (b) assigne la probabilité 0,144 à YLAPSH et 0.016 à YMVPSH : la première séquence a plus de chances d’appartenir à la famille considérée.

#### 2.3.2 Modèles de Markov cachés

Un processus stochastique est dit *markovien* pour une suite d’événements  $X_1, X_2, \dots, X_n$  si la distribution probabiliste pour l’événement  $X_i$  dépend uniquement de l’événement précédent  $X_{i-1}$ . Un modèle de Markov caché (noté HMM) est constitué d’un processus markovien auquel on associe, pour chaque événement  $X_i$ , une distribution probabiliste d’observations  $O(X_i)$ . Seules les observations et leur distribution pour un événement sont connues. On souhaite alors déterminer la suite ou les suites d’événements les plus probables qui engendrent la suite d’observations. Dans le cadre de la comparaison de séquences, on peut par exemple fixer les types d’événements (par exemple les événement d’édition : match ( $m_j$ ), insertion ( $i_j$ ), délétion ( $d_j$ )) et créer une topologie adéquate de HMM  $\mathcal{M}$  (voir la figure 14).

Dès que le modèle  $\mathcal{M}$  est entraîné, il calcule pour chaque séquence  $w$  un *score d’adéquation*  $[\mathcal{M}](w)$ . L’algorithme de Viterbi indique la suite d’événements la plus

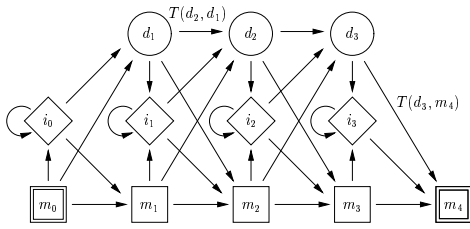


FIG. 14 – Exemple d’une topologie  $\mathcal{M}$  de modèle de Markov caché, comme proposé par Haussler dans [Haussler et al., 1993].

probable correspondant à la séquence  $w$ , c’est-à-dire le meilleur chemin dans la topologie du HMM et donc le meilleur alignement avec le motif consensus.

Les HMM sont utilisés depuis les années 1980 dans la reconnaissance de la parole. Leur première utilisation dans un contexte de bio-informatique fut en 1993 par Haussler [Haussler et al., 1993].

Les modélisations présentées dans la partie précédente peuvent se généraliser par des *automates* : le motif recherché appartiendra à un langage reconnu par un automate fini (expression rationnelle), ou, plus généralement, par un automate pondéré. Les automates pondérés affectent des poids aux transitions et décrivent des classes de langages strictement supérieures aux langages rationnels [Sakarovitch, 2003, Kuich et Salomaa, 1986]. Leurs poids ne sont pas nécessairement liés à des probabilités, et leur topologie peut être quelconque. On considère un semi-anneau  $(\mathbb{K}, \oplus, \otimes)$  où  $\bar{0}$  est l’élément neutre de  $\oplus$ .

### 2.3.3 Automates pondérés

**Définition 1** Un automate pondéré (weighted finite automaton, WFA) est un quintuplet  $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ , où  $Q$  est un ensemble fini d’états,  $\Sigma$  un alphabet fini,  $\Delta \subset Q \times Q \times (\delta : \Sigma \cup \{\varepsilon\} \mapsto \mathbb{K})$  la table de transitions finie,  $I \subset Q$  et  $F \subset Q$  les ensembles d’états initiaux et finaux [Eramian, 2002, Mohri, 1997a]

Soit une transition  $t = (q, q', \delta) \in \Delta$ . On appelle  $p[t] = q$  son état *source*,  $n[t] = q'$  son état *cible*, et  $\delta[t] = \delta$  sa *fonction de poids*. La transition  $(q, q', \delta)$  avec  $\delta(\alpha) = x$  pour  $\alpha \in \Sigma \cup \{\varepsilon\}$  signifie qu’un poids  $x$  est associé à la transition  $q \mapsto_\alpha q'$ .

Une transition  $(q, q', \delta)$  est *vide* si  $\delta(\alpha) = -\infty$  pour tous les  $\alpha \in \Sigma \cup \{\varepsilon\}$ . Sans perte de généralité, on suppose que  $\Delta$  ne contient que des transitions non vides. Le *nombre de transitions* du WFA est  $|\Delta|$ . Un WFA sans  $\varepsilon$ -transitions est un WFA tel que  $\delta(\varepsilon) = -\infty$  pour chaque transition  $(q, q', \delta)$ .

### 2.3.4 Chemins et poids

Les *chemins* sont définis comme des suites de transitions étiquetées :

**Définition 2** Un chemin  $\pi = (t_1, \alpha_1) \dots (t_k, \alpha_k) \in (\Delta \times (\Sigma \cup \{\varepsilon\}))^*$  dans un WFA  $\mathcal{A}$  est une succession de paires de transitions et de caractères telles que

les transitions  $t_1 \dots t_k$  soient consécutives, c’est-à-dire  $n[t_i] = p[t_{i+1}]$  pour  $i = 1 \dots k - 1$ , et où les caractères  $\alpha_i$  sont dans  $\Sigma \cup \{\varepsilon\}$ . L’étiquette de  $\pi$  est le mot  $\ell[\pi] = \alpha_1 \dots \alpha_k$ .

La fonction de poids  $\delta$  s’étend aux chemins : pour un chemin  $\pi = (t_1, \alpha_1) \dots (t_k, \alpha_k)$ , on définit

$$\delta(\pi) = \delta[t_1](\alpha_1) + \dots + \delta[t_k](\alpha_k).$$

Soit un chemin  $\pi = (t_1, \alpha_1) \dots (t_k, \alpha_k)$ . On appelle les états  $q = p[t_1]$  et  $q' = n[t_k]$  le *début* et la *fin* du chemin  $\pi$ . Étant donné un mot  $w \in \Sigma^*$ , on note  $\Pi_{\mathcal{A}}(w)$  l’ensemble des chemins étiquetés par  $w$  débutant dans un état initial  $q_i \in I$  et terminant dans un état final  $q_f \in F$ . Puisque l’opérateur  $\oplus$  est commutatif, on peut définir le poids calculé par un automate pondéré :

**Définition 3** Le WFA  $\mathcal{A}$  calcule pour chaque mot  $w \in \Sigma^*$  un poids  $\llbracket \mathcal{A} \rrbracket(w)$  défini par

$$\llbracket \mathcal{A} \rrbracket(w) = \max_{\pi \in \Pi_{\mathcal{A}}(w)} \delta(\pi).$$

Ce poids est le maximum entre les poids des chemins étiquetés par  $w$  débutant dans un état initial et terminant dans un état final. On définit un *ensemble de reconnaissance*  $\mathbb{J} \subset \mathbb{K}$  et on convient que le mot  $w$  sera reconnu par  $\mathcal{A}$  lorsque  $\llbracket \mathcal{A} \rrbracket(w) \in \mathbb{J}$  (figure 15).

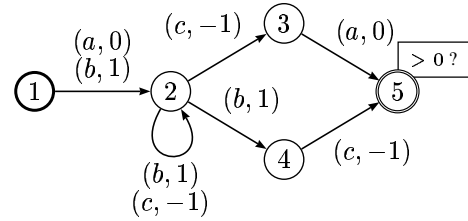


FIG. 15 – Automate pondéré  $\mathcal{A}_2$ . Un automate pondéré fait correspondre à chaque mot  $w$  un poids qui est égal au poids le plus grand le long des chemins étiquetés par  $w$  conduisant d’un état initial à un état final. Pour l’automate pondéré  $\mathcal{A}_2$ , ce poids  $\llbracket \mathcal{A}_2 \rrbracket(w)$  est la différence du nombre de  $b$  et du nombre de  $c$  contenus dans  $w$ . Avec un ensemble de reconnaissance limité aux entiers positifs, cet automate reconnaît le langage non-rationnel  $\mathcal{L}_2 = \{w \in \mathcal{L}_1 \mid |w|_b > |w|_c\}$  où  $\mathcal{L}_1 = (a|b)(b|c)^*(ca|bc)$  est le langage reconnu par l’automate fini  $\mathcal{A}_1$  (figure 7).

Si l’automate pondéré n’a pas d’ $\varepsilon$ -transition et si, pour tout état  $q_1$  et pour tout caractère  $\alpha$ , il existe au plus un état  $q_2$  avec une transition (non vide)  $(q_1, q_2, \delta) \in \Delta$ , le WFA est *déterministe*. Il n’est pas toujours possible de déterminer un automate pondéré [Buchsbaum et al., 2001], ce qui compromet des méthodes de simulation où l’on se souvient d’un seul état. Les algorithmes existants de simulation logicielle reviennent ainsi à explorer les différents chemins possibles, comme celui proposé par Eramian [Eramian, 2002].

Les automates pondérés sont aussi utilisés en compression d’image ou en reconnaissance de la voix [Culik II et Kari, 1993, Mohri, 1997b].

### 2.3.5 Semi-anneaux particuliers

Les automates finis ne sont qu'un cas particulier d'automates pondérés sur le semi-anneau booléen  $(\{F, V\}, \vee, \wedge)$  avec l'ensemble de reconnaissance  $\mathbb{J} = \{V\}$ . Dans ce cas, un mot  $w$  est reconnu par l'automate s'il existe un chemin étiqueté par  $w$  depuis un état initial jusqu'à un état final. On peut coder par automates finis les expressions régulières telles que les motifs PROSITE précédemment évoqués.

Les autres semi-anneaux couramment utilisés sont  $(\mathbb{R}^+, +, \times)$  (calcul des probabilités),  $(\mathbb{R} \cup \{-\infty\}, \oplus_{\log}, +)$  (passage au logarithme) et  $(\mathbb{R} \cup \{-\infty\}, \max, +)$  (approximation de Viterbi).

On peut simplement considérer les automates sur le semi-anneau  $(\mathbb{Z} \cup \{-\infty\}, \max, +)$  et l'ensemble de reconnaissance sera sous la forme  $\mathbb{J} = \{x \in \mathbb{Z} \mid x \geq s_0\}$ , où  $s_0$  est le seuil à partir duquel les données sont reconnues. Ainsi, avec le seuil  $s_0 = 0$ , l'automate  $\mathcal{A}_2$  représenté en figure 15 reconnaît le sous-ensemble des mots de  $\mathcal{L}_1$  contenant plus de  $b$  que de  $c$ . Ce langage  $\mathcal{L}_2$  n'est d'ailleurs pas rationnel.

## 2.4 Recherches de motifs

### 2.4.1 Recherche continue de motifs

Le tableau 4 récapitule quelques possibilités que nous avons vues pour modéliser un motif. Étant donné une modélisation, le but est alors de rechercher dans un grand ensemble – typiquement des nouveaux génomes – les sous-séquences vérifiant ce modèle [Crochemore et Hancart, 1997].

**Définition 4** Soit  $\mathcal{B}$  un mot appelé la banque de données, et  $\mathcal{L}$  un langage représentant un motif. Le problème de la recherche continue de motifs (continuous pattern matching) est de trouver tous les facteurs  $v$  de  $\mathcal{B}$  tels que  $v \in \mathcal{L}$ .

Ce problème pouvant avoir  $\mathcal{O}(n^2)$  solutions (comme pour l'expression rationnelle  $a^*$  dans le mot  $a^n$ ), on se limitera à trouver toutes les positions dans le mot initial terminant les facteurs reconnus (figure 16), à savoir déterminer l'ensemble

$$\text{Pos}(\mathcal{L}, \mathcal{B}) = \{ j \in [1; n] \mid \exists i \in [1; j], \mathcal{B}_i \mathcal{B}_{i+1} \dots \mathcal{B}_j \in \mathcal{L} \}.$$

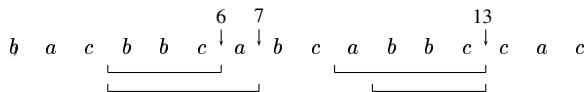


FIG. 16 – Recherche continue de motifs selon le langage  $\mathcal{L}$  défini par l'automate  $\mathcal{A}_2$  (Fig. 15) et l'ensemble de reconnaissance  $\mathbb{J} = \{x \in \mathbb{Z} \mid x > 0\}$  dans la banque  $\mathcal{B} = bacbbcabbbccac$ . Seules les positions terminant les facteurs reconnus sont conservées : l'ensemble  $\text{Pos}(\mathcal{L}, \mathcal{B})$  est ici  $\{6, 7, 13\}$ . Les deux facteurs  $abbc$  et  $bbc$  ont la même position finale 13 : une recherche complémentaire peut les distinguer.

### 2.4.2 Heuristique de filtrage et mesures de qualité

Lors du balayage de grandes banques de données, une recherche de motifs opère un *filtrage* qui retourne les positions des meilleures séquences (figure 17) : certains objets (des sous-séquences) sont sélectionnés parmi un grand ensemble (la banque).

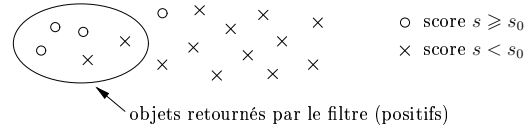


FIG. 17 – Schéma d'une heuristique retournant quelques objets (positifs) parmi un ensemble plus grand. Ici l'heuristique accepte deux objets en trop (faux positifs) et oublie un objet (faux négatif).

Les objets retournés se répartissent en  $T^\oplus$  vrais positifs (objets qui sont réellement similaires) et  $F^\oplus$  faux positifs (à cause de l'heuristique). De même, les objets éliminés se répartissent en  $T^\ominus$  vrais négatifs et  $F^\ominus$  faux négatifs. La sélectivité  $S_\ell = (T^\oplus + F^\oplus) / (T^\oplus + F^\oplus + T^\ominus + F^\ominus)$  mesure le taux de filtrage, mais il est plus intéressant de mesurer la *qualité* du filtrage par les mesures  $S_n = T^\oplus / (T^\oplus + F^\oplus)$  et  $S_p = T^\oplus / (T^\oplus + F^\oplus)$ . La *sensibilité*  $S_n$  est la fraction des résultats positifs qui a été retournée, et la *sélectivité*  $S_p$  celle des résultats intéressants parmi les résultats retournés.

## 3 IMPLÉMENTATIONS

Cette partie passe en revue diverses solutions pour résoudre les recherches de motifs exposés en partie 2. On connaît de nombreux algorithmes efficaces (3.1) pour rechercher un motif simple sans erreurs. La recherche avec erreurs par programmation dynamique est assez lourde (3.2) : des heuristiques à base de graines, dont BLAST, sont aujourd'hui massivement utilisées par les biologistes (3.3). Enfin, nous présentons les recherches de motifs plus généraux par modèles (3.4).

Dans chacune des parties, nous indiquons comment les algorithmes peuvent être accélérés par des architectures spécialisées utilisant des circuits FPGA. Les FPGA sont présentés en annexe A.

### 3.1 Recherches exactes et structures d'indexation

#### 3.1.1 Recherches sans pré-traitements de la banque

Lorsque  $\mathcal{L}$  est réduit à un singleton  $\{v\}$ , la recherche de motifs devient un problème de recherche d'un facteur fixé  $v$  dans un mot de longueur  $n$ .

Il est possible d'utiliser une approche *brute* d'énumération de la séquence : une approche naïve pour la recherche du motif  $v$  dans un texte de longueur  $n$  donne un algorithme en temps  $\mathcal{O}(n \times |v|)$ .

L'algorithme de Knuth-Morris-Pratt [Knuth et al., 1977] effectue une recherche du motif dans le texte en temps linéaire ( $\mathcal{O}(n + |v|)$ ). Un pré-traitement du motif  $v$  (réalisé en temps  $\mathcal{O}(|v|)$ ) permet de calculer une fonction de décalage pour tous les préfixes de  $v$ . La comparaison du

$\mathcal{L} = \{v\}$	recherche exacte d'un motif
$\mathcal{L} = \{v_1, v_2, \dots, v_q\}$	recherche exacte parmi plusieurs mots (dictionnaire)
$\mathcal{L} = \{w \in \Sigma^* \mid \mathcal{H}(w, v) \geq h_0\}$	recherche d'un motif par similarité
$\mathcal{L} = \{w \in \Sigma^* \mid \llbracket \mathcal{A} \rrbracket(w) \geq s_0\}$	recherche d'un motif selon un automate ou un modèle

TAB. 4 – Recherches de motifs.

motif est ensuite réalisée sur le motif et le texte par un parcours de la gauche vers la droite. Une fonction de décalage est utilisée à chaque fois que les lettres à comparer ne concordent plus (figure 18).

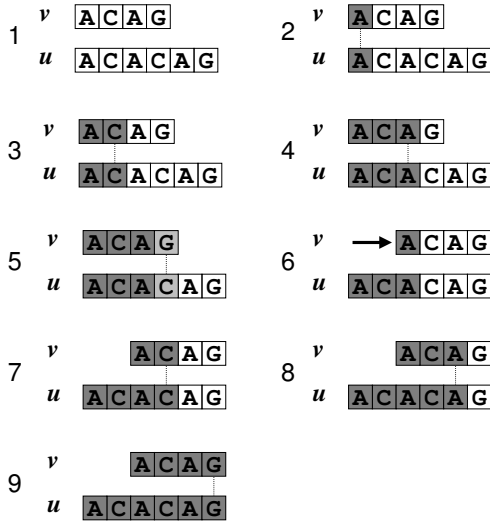


FIG. 18 – Algorithme de Knuth-Morris-Pratt appliqué à la recherche du motif  $v = ACAG$  dans le texte  $u = ACACAG$ . Les étapes 2-4 correspondent à la comparaison du préfixe de  $v$  et du texte par lecture d'un caractère lors de chaque étape. A l'étape 5, les caractères comparés dans  $u$  et  $v$  ne correspondent pas : le motif est alors décalé à l'étape 6 de sorte que le préfixe déjà comparé de  $v$  (A) soit égal au suffixe de  $u$ .

Un deuxième algorithme, l'algorithme de Boyer-Moore, réalise une recherche du motif  $v$  dans un texte de longueur  $n$  en temps moyen  $\mathcal{O}(\frac{n}{|v|})$ . La banque est parcourue de gauche à droite comme pour l'algorithme de Knuth-Morris-Pratt, mais cette fois-ci le motif est comparé de la droite vers la gauche. L'algorithme original possède l'inconvénient d'être quadratique dans le pire cas, mais une extension [Galil, 1979] donne une complexité en  $\mathcal{O}(n + |v|)$ . Une version simplifiée de cet algorithme est par exemple utilisée dans le logiciel Word.

Lorsque le motif à rechercher est un dictionnaire avec plusieurs mots  $\{v_1, v_2, \dots, v_q\}$ , les approches par automates sont les plus fréquentes.

L'algorithme de [Aho et Corasick, 1975], en particulier, permet de construire un automate des préfixes avec une fonction de transition qui simule la fonction de décalage de l'algorithme de Knuth-Morris-Pratt (figure 19). C'est en ce sens une extension de l'algorithme de Knuth-Morris-Pratt pour la recherche parallèle de plusieurs motifs.

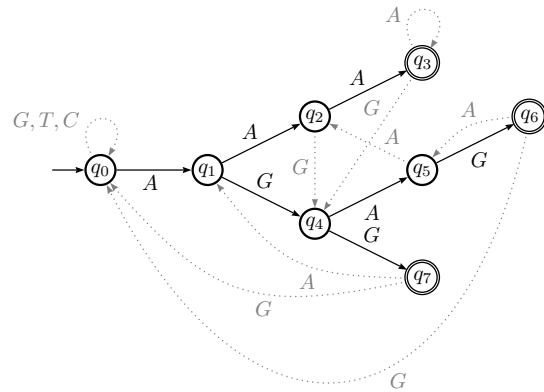


FIG. 19 – Automate de Aho-Corasick du motif défini par l'ensemble des mots  $\{v_1 = AAA, v_2 = AGAG, v_3 = AGG\}$ . L'arbre des préfixes (représenté en trait continu) est d'abord construit, puis complété à l'aide d'une fonction de bord afin d'ajouter les transitions nécessaires pour rendre l'automate *complet*.

Ces approches peuvent être accélérées par du parallélisme au niveau du bit [Wu et Manber, 1992].

### 3.1.2 Recherches avec pré-traitements de la banque

De nombreux algorithmes commencent par traiter la banque de séquences et en construire une certaine représentation indexée. Les pré-traitements sont souvent longs, bien qu'en  $\mathcal{O}(n)$ , et simplifient les requêtes ultérieures, idéalement en  $\mathcal{O}(|v|)$ . [Navarro et Raffinot, 2002] et [Marsan, 2002] proposent une présentation détaillée de ces algorithmes.

Des structures comme les arbres à suffixes [Gusfield, 1997] ou les DAWG (*directed acyclic word graphs, figure 21*) [Blumer et al., 1985] donnent de tels résultats sur de nombreux problèmes. Un arbre des suffixes est l'automate fini canonique reconnaissant tous les suffixes de la banque (figure 20). Il est aussi possible de construire des arbres des suffixes généralisés pour un ensemble de mots.

L'oracle des facteurs est utilisé dans FORRepeats [Lefebvre et al., 1999]. Cet oracle reconnaît au moins tous les facteurs de la banque (figure 22).

L'approche indexée, bien qu'algorithmiquement plus satisfaisante, reste souvent limitée à des cas de reconnaissance exacte et s'étend mal à des classes de motifs plus complexes, notamment pour la gestion des erreurs. Par exemple, chercher un mot  $v$  avec  $k$  erreurs de substitution dans l'arbre des suffixes demande l'exploration d'au moins  $\mathcal{O}(|\Sigma|^k)$  branches différentes.

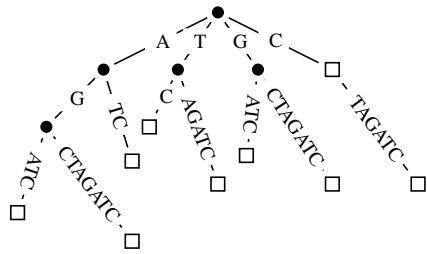


FIG. 20 – Arbre des suffixes pour la chaîne AGCTAGATC. Les nœuds marqués par des carrés sont des suffixes de la chaîne initiale. Certains arcs ont des labels avec plusieurs caractères : l’arbre est compressé. En mémoire, les arcs sont représentés par un couple de positions, comme (7, 10) pour l’arc ATC (en bas à gauche), afin que l’arbre garde une taille linéaire.

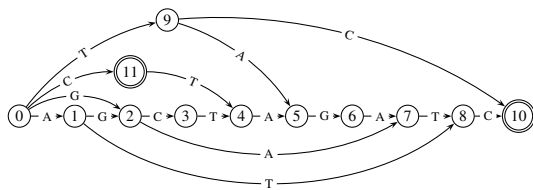


FIG. 21 – DAWG pour la chaîne AGCTAGATC.

Enfin, les banques de données étant régulièrement mises à jour, les pré-traitements demandés peuvent être fréquents et finalement coûteux en temps.

### 3.2 Programmation dynamique et architectures systoliques

#### 3.2.1 Implémentations logicielles

Une matrice de programmation dynamique comporte  $\mathcal{O}(mn)$  cellules, mais les  $m$  cellules d’une même anti-diagonale peuvent être calculées simultanément (figure 11, à droite). Un espace  $\mathcal{O}(m + n)$  est donc suffisant pour calculer toute la matrice, mais il faut toujours un temps  $\mathcal{O}(mn)$  pour calculer la similarité globale.

La première borne sous-quadratique fut atteinte par Masek et Paterson en 1980. Leur algorithme utilise une décomposition par blocs et calcule la similarité globale en temps  $\mathcal{O}(n^2/\log n)$  [Masek et Paterson, 1980]. Cet algorithme s’applique à des matrices de score rationnelles. En 2002, Crochemore, Landau et Ziv-Ukelson ont obtenu un algorithme plus général travaillant sur une factorisation des séquences en  $\mathcal{O}(hn^2/\log n)$  où  $h$  est l’entropie de la séquence : il est en pratique plus rapide lorsque les séquences sont compressibles [Crochemore et al., 2003].

#### 3.2.2 Architectures systoliques

En suivant la méthodologie de Kung [Kung et Leiserson, 1980], la matrice de program-

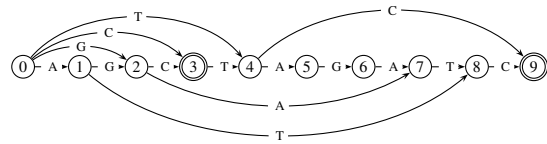


FIG. 22 – Oracle des facteurs pour la chaîne AGCTAGATC. L’oracle reconnaît tous les facteurs de la chaîne initiale, tout comme quelques mots en plus tels que GCTC.

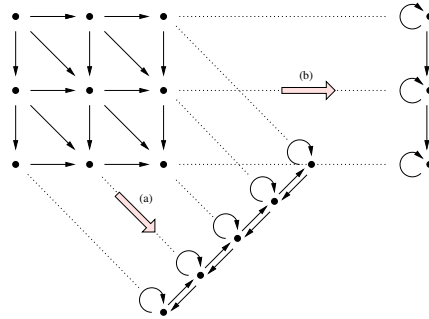


FIG. 23 – Projections d’une matrice de calcul sur une architecture systolique. La projection diagonale (a) conduit à une architecture bidirectionnelle. La projection horizontale (b) conduit à une architecture unidirectionnelle.

mation dynamique peut être projetée sur une architecture systolique (figure 23). En 1985, Lipton et Lopresti proposèrent une architecture bidirectionnelle [Lipton et Lopresti, 1985] dans laquelle les deux séquences se propagent dans des directions opposées (figure 23a).

Des architectures unidirectionnelles ont été proposées dans [Chow et al., 1991] puis [Hoang, 1993]. Une séquence est chargée dans le réseau, puis l’autre est introduite (figure 23b). Au minimum  $\min(m + 1, n + 1)$  cellules sont nécessaires et le calcul se fait en  $\mathcal{O}(m + n)$  cycles. En comparaison, un réseau bidirectionnel demande  $m + n - 1$  cellules : en général, le réseau unidirectionnel est plus performant. Cependant, on peut utiliser un réseau bidirectionnel pour comparer deux grandes séquences similaires pour un alignement restant proche de la diagonale principale (figure 24), comme dans l’algorithme *k-band* de Fickett [Fickett, 1984, Andonov et al., 2003].

#### 3.2.3 Retour en arrière

Les équations NW/SW donnent seulement le meilleur score d’alignement global ou local. Bien qu’il soit suffisant dans certaines applications, on souhaite pouvoir exhiber l’alignement qui a conduit à ce score. Il faut alors, dans chaque cellule de la matrice, tracer l’information en se souvenant d’où venait le meilleur score. Cette infor-

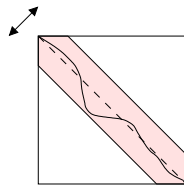


FIG. 24 – Pour aligner globalement deux longues séquences similaires, on peut généralement se restreindre au calcul de quelques cellules autour de la diagonale.

mation à conserver, montrée par les flèches continues sur la figure 10, prend une des valeurs  $\leftarrow$ ,  $\swarrow$ , or  $\uparrow$ .

Dans les implémentations matérielles, la cellule systolique peut stocker ces valeurs dans une pile locale, et une phase de retour en arrière suit celle de calcul du score [Hoang, 1993].

### 3.2.4 Distance $d_{12}$ et encodage modulo

Si on considère la distance  $d_{12}$  définie à la fin de la partie 2.2.1, on a la propriété suivante, établie dans [Lipton et Lopresti, 1985] :

$$\begin{cases} H(i, j) = H(i-1, j) \pm 1 \\ H(i, j) = H(i, j-1) \pm 1 \end{cases}$$

$$\text{Donc } H(i, j) - H(i-1, j-1) \in \{-2, 0\}$$

L'équation NW (1) peut alors se réécrire en :

$\forall i, j, ij \neq 0$  :

$$H(i, j) = \max \begin{cases} H(i-1, j-1) \\ \text{si } H(i-1, j) = H(i-1, j-1) + 1 \\ \text{ou } H(i, j-1) = H(i-1, j-1) + 1 \\ \text{ou } (x_i = y_j) \\ H(i-1, j-1) - 2 \\ \text{sinon} \end{cases}$$

La valeur calculée dans chaque cellule se représente donc modulo 4, et même seulement avec le deuxième bit : il n'y a ainsi jamais de dépassement de capacité.

Pour un alignement global NW, l'utilisation de la distance  $d_{12}$  réduit le problème à la recherche de la plus longue sous-séquence commune (*longest common subsequence* (LCS)), problème qui peut s'accélérer par du parallélisme au niveau du bit [Crochemore et al., 2001, Dydel et Bala, 2004].

### 3.2.5 Implémentations FPGA

À la suite de [Lopresti, 1987], de nombreuses implémentations des architectures systoliques utilisent la distance  $d_{12}$  avec l'encodage modulo. D'autres architectures implémentent une équation SW générique [Chow et al., 1991] (voir [Lavenier et Giraud, 2005] pour une présentation plus détaillée). Van Court et Herbordt ont proposé en 2004 une modélisation systématique [Court et Herbordt, 2004].

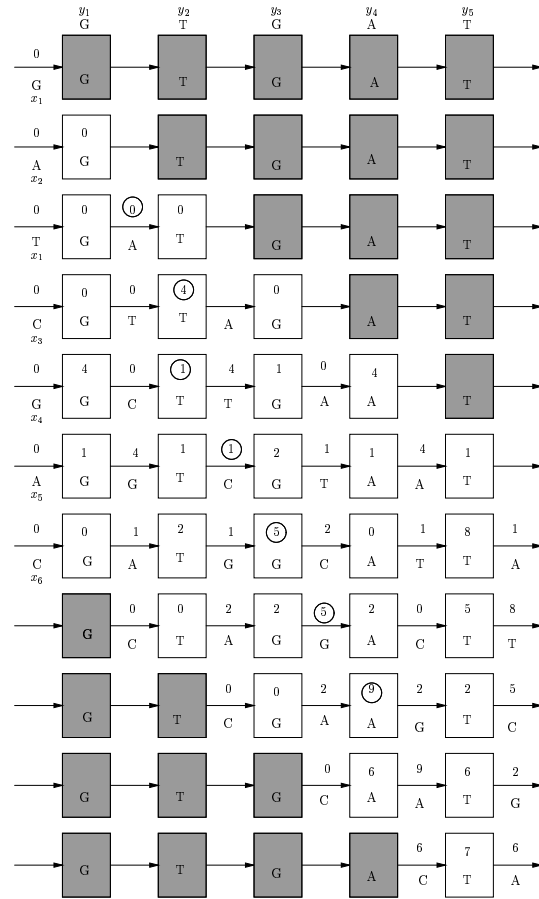


FIG. 25 – Fonctionnement d'une architecture systolique unidirectionnelle similaire à celle proposée par Hoang [Hoang, 1993] comparant  $X=ATCGAC$  à  $Y=GTGAT$ . Cet exemple est identique à la matrice de programmation dynamique utilisant SW de la figure 10b. La chaîne  $Y$  est supposée être chargée avant le début du calcul. Les valeurs encadrées sont celles de l'alignement optimal : le meilleur score est 9.

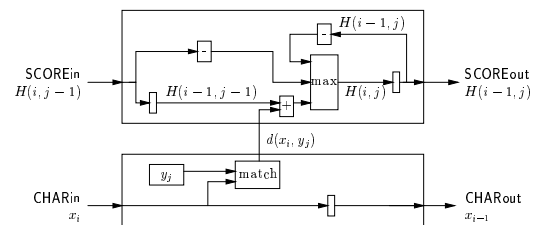


FIG. 26 – Détail d'une cellule systolique. Les équations de programmation dynamique ont besoin du coût  $d(x_i, y_j)$ , mais non des caractères  $x_i$  et  $y_j$ . On peut donc séparer la phase de comparaison (calcul de  $d(x_i, y_j)$ , en bas) et la phase de programmation dynamique (calcul récursif de  $H(i, j)$ , en haut) [Court et Herbordt, 2004]. Ce schéma ne montre pas la logique de contrôle nécessaire à l'initialisation (mémorisation de  $CHAR_{in}$  dans  $y_j$ ) et à la finalisation du calcul (avec un éventuel retour en arrière).

Ces implémentations matérielles (ou logicielles) doivent balayer toute la banque de séquences, un accès particulièrement rapide aux données doit être assuré.

### 3.3 Heuristiques à bases de graines

Les relations de programmation dynamique exposées en partie 2.2 et implémentées dans la partie précédente calculent exactement des distances de similarité entre deux séquences. Cependant, leur temps de calcul est quadratique, ce qui empêche de les utiliser pour comparer des chaînes très grandes comme des génomes entiers de milliards de bases.

Il est donc intéressant d'effectuer un pré-filtrage des séquences, filtrage auquel s'applique les mesures de qualité vues en partie 2.4.2. Des heuristiques pour ce type de recherche ont été proposées il y a une quinzaine d'années par [Pearson et Lipman, 1988] (FASTA) puis [Altschul et al., 1990] (BLAST). Bien qu'aujourd'hui d'autres heuristiques sont plus performantes (partie 3.3.2), le programme BLAST reste la référence pour les biologistes (Table 5).

#### 3.3.1 Blast, une heuristique en 3 étapes

Le principe des heuristiques accélérant les calculs de programmation dynamique est d'omettre certaines parties de la matrice. Elles supposent que, la plupart du temps, les alignements significatifs comprennent des *graines*, c'est-à-dire des petits mots exactement conservés, comme GA dans la figure 10b (page 7). Ces graines représentent des diagonales dans la matrice de programmation dynamique. Les calculs complets de la matrice sont effectués seulement au voisinage de ces graines. L'heuristique à base de graines utilisée par Blast a trois étapes (figure 27) :

- L'étape 1 recherche des graines exactes (par défaut de taille  $f = 11$  pour les chaînes nucléiques) qui apparaissent dans les deux chaînes.
  - L'étape 2 essaie d'étendre chaque graine en admettant un nombre limité d'erreurs de substitutions. Puisque les insertions comme les délétions ne sont pas considérées, l'extension est toujours le long de la même diagonale. Seules les graines qui ont été étendues avec un score suffisant sont conservées.
  - L'étape 3 calcule la matrice de programmation dynamique seulement au voisinage des graines conservées.
- La taille 11 est un compromis entre sensibilité et efficacité : plus la taille  $f$  est petite et plus les graines sont nombreuses, d'où une bonne sensibilité. Cependant, un nombre trop élevé de graines sélectionnées à l'étape 1 ralentit l'étape 2.

#### 3.3.2 Améliorations des heuristiques

Des techniques à l'aide de *graines espacées* améliorent la sensibilité des heuristiques d'alignement pour une taille  $f$  donnée.

L'idée consiste à ne plus rechercher des mots contigus dans un texte mais plutôt des mots dits *espacés*. Par exemple si l'on recherche les mots de deux lettres avec un espacement d'une lettre, nous obtenons sur le texte GTGATCT les mots espacés suivants : G-G, T-A, G-T, A-

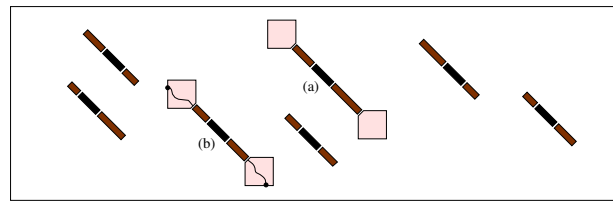


FIG. 27 – Les 3 étapes de BLAST. 1. Localisation des graines (noir). 2. Extension des graines en acceptant quelques substitutions (gris). La majorité des graines ne sont pas conservées. 3. Calcul de la matrice au voisinage d'un tout petit nombre de séquences (gris clair). Ici seule la graine (b) mène à une séquence positive.

C, et T-T. La graine possède alors un symbole donné par #-# pour représenter ces mots : le symbole # représente la position d'une lettre considérée tandis que le symbole - est synonyme de *joker*. Bien entendu des symboles plus complexes comme par exemple #-#-## sont possibles. Ainsi la graine #-#-## appliquée au texte GTGATCT donne les mots espacés { G-G-TC, T-A-CT } ce qui permet de détecter GTGATCT comme étant similaire à GCGTTCA (les deux textes possèdent le même mot espacé commun G-G-TC). En contrepartie GTGATCT ne sera pas détecté comme étant similaire à GTGTACT (aucun mot espacé commun selon le symbole #-#-##).

Afin d'améliorer les méthodes d'alignement, l'idée consiste à concevoir un ou plusieurs symboles de graines sur l'alphabet { #, - } qui soient plus efficaces que les simples mots composés uniquement sur l'alphabet { # }. En effet une graine espacée, si elle est bien conçue, a plus de chances d'avoir au moins une occurrence dans un alignement significatif qu'une graine contiguë, et ce pour le même nombre  $f$  de lettres.

Le choix des symboles des graines a d'abord été fait de manière purement aléatoire [Califano et Rigoutsos, 1993, Buhler, 2002]. Des auteurs ont proposé un pré-calcul pour sélectionner les symboles de graines les plus efficaces sur des modèles d'alignement. Les meilleures graines sont celles qui détectent soit le plus d'alignements possibles [Ma et al., 2002], soit tous les alignements d'une taille minimale donnée et avec un nombre maximum de substitutions donné [Burkhardt et Kärkkäinen, 2001].

Ce choix induit des particularités dans la forme des graines. Ainsi des graines comme par exemple ###-##-##-## détectent un maximum d'alignements d'une distribution donnée sans pour autant les détecter tous : leurs symboles semblent plutôt aléatoires et peu corrélés. La probabilité de détection d'alignements dépend de la similarité des séquences (figure 28).

Au contraire, des graines destinées à trouver tous les alignements d'une taille minimale donnée avec un nombre d'erreurs maximal fixé ont souvent un symbole régulier et donc corrélé. Par exemple, la graine de symbole ###-##-##-## est la seule graine possédant 12 symboles # capable de trouver tous les alignements d'une taille minimale 25 avec au plus deux erreurs de substi-

	Requête nucléique ( $\Sigma_4$ )	Requête protéique ( $\Sigma_{20}$ )
Banque nucléique ( $\Sigma_4$ )	<i>blastn</i> / <i>tblastx</i>	<i>blastn</i>
Banque protéique ( $\Sigma_{20}$ )	<i>blastx</i>	<i>blastp</i>

TAB. 5 – Versions de BLAST. Alors que le programme *blastn* compare directement deux séquences nucléique, le programme *tblastx* compare les traductions protéiques suivant les 6 phases de lecture.

tution : on peut alors remarquer la périodicité du sous-symbole `###-#--`.

Les *graines espacées* ont été étendues vers d'autres modèles comme les *graines vecteurs* [Brejova et al., 2003] ou les *graines sous-ensemble* [Kucherov et al., 2004], permettant d'améliorer encore la sensibilité du filtrage.

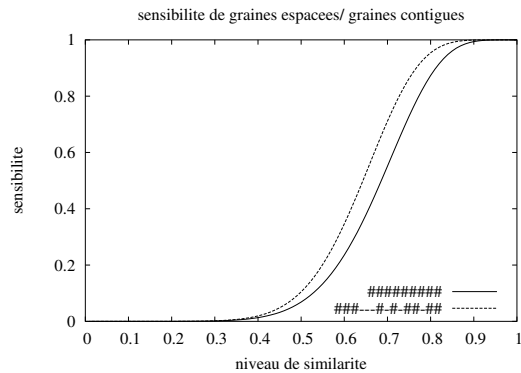


FIG. 28 – Sensibilité de graines en fonction de la similarité des deux séquences. Entre 50 et 80% de similarité, la graine espacée est 10 à 20% plus sensible que la graine contiguë.

### 3.3.3 Indexation des graines

La recherche des graines prend de 70% à 90% des temps de calcul de BLAST [Krishnamurthy et al., 2004, Muriki et al., 2005]. Cette étape est en fait un *problème d'appartenance* [Carter et al., 1978] : une graine prise à une certaine position dans une des séquences appartient-elle à l'ensemble des graines de la seconde séquence ?

La première solution est de conserver un tableau de taille  $|\Sigma|^w$  et de marquer dans ce tableau toutes les  $n$  graines d'une séquence. Typiquement, chaque case du tableau contient un pointeur vers la position ou les positions où la graine apparaît. Comme ce tableau peut vite devenir grand, diverses méthodes de compression ont été proposées [Chang, 2004].

La seconde solution est d'accepter quelques faux positifs dans le problème d'appartenance, ceux-ci étant certainement éliminés aux étapes ultérieures. Les graines peuvent donc être conservées dans une table de hachage de taille  $N$  (Figure 29), ce qui conduit à une probabilité de faux positifs de  $P(F^\oplus) = 1 - (1 - \frac{1}{N})^n$ . On peut aussi utiliser des techniques plus efficaces telles que les filtres Bloom [Bloom, 1970]. L'interrogation d'une table de hachage de taille  $M$  est remplacée par  $d$  interrogations dans une table de taille  $N \leq M$ , chaque interrogation ayant sa propre

fonction de hachage. La graine est supposée présente si toutes les interrogations ont fonctionné. La probabilité de faux positifs devient alors  $P(F^\oplus) = (1 - (1 - \frac{d}{N})^n)^d$ .

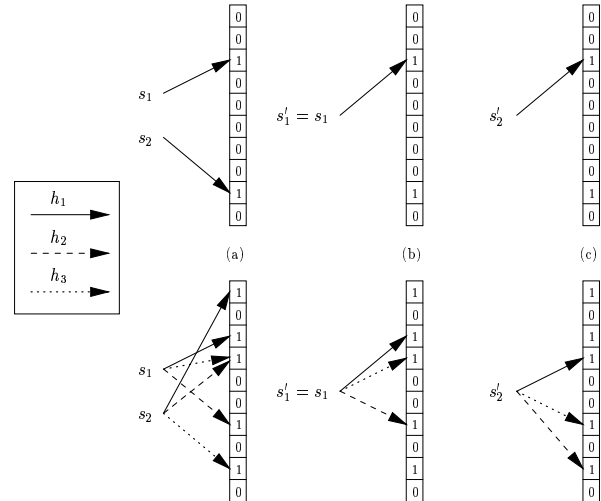


FIG. 29 – Tables de hachage (haut) et filtres Bloom (bas). Durant la phase d'initialisation (a), des clés hachées correspondant aux graines  $s_1$  et  $s_2$  sont enregistrées dans la table. Durant la phase de lecture (b), la graine  $s'_1 = s_1$  est correctement reconnue. La phase de lecture peut occasionner certains faux positifs : la graine  $s'_2$  est reconnue alors qu'elle ne figurait pas dans les graines originales (c).

### 3.3.4 Accélération matérielle

Le programme BLAST étant une référence pour les biologistes, de nombreuses équipes tentent de l'accélérer. La décomposition en 3 étapes est peu souvent remise en question, et on oublie alors que l'algorithme utilisé par BLAST est lui-même une heuristique initialement destinée à accélérer les traitements. On peut citer une implémentation sur cluster de PC, mpiBLAST [Darling et al., 2003], ainsi que plusieurs architectures matérielles qui utilisent une heuristique à base de graines [Singh et al., 1993, Chang, 2004, Muriki et al., 2005].

La phase d'indexation des graines est particulièrement intéressante à paralléliser, notamment pour les filtres Bloom qui calculent différentes fonctions de hachage simultanément. En répliquant la table de hachage, on peut faire des accès concurrents, et certaines mémoires disponibles (comme les BRAMs de Xilinx) permettent un double accès par cycle [Dharmapurikar et al., 2004, Krishnamurthy et al., 2004].

D'autres projets ne suivent pas exactement les étapes

de Blast. On peut citer une heuristique à base de graines similaire à l'étape 2 de Blast sur la plateforme Rdisk [Guytant, 2004, Lavenier et al., 2003], ainsi que l'approche de Gardner-Stephen et Knowles qui ont développé conjointement un nouvel algorithme, DASH, et une plateforme FPGA pour l'exécuter [Gardner-Stephen et Knowles, 2004, Knowles et Gardner-Stephen, 2004].

### 3.4 Autres implémentations

Nous évoquons ici l'implémentation des autres modélisations présentées en parties 2.3 et 2.3. Profils, expressions régulières, et automates se simulent exhaustivement en logiciel : ces calculs vont fortement s'accélérer sur des architectures spécialisées.

#### 3.4.1 Profils et modèles

Parmi les outils logiciels qui recherchent des expressions rationnelles, on peut citer *agrep* qui utilise le parallélisme au niveau du bit pour chercher des motifs à l'intérieur de longs fichiers [Wu et Manber, 1992]. Ce programme recherche des expressions rationnelles, mais il est possible de tolérer jusqu'à quatre erreurs. D'autres programmes sont plus spécifiquement utilisés en biologie tels que ScanPROSITE [Gattiker et al., 2002], PATTINPROT [PATTINPROT, ] ou PATTERNp [Cockwell et Giles, 1989]. Certains permettent d'effectuer des recherches avec erreurs tout en se limitant à des syntaxes de type PROSITE qui sont loin d'utiliser les possibilités des langages rationnels.

Les profils consensus peuvent être aisément réalisés en FPGA avec un score circulant à travers un réseau de cellules. Hughey a proposé en 1993 une implémentation des modèles de Markov utilisant une grille de processeurs [Hughey, 1993]. Mosanya et Sanchez ont réalisé sur FPGA un modèle similaire (profil généralisé) avec un réseau systolique utilisant de l'arithmétique en ligne [Mosanya et Sanchez, 1999]. Une autre architecture FPGA fut présentée par Gupta en 2004 [Gupta, 2004].

#### 3.4.2 Automates

Pour simuler un *automate fini* à  $r$  états et  $t$  transitions sur un mot de longueur  $n$ , on peut commencer par le déterminer, ce qui risque de produire un nombre d'états exponentiel. D'autres méthodes, directes, utilisent un vecteur de taille  $r$  et aboutissent à une complexité en  $\mathcal{O}(r^2n)$  ou  $\mathcal{O}(tn)$ . Quant aux *automates pondérés*, une simulation directe est nécessaire car ils ne sont pas tous déterminisables [Buchbaum et al., 2001]. Mark G. Eramian a proposé un algorithme [Eramian, 2002] résolvant le problème en temps  $\mathcal{O}(tn)$ .

Les recherches de motifs par automates s'accélèrent fortement sur FPGA, notamment parce qu'il est possible de réaliser directement le *non-déterminisme*. On calcule simultanément toutes les transitions et les états d'un automate fini ou pondéré avec un *encodage linéaire*, c'est-à-dire une matérialisation dans laquelle une cellule matérielle correspond à chaque état (figure 3.4.2). Les automates finis puis pondérés ont été

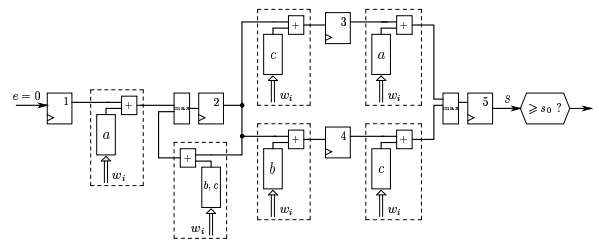


FIG. 30 – Encodage linéaire pour l'automate pondéré  $\mathcal{A}_2$  (figure 15). Chaque état est matérialisé par un registre à  $p$  bits, et chaque transition est une fonction  $\langle 5 \mapsto 1 \rangle$  suivie d'un additionneur.

ainsi implémentés sur des architectures reconfigurables [Sidhu et Prasanna, 2001, Giraud et Lavenier, 2004].

#### 3.4.3 Motifs algébriques et autres

Lorsque le motif est défini par une grammaire algébrique quelconque, on peut utiliser directement un automate à pile [Hopcroft et Ullman, 1969], mais à notre connaissance cela n'a jamais été réalisé pour de vraies applications, en logiciel ou matériel. Une architecture FPGA pour une recherche de grammaires algébriques a été proposée dans la thèse de Ciressan [Ciressan, 2002], mais il n'y a pas encore eu d'implémentation FPGA de recherche de motifs algébriques dans des séquences biologiques. En effet, la recherche des motifs algébriques est quadratique par rapport à la taille des séquences et peut concerner des dépendances arbitrairement espacées (tableau 3) : un reconnaissseur générique peut être difficilement efficace.

Un certain nombre de travaux se sont donc concentrés sur quelques structures non régulières telles que les *répétitions* ou les *palindromes*. Ces structures peuvent être facilement détectées logiquement (par exemple par des arbres des suffixes) ou matériellement par des réseaux systoliques [Kung et Leiserson, 1980, Quinton et Robert, 1989]. Conti a proposé une implémentation sur FPGA qui tolère les erreurs de substitution [Conti et al., 2004].

Une approche plus générique a été formalisée par les grammaires SVG (*string variable grammars*) définies par Searls et qui ont une expressivité située entre les grammaires algébriques et les grammaires indexées. Une implémentation des SVG, GenLang, utilise Prolog pour explorer les dérivations possibles à partir des règles de la grammaire [Searls, 1993b].

## 4 CONCLUSIONS ET PERSPECTIVES

Rechercher un motif signifie retrouver dans de nouvelles séquences des facteurs appartenant à un langage connu. Dans sa forme la plus simple, un motif est un simple mot, mais il peut être un ensemble de mots, ensemble généré par des opérations d'édition à partir d'un mot initial ou directement par un modèle dont la construction regroupe plusieurs séquences.

Les structures de données indexant les banques de données sont efficaces pour les problèmes exacts. Lorsqu'on

recherche un motif avec des erreurs d'édition, ou cherche des similarités entre deux séquences et cela demande de calculer une matrice de programmation dynamique. Grâce aux heuristiques à base de graines, il est possible de ne pas calculer toute la matrice.

Les modélisations plus complexes, notamment celles incluant des motifs algébriques, sont encore peu utilisées et difficiles à implémenter. Bien que les langages algébriques ou plus complexes sont indispensables pour modéliser correctement certaines structures présentes dans les séquences, ils rendent la recherche de motifs délicate. Une perspective actuelle de recherche est de combiner les différentes approches en utilisant des motifs relativement simples à rechercher, par exemple par automates, auxquels on ajoute certaines structures plus complexes. Appliquées à de grandes banques de données, les recherches de motifs couplent des défis algorithmiques (par exemple sur les méthodes d'indexation) et calculatoires (structures de données compactes, parallélisme au niveau du bit, architectures reconfigurables et machines spécialisées). Leurs applications ne se limitent pas à la bioinformatique mais s'étendent à l'image, la voix, et aux réseaux.

### Remerciements

Les auteurs remercient les relecteurs et relectrices anonymes pour leurs commentaires de grande qualité.

### BIBLIOGRAPHIE

- [ITR, 2004] (2004). International technology roadmap for semiconductors. ITRS Roadmap Committee.
- [Aho et Corasick, 1975] Aho, A. V. et Corasick, M. J. (1975). Efficient string matching : an aid to bibliographic search. *Commun. ACM*, 18(6) :333–340.
- [Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., et Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3) :403–410.
- [Andonov et al., 2003] Andonov, R., Balev, S., Rajopadhye, S., et Yanev, N. (2003). Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14(9) :944–960.
- [Andonov et al., 2004] Andonov, R., Balev, S., et Yanev, N. (2004). Protein threading problem : From mathematical models to parallel implementations. *INFORMS Journal on Computing, Special Issue on Computational Molecular Biology/Bioinformatics*, 16(4) :393–405.
- [Bloom, 1970] Bloom, B. (1970). Space-time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7) :422–426.
- [Blumer et al., 1985] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M. T., et Seiferas, J. (1985). The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40 :31–55.
- [Brejova et al., 2003] Brejova, B., Brown, D., et Vinar, T. (2003). Vector seeds : an extension to spaced seeds allows substantial improvements in sensitivity and specificity. In Benson, G. et Page, R., editors, *3rd International Workshop in Algorithms in Bioinformatics (WABI), Budapest (Hungary)*, volume 2812 of *Lecture Notes in Computer Science*, pages 39–54.
- [Buchsbaum et al., 2001] Buchsbaum, A. L., Raffaele, G., et Westbrook, J. R. (2001). On the Determinization of Weighted Finite Automata. *SIAM Journal on Computing*, 30(5) :1502 – 1531.
- [Buhler, 2002] Buhler, J. (2002). Provably sensitive indexing strategies for biosequence similarity search. In *Proceedings of the 6th Annual International Conference on Computational Molecular Biology (RECOMB02), Washington, DC (USA)*, pages 90–99. ACM Press.
- [Burkhardt et Kärkkäinen, 2001] Burkhardt, S. et Kärkkäinen, J. (2001). Better filtering with gapped q-grams. In *Proceedings of the 12th Symposium on Combinatorial Pattern Matching, CPM'01*, pages 73–85.
- [Califano et Rigoutsos, 1993] Califano, A. et Rigoutsos, I. (1993). Flash : A fast look-up algorithm for string homology. In *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology*, pages 56–64.
- [Carter et al., 1978] Carter, L., Floyd, R., Gill, J., Markowsky, G., et Wegman, M. (1978). Exact and approximate membership testers. In *Proc. of the 10th ACM Symposium on Theory of Computing (STOC 78)*, pages 59–65.
- [Chang, 2004] Chang, C. (2004). BLAST implementation on BEE2. University of California at Berkeley.
- [Chomsky, 1957] Chomsky, N. (1957). Syntactic structures. *Mouton, La Haye*.
- [Chow et al., 1991] Chow, E., Hunkapiller, T., et Peterson, J. (1991). Biological information signal processor. In *ASAP'91, International Conference on Application Specific Array Processors*, pages 144–160, Barcelona, Spain.
- [Ciressan, 2002] Ciressan, C. (2002). *An FPGA-based syntactic parser for real-life context free grammars*. Thèse de doctorat, EPFL.
- [Cockwell et Giles, 1989] Cockwell, K. Y. et Giles, I. G. (1989). Software tools for motif and pattern scanning : program descriptions including a universal sequence reading algorithm. *Comput. Appl. Biosci.*, 5 :227–232.
- [Conti et al., 2004] Conti, A., Court, T. V., et Herbordt, M. (2004). Processing repetitive sequence structures with mismatches at streaming rate. In *Proc. of FPL 2004*, pages 1080–1083.
- [Court et Herbordt, 2004] Court, T. V. et Herbordt, M. C. (2004). Families of FPGA-based algorithms for approximate string matching. In *Proceedings of ASAP'04*.
- [Crochemore et Hancart, 1997] Crochemore, M. et Hancart, C. (1997). *Handbook of Formal Languages*, chapitre Automata for Matching Patterns. Springer.
- [Crochemore et al., 2001] Crochemore, M., Iliopoulos, C., Pinzon, Y., et Reid, J. (2001). A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6) :279–285.

- [Crochemore et al., 2003] Crochemore, M., Landau, G. M., et Ziv-Ukelson, M. (2003). A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6) :1654–1673.
- [Culik II et Kari, 1993] Culik II, K. et Kari, J. (1993). Image Compression Using Weighted Finite Automata. In *Mathematical Foundations of Computer Science (MFCS 93)*, volume 711 of *Lecture Notes in Computer Science*, pages 392–402.
- [Darling et al., 2003] Darling, A., Carey, L., et Feng, W. (2003). The design, implementation, and evaluation of mpiblast. In *Proc. of CWCE 2003*.
- [Dharmapurikar et al., 2004] Dharmapurikar, S., Attig, M., et Lockwood, J. (2004). Design and implementation of a string matching system for network intrusion detection using FPGA-based Bloom filters. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*.
- [Dydel et Bala, 2004] Dydel, S. et Bala, P. (2004). Large scale protein sequence alignment using FPGA reprogrammable logic devices. In Springer, editor, *Proceedings of FPL'04, LNCS 3203*, pages 23–32.
- [Eramian, 2002] Eramian, M. G. (2002). Efficient simulation of nondeterministic weighted finite automata. In *Fourth Workshop on Descriptive Complexity of Formal Systems (DCFS 02)*.
- [Fickett, 1984] Fickett, J. (1984). Fast optimal alignment. *Nucleic Acids Research*, 12(1) :175–179.
- [Galil, 1979] Galil, Z. (1979). On improving the worst case running time of the boyer-moore string matching algorithm. *Commun. ACM*, 22(9) :505–508.
- [Gardner-Stephen et Knowles, 2004] Gardner-Stephen, P. et Knowles, G. (2004). Dash : Localising dynamic programming for order of magnitude faster, accurate sequence alignment. In *Proceedings of the IEEE Computational Systems Bioinformatics Conference*, pages 732–735, San Francisco, CA.
- [Gattiker et al., 2002] Gattiker, A., Gasteiger, E., et Bairoch, A. (2002). ScanProsite : a reference implementation of a PROSITE scanning tool. *Applied Bioinformatics*, 1 :107–108.
- [Giraud et Lavenier, 2004] Giraud, M. et Lavenier, D. (2004). Linear encoding scheme for weighted finite automata. In *Ninth International Conference on Implementation and Application of Automata (CIAA 2004)*, volume 3317, pages 146–155. Springer-Verlag.
- [Gotoh, 1982] Gotoh, O. (1982). An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162(3) :705–708.
- [Gupta, 2004] Gupta, S. (2004). Hardware acceleration of hidden markov models for bioinformatics applications. Master's thesis, Boise State University.
- [Gusfield, 1997] Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.
- [Guyetant, 2004] Guyetant, S. (2004). *Architecture reconfigurable pour le filtrage de banques de données non structurées ; application à la génomique*. Thèse de doctorat, Université de Rennes 1.
- [Haussler et al., 1993] Haussler, D., Krogh, A., Mian, I., et Sjolander, K. (1993). Protein modelling using hidden markov models : Analysis of globins. In *Hawaii Int. Conf. System Sciences*, pages 792–802.
- [Henikoff et Henikoff, 1992] Henikoff, J. et Henikoff, S. (1992). Amino Acid Substitution Matrices form Protein Blocks. *Proc. Natl. Acad. Sci. USA*, 89 :10915–10919.
- [Hoang, 1993] Hoang, D. (1993). Searching genetic databases on splash2. In *FCCM'93, IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Napa, California.
- [Hopcroft et Ullman, 1969] Hopcroft, J. et Ullman, J. (1969). *Formal Languages and Their Relation to Automata*. Addison-Wesley.
- [Hughey, 1993] Hughey, R. (1993). Massively parallel biosequence analysis. Technical Report UCSC-CRL-93-14, University of California, Santa Cruz.
- [Hulo et al., 2004] Hulo, N., Sigrist, C., Le Saux, V., Langendijk-Genevaux, P., Bordoli, L., Gattiker, A., De Castro, E., Bucher, P., et Bairoch, A. (2004). Recent improvements to the PROSITE database. *Nucl. Acids. Res.*, 32.
- [Jonassen et al., 1995] Jonassen, I., Collins, J. F., et Higgins, D. (1995). Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4(8) :1587–1595.
- [Knowles et Gardner-Stephen, 2004] Knowles, G. et Gardner-Stephen, P. (2004). A new hardware architecture for genomic and proteomic sequence alignment. In *Proceedings of the IEEE Computational Systems Bioinformatics Conference*, San Francisco, CA.
- [Knuth et al., 1977] Knuth, D. E., Morris, J. H., et Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6 :323–350.
- [Krishnamurthy et al., 2004] Krishnamurthy, P., Buhler, J., Chamberlain, R., Franklin, M., Gyang, K., et Lancaster, J. (2004). Biosequence similarity search on the mercury system. In *Proceedings of the 15th IEEE Int. Conference on Application-Specific Systems, Architectures and Processors (bASAP'04)*, pages 365–375.
- [Kucherov et al., 2004] Kucherov, G., Noé, L., et Roytberg, M. (2004). A unifying framework for seed sensitivity and its application to subset seeds. Rapport de recherche, INRIA.
- [Kuich et Salomaa, 1986] Kuich, W. et Salomaa, A. (1986). *Semirings, Automata, Languages*. Springer-Verlag.
- [Kung et Leiserson, 1980] Kung, H. T. et Leiserson, C. (1980). *Algorithms for VLSI processors arrays*. Addison-Wesley.
- [Lavenier et Giraud, 2005] Lavenier, D. et Giraud, M. (2005). *Reconfigurable Computing*, chapitre Bioinformatics Applications. Springer.
- [Lavenier et al., 2003] Lavenier, D., Guyetant, S., Derrien, S., et Rubini, S. (2003). A Reconfigurable Parallel Disk System for Filtering Genomic Banks. In *Inter-*

- national Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 03)*, pages 56–59.
- [Lefebvre et al., 1999] Lefebvre, A., Lecroq, T., Dauchel, H., et Alexandre, J. (1999). FORRepeats : detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3) :319–326.
- [Leung et al., 2001] Leung, S. W., Mellish, C., et Robertson, D. (2001). Basic gene grammars and DNA-ChartParser for language processing of Escherichia coli promoter DNA sequences. *Bioinformatics*, 17(3) :226–236.
- [Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions. *Soviet Physics Doklady*, 10(8) :707–710.
- [Lipton et Lopresti, 1985] Lipton, R. et Lopresti, D. (1985). *A systolic array for rapid string comparison*, pages 363–376. H. Fuchs, Ed. Rockville, MD : Computer Science Press.
- [Lopresti, 1987] Lopresti, D. (1987). P-nac : A systolic array for comparing nucleic acid sequences. *Computer*, 20(7) :81–88.
- [Ma et al., 2002] Ma, B., Tromp, J., et Li, M. (2002). PatternHunter : faster and more sensitive homology search. *Bioinformatics*, 18(3) :440–445.
- [Marsan, 2002] Marsan, L. (2002). *Inférence de motifs structurés : algorithmes et outils appliqués à la détection de sites de fixation dans des séquences génomiques*. Thèse de doctorat, Université de Marne-la-Vallée.
- [Masek et Paterson, 1980] Masek, W. J. et Paterson, M. S. (1980). A faster algorithm for computing string edit distances. *J. Comput. Systems Sci.*, 20 :18–31.
- [Mohri, 1997a] Mohri, M. (1997a). Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2) :269–311.
- [Mohri, 1997b] Mohri, M. (1997b). Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2) :269–311.
- [Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).
- [Mosanya et Sanchez, 1999] Mosanya, E. et Sanchez, E. (1999). A FPGA-based hardware implementation of generalized profile search using online arithmetic. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, Monterey, California.
- [Muriki et al., 2005] Muriki, K., Underwood, K., et Sass, R. (2005). RC-BLAST : Towards an open source hardware implementation. In *HICOMB*.
- [Navarro et Raffinot, 2002] Navarro, G. et Raffinot, M. (2002). *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press.
- [Needleman et Wunsch, 1970] Needleman, S. et Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48 :443–453.
- [PATTINPROT, ] PATTINPROT. PATTINPROT. <http://npsa-pbil.ibcp.fr>.
- [Pearson et Lipman, 1988] Pearson, W. et Lipman, D. (1988). Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85 :3244–3248.
- [Quinton et Robert, 1989] Quinton, P. et Robert, Y. (1989). *Algorithmes et architectures systoliques*. Masson.
- [Rivas et Eddy, 2000] Rivas et Eddy (2000). The language of RNA : A formal grammar that includes pseudoknots. *BIOINF : Bioinformatics*, 16.
- [Sakarovitch, 2003] Sakarovitch, J. (2003). *Éléments de théorie des automates*. Vuibert.
- [Searls, 1992] Searls, D. B. (1992). The linguistics of DNA. *American Scientist*, 80 :579–591.
- [Searls, 1993a] Searls, D. B. (1993a). The computational linguistics of biological sequences. In Hunter, L., editor, *Artificial Intelligence and Molecular Biology*, pages 47–120. AAAI Press.
- [Searls, 1993b] Searls, D. B. (1993b). String variable grammar : a logic grammar formalism for the biological language of DNA. *Journal of Logic Programming*, 12 :1–30.
- [Searls, 1997] Searls, D. B. (1997). Linguistic approaches to biological sequences. *Computer Applications in the Biosciences*, 13(4) :333–344.
- [Sidhu et Prasanna, 2001] Sidhu, R. et Prasanna, V. K. (2001). Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM 01)*.
- [Singh et al., 1993] Singh, R. et al. (1993). *Research on Integrated System*, chapitre A Scalable Systolic Multiprocessor System for Analysis of Biological Sequences, pages 168 – 182. G. Borriello and C. Ebeling.
- [Smith et Waterman, 1981] Smith, T. et Waterman, M. (1981). Identification of common molecular subsequences. *J. Mol. Biol*, 147 :195–197.
- [Stoesser et al., 2003] Stoesser, G., Baker, W., Van den Broek, A., Garcia-Pastor, M., Kanz, C., Kulikova, T., Leinonen, R., Lin, Q., Lombard, V., Lopez, R., Mancuso, R., Nardone, F., Stoehr, P., Tuli, M. A., Tzouvara, K., et Vaughan, R. (2003). The EMBL nucleotide sequence database : major new developments. *Nucleic Acids Research*, 31(1) :17–22.
- [Thompson et al., 1994] Thompson, J., Higgins, D., et Gibson, T. (1994). Clustalw : improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22 :4673–4680.
- [Williams et Zobel, 2002] Williams, H. et Zobel, J. (2002). Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1) :63–78.
- [Wu et Manber, 1992] Wu, S. et Manber, U. (1992). Fast Text Searching Allowing Errors. *Communications of the ACM*, 35(10) :83–91.

## A CIRCUITS RECONFIGURABLES FPGA

Les circuits reconfigurables FPGA (*matrice de portes programmables, Field Programmable Gate Array*) permettent d'exploiter un parallélisme fort à certaines applications comme les recherches de similarités, de motifs ou de modèles sur des grandes banques de données. Ces circuits sont un intermédiaire entre les microprocesseurs conventionnels (architectures de Von Neumann) dont le circuit est figé et les technologies ASIC (*Application Specific Integrated Circuit*) où on réalise un processeur par application.

Un circuit FPGA est un tableau de cellules, les *tables de scrutation (Look-Up Tables, LUTs)* avec une interconnexion flexible (figure 31). Chaque LUT est une mémoire à  $2^n$  bits qui, configurée, calcule n'importe quelle fonction  $\langle n \mapsto 1 \rangle$  ( $n$  entrées binaires, 1 sortie binaire). Cette LUT est souvent reliée à un registre à 1 bit, l'ensemble étant appelé une *cellule logique*. Les FPGAs en vente en 2005 peuvent avoir jusqu'à 200 000 LUTs.

Il est possible de câbler n'importe quelle fonction ou algorithme en quelques millisecondes sur un tel support reconfigurable dès qu'il y a assez de cellules logiques et de réseau d'interconnexion. Comparé à une architecture conventionnelle, il est possible de paralléliser beaucoup d'opérateurs et ainsi de gagner en efficacité. Comparé à un ASIC dont le circuit est définitivement fixé, un circuit FPGA est programmable.

La gamme des Spartan 3-E de Xilinx est un exemple de FPGA à bas coût en 2005. Ces FPGAs contiennent jusqu'à plus de 30 000 LUTs de 16 bits pouvant réaliser n'importe quelle fonction  $\langle 4 \mapsto 1 \rangle$ .

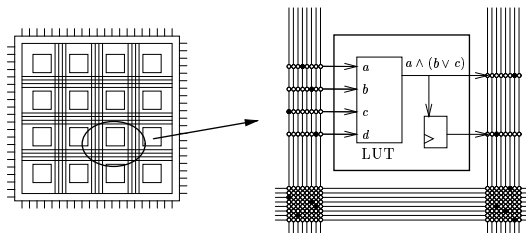


FIG. 31 – Structure simplifiée de la logique reconfigurable d'un FPGA. À gauche, vue d'ensemble. À droite, détail d'une table de scrutation à 16 bits qui calcule une fonction  $\langle 4 \mapsto 1 \rangle$  et dispose d'un bit de mémorisation.