Compression de séquences d'A.D.N. à base de grammaires minimales

Matthieu Perrin

3 septembre 2010

Rapport de stage de 1ere année du Magistère informatique et télécommunications, ENS Cachan/Bretagne et Université Rennes 1.
Stage effectué du 10 juin 2010 au 9 juillet 2010 à l'IRISA.

Encadrants

Responsable FranÃ§ois Coste, projet Symbiose, IRISA/INRIA Rennes-Bretagne Atlantique.

Doctorant Matthias Gallé, projet Symbiose, IRISA/INRIA Rennes-Bretagne Atlantique.

Résumé Les compresseurs de données généraux peinent généralement à compresser les séquences d'A.D.N. Par ailleurs, les codages à base de grammaires, bien que jugés très élégants, ne se sont pas montrés fructueux. Ce rapport présente des pistes pour le codage de grammaires algébriques sans boucle représentant de l'ADN ou d'autres types de données.

Mots clés ADN, compression, codage, grammaire, IRR.

1 Introduction

La taille des bases de données d'A.D.N., policière ou de recherche biologique, croît de faÃ§on exponentielle, ce qui implique de bonnes techniques de compression pour éviter de multiplier le matériel de stockage. Or, l'A.D.N. est justement réputé difficile à comprimer. Il s'agit d'une séquence sur un alphabet à quatre symboles, (A, C, G et T) et on peut donc imaginer un algorithme simple qui code chaque symbole sur deux bits. Cette borne de $2b/sym$ restera notre base.

1.1 Compresseurs

1.1.1 Compresseurs généraux

Les compresseurs classiques, dont gzip et bzip2, utilisent des algorithmes à dictionnaire du type de ceux de Lampel-Ziv (LZ77, LZ78) [7]. Ce dictionnaire maintient le décompte des occurences de sous-mots rencontrés. Lorsqu'une nouvelle occurence de ce sous-mot est rencontrée, et que cela est rentable, cette occurence est remplacée par une référence à la première occurence.

Ces compresseurs peinent généralement à atteindre la borne de $2b/sym$, et sont donc de mauvais compresseurs pour l'A.D.N. Pourtant, l'A.D.N. possède des caractéristiques propres héritées de son origine biologique, en particulier un taux élevé de répétitions, exactes ou approximatives, parfois issues de recopies du brin complémentaire. Les séquences considérées ne sont donc pas aléatoires et peuvent donc en théorie être compressées.

1.1.2 Compresseurs spécifiques

Ainsi, des compresseurs spécifiques à l'A.D.N. ont été développés, qui prennent en compte toutes ces spécificités. Ce sont généralement des compresseurs de type Lampel-Ziv, améliorés pour tenir compte des répétitions inexactes et des complémentaires [4][5][6].

Un algorithme, DNASequitur [2], utilise un tout autre principe. Il représente la séquence à comprimer sous forme de grammaire qui contient toutes les répétitions, puis il encode cette grammaire. Ce type d'encodage à base de grammaires peut en théorie permettre de bons résultats [1]. Cependant, les résultats obtenus furent plutôt décevant, mais de gros progrès dans le calcul des grammaires minimales nous incitent à continuer dans cette voie.

La question qui s'est donc posée pendant mon stage fut : Étant donnée une grammaire représentant une séquence d'A.D.N., comment représenter cette grammaire comme une séquence de bits la plus courte possible ? Nous verrons tout d'abord les techniques d'encodage d'une suite de caractères sous forme de bits, et en particulier l'encodeur arithmétique, puis ce qu'est plus précisément ce qu'est une grammaire, comment on peut s'en servir pour représenter des données et comment on peut l'obtenir. Enfin, nous présenterons les pistes explorées pour réduire la taille du code obtenu.

2 Techniques d'encodage

Un code est une suite finie de 0 et de 1 représentant des données. Généralement, ces données sont d'abbord encodées sous forme d'un mot $w =$

$w_1 \dots w_n$ sur un alphabet $\Sigma = \{\sigma_1 \dots \sigma_k\}$ dans une étape de linéarisation, puis w est réécrit en binaire, ce qui forme l'encodage proprement dit. C'est cette seconde étape que l'on se propose d'examiner ici.

Une première faÃ§on d'encoder le mot est de remplacer chaque symbole de Σ par un mot sur un nombre fixe de bits. Par exemple, pour encoder directement le brin d'A.D.N., on peut remplacer tous les A par $c(A) = 00$, les C par $c(C) = 01$, les G par $c(G) = 10$ et les T par $c(T) = 11$. La séquence $ACAGGAT$ devenant 00010010100011. Cette technique est très simple à décoder puisqu'il suffit de lire les bits deux par deux et d'appliquer la même transformation dans l'autre sens. Cependant, si l'on ajoute un symbole X à notre alphabet, on est obligé de rajouter un bit dans l'encodage de tous les symboles, soit : $c(A) = 000$, $c(C) = 001$, $c(G) = 010$, $c(T) = 001$ et $c(X) = 100$. Dans ce cas, certains codes n'ont pas d'antécédents, et l'information n'est donc pas suffisemment condensée. Pour palier ce problème, on peut tenter de coder les symboles par un nombre non-entier de chiffres en considérant que Σ est un ensemble de chiffres, et que le mot à coder est un nombre écrit en base k, le codage étant le même nombre écrit en base 2. Ainsi, chaque symbole est codé sur $log_2(k)$ bits. Si la base de départ est une puissance de 2, on retrouve exactement le codage précédent. Cependant, le changement de base peut être délicat à réaliser dans certains cas. Une autre solution est de donner une taille non-fixe pour le codage des symboles de Σ, comme dans le codage unaire où on aurait $c(A) = 1$, $c(C) = 01$, $c(G) = 001$, $c(T) = 0001$ et $c(X) = 00001$. On retrouve un codage de trois bits par symbole en moyenne, mais on peut choisir quel symbole aura quel code. En particulier, on peut faire en sorte que le symbole le plus fréquent ait le plus petit code.

Le coût d'encodage de w est

$$\sum_{i=1}^{n} |c(w_i)| = \sum_{j=1}^{k} f(\sigma_j)|c(\sigma_j)|$$

où $f(\sigma_j)$ est le nombre d'occurences de σ_j dans w. La théorie de l'information de Shannon nous apprend que les valeurs envisageables de $|c(\sigma_j)|$ qui aboutiront aux meilleurs résultats sont $|c(\sigma_j)| = -log_2(p(\sigma_j))$ pour tout j, avec $p(\sigma_j)$ la probabilité de rencontrer le symbole σ_j.

Des techniques, principalement l'arbre de Huffman, permettent d'affecter un code à chaque symbole étant donné un jeu de probabilités. L'encodeur arithmétique [8] (A.C. pour arithmetic coder en anglais) permet de combiner tous ces principes harmonieusement en permettant une grande précision dans le choix des probabilités. Son principe est non-plus de voir la séquence comme une écriture d'un nombre entier, mais comme celle d'un développement décimal, ou encore d'un nombre compris entre 0 et 1.

Le principe est d'encadrer ce nombre par une suite d'intervalles. Initialement, on se place sur $[0, 1[$, que l'on divise selon les probabilités données. Quand on encode un symbole, on se place sur la partition correspondante, et ainsi de suite. Les bits que l'on écrit permettent au décodeur de se placer en permanance sur le bon intervalle.

Prenons l'exemple de la séquence $AACGAAGACG$ sur notre alphabet de bases azotées. Les probabilités assignées aux symboles sont :

A	C	G	T
$\frac{1}{2}$	$\frac{1}{5}$	$\frac{3}{10}$	0

On commence le codage sur l'intervalle $[0, 1[$, partagé en $[0, \frac{1}{2}[\cup[\frac{1}{2}, \frac{7}{10}[\cup[\frac{7}{10}, 1[$. Le premier symbole rencontré est un A. L'intervalle courant est maintenant $[0, \frac{1}{2}[$. On écrit un 0 car tout l'intervalle est contenu dans $[0, \frac{1}{2}[$, et on étend l'intervalle courant à $[0, 1[$. On se retrouve exactement dans la même situation que précédemment, donc le second A aura les mêmes effets. Cette fois, le symbole rencontré est un C. On se place sur l'intervalle $[\frac{1}{2}, \frac{7}{10}[$. Comme il est contenu dans $[\frac{1}{2}, 1[$, on l'étend à $[0, \frac{2}{5}[$ en écrivant un 1, puis à $[0, \frac{4}{5}[$ en écrivant un 0. On partage cet intervalle en $[0, \frac{2}{5}[\cup[\frac{2}{5}, \frac{28}{50}[\cup[\frac{28}{50}, \frac{4}{5}[$. Le nouveau G va placer l'encodeur dans l'état $[\frac{28}{50}, \frac{4}{5}[$, et ainsi de suite. Les différents états de l'encodeur sont :

Symbole reÃ§u	Symbole envoyé	Intervalle
		$[0, 1[$
A		$[0, \frac{1}{2}[$
	0	$[0, 1[$
A		$[0, \frac{1}{2}[$
	0	$[0, 1[$
C		$[\frac{1}{2}, \frac{7}{10}[$
	1	$[0, \frac{2}{5}[$
	0	$[0, \frac{4}{5}[$
G		$[\frac{28}{50}, \frac{4}{5}[$
	1	$[\frac{3}{25}, \frac{3}{5}[$
A		$[\frac{3}{25}, \frac{9}{25}[$
	0	$[\frac{6}{25}, \frac{18}{25}[$
A		$[\frac{6}{25}, \frac{12}{25}[$
	0	$[\frac{12}{25}, \frac{24}{25}[$
G		$[\frac{102}{125}, \frac{24}{25}[$
	1	$[\frac{79}{125}, \frac{23}{25}[$
	1	$[\frac{33}{125}, \frac{21}{25}[$
A		$[\frac{33}{125}, \frac{69}{125}[$
C		$[\frac{51}{125}, \frac{291}{625}[$
	0	$[\frac{102}{125}, \frac{582}{625}[$
	1	$[\frac{79}{125}, \frac{539}{625}[$
	1	$[\frac{33}{125}, \frac{453}{625}[$
G		$[\frac{3666}{6250}, \frac{453}{625}[$
	1	$[\frac{541}{3125}, \frac{281}{625}[$
	0	$[\frac{1082}{3125}, \frac{562}{625}[$

Le code produit est alors 00101001101110, plus éventuellement un ou deux symboles pour supprimer les ambiguïtés, selon l'implémentation mais cette constante devient négligeable quand la taille de la séquence augmente, soit 1.4 bits par symbole, contre $log_2(3) \simeq 1.58$ bits par symboles avec une répartition équiprobable.

De plus, le codeur arithmétique permet de changer la distribution des probabilités au cours du temps, tant que l'encodeur et le décodeur utilisent les mêmes modèles. Par ailleurs, donner toutes les probabilités peut s'avérer très lourd. Dans le cas précédent, si l'on veut coder "$2, 3, 5,$", 7 bits sont nécessaires pour coder les nombres, et environ autant, selon la technique utilisée, pour coder les virgules, sans lesquelles le code produit n'est pas décodable. Une technique possible, utilisée par l'encodeur arithmétique adaptatif (adaptatif arithmetic

coder), est d'estimer la probabilité d'un symbole par son nombre d'occurences dans le préfixe déjà encodé.

Tous nos essais d'encodage, et la plupart des compresseurs, utilisent cet encodeur.

3 Grammaires algébriques sans boucle

Notre stratégie de compression utilise des grammaires formelles pour représenter les séquences d'A.D.N. Une grammaire est constituée de :

- L'ensemble fini Σ des symboles terminaux, dans notre cas l'ensemble $\{A, C, G, T\}$;
- L'ensemble fini Γ des symboles non-terminaux, désignés par des nombres ;
- Un élément particulier de l'ensemble des non-terminaux, appelé axiome ou séquence,
- Un ensemble de règles de production, qui sont des paires de suites de terminaux et de non-terminaux. Le premier élément de la règle est sa partie gauche (lhs) et le deuxième est sa partie droite (rhs).

Le langage reconnu par une telle grammaire est l'ensemble des mots sur l'alphabet des terminaux que l'on peut obtenir à partir de l'axiome en remplaÃ§ant successivement les parties gauches de règles par leur parties droites respectives.

Nous utilisons une classe restreinte de grammaires, les grammaires hors-contexte (ou algébriques du nom de la classe des langages qu'elles engendrent) sans boucle. Une grammaire hors-contexte est une grammaire formelle dans laquelle la partie gauche des règles est constituée uniquement d'un symbole non-terminal. Cette grammaire est sans boucle si pour chaque règle, le non-terminal de sa partie gauche n'apparaît pas dans la dérivation issue sa partie droite. Enfin, nous exigeons que chaque non-terminal apparaîsse une et une seule fois comme partie gauche d'une règle. Ainsi, on peut parler indifféremment de règle et de non-terminal.

Ces trois conditions imposent que le langage défini par une grammaire soit un singleton. L'encodage de la grammaire sera donc suffisant pour retrouver la séquence initiale.

Exemple :

$$\left\{ \begin{array}{lll} 0 & \rightarrow & 12 \\ 1 & \rightarrow & 2gt \\ 2 & \rightarrow & ac \end{array} \right.$$

Les terminaux de cette grammaire sont les quatre bases azotées et les non-terminaux sont les chiffres 0 (l'axiome), 1 et 2. On retrouve le mot constitutif du langage en partant de l'axiome :

$$0 \rightarrow 12 \rightarrow (2gt)(ac) \rightarrow ((ac)gt)(ac).$$

Le langage reconnu par cette grammaire est donc $\{acgtac\}$. Cependant, le même langage est reconnu par la grammaire :

$$\left\{ \begin{array}{lll} 0 & \rightarrow & 21 \\ 1 & \rightarrow & gt2 \\ 2 & \rightarrow & ac \end{array} \right. .$$

Pour avoir un codage efficace, on aimerait avoir la grammaire la plus petite possible. Dans l'exemple ci-dessus, la grammaire la plus concise pour représenter notre chaîne est :

$$\left\{ \begin{array}{lll} 0 & \rightarrow & 1gt1 \\ 1 & \rightarrow & ac \end{array} \right. .$$

Il existe plusieurs faÃ§ons de définir les grammaires minimales. Nous nous basons sur celle qui minimise la somme de la taille des parties droites, plus le nombre de règles. Trouver la grammaire minimale pour représenter une séquence est un problème NP-complet. Aussi, il existe plusieurs algorithmes d'approximation :

Sequitur [2] est un algorithme on-line, c'est à dire qu'il construit une grammaire au fur et à mesure qu'il parcourt la séquence, ce qui lui donne une grande rapidité. Cependant, les grammaires apprises avec Sequitur ne sont pas particulièrement petites, et ne respectent pas la structure de la séquence, ce qui empêche un fort taux de compression.

IRR [3] est un shéma général off-line qui utilise un algorithme glouton. A chaque étape, on choisit de s'arrêter ou bien de créer une nouvelle règle à partir d'une répétition pour minimiser une fonction de coût. Celle-ci peut être, entre autre :
- ml (maximal lenght) : choisit la répétition la plus longue.
- mr (maximal repeat) : choisit la répétition la plus fréquente.
- mc (maximal compression) : choisit la répétition qui minimise la taille de la grammaire.
- ts (total entropy) : choisit la répétition qui minimise l'entropie de Shannon empirique de la grammaire.

Bien que ts permette d'obtenir de meilleurs résultats en compression, nous nous basons sur mc, qui crée des grammaires plus petites mais avec plus de règles, et qui met mieux en évidence la structure de la séquence.

4 Encodage d'une grammaire

4.1 Linéarisation

Ni le nom des règles ni leur ordre n'a d'importance, et l'on peut renommer (tant que les symboles non-terminaux correspondants sont modifiés partout de la même faÃ§on) et réorganiser la grammaire sans risque. Par exemple, les deux grammaires

$$\left\{ \begin{array}{lll} 0 & \rightarrow & 12 \\ 1 & \rightarrow & 2gt \\ 2 & \rightarrow & ac \end{array} \right. \text{ et } \left\{ \begin{array}{lll} S & \rightarrow & YX \\ X & \rightarrow & ac \\ Y & \rightarrow & Xgt \end{array} \right.$$

reconnaissent exactement la même séquence et utilisent le même nombre de symboles. Cela nous donne deux pistes pour la compression.

Premièrement, le nom des règles peut être implicite : les non-terminaux sont des nombres entiers reliés à l'ordre des règles. Il n'est donc pas nécessaire de les encoder. Les deux grammaires précédentes seront donc représentées par $12|2gt|ac$ et $21|ac|1gt$.

Deuxièmement, l'ordre des règles peut être choisi de faÃ§on à favoriser diverses stratégies de prédiction du prochain symbole.

4.2 Algorithme M.T.F.

La principale difficulté rencontrée en compression à base de grammaires est l'augmentation du nombre de symboles. Les grammaires créées pour caractériser les séquences des corpus standards ont généralement autour d'un millier de règles. Il faut donc environ dix bits pour coder un symbole, et l'encodage n'est utile que si la somme des tailles des parties droites des règles est cinq fois plus petite que la taille de la séquence initiale.

Une idée est d'utiliser un alphabet plus petit, mais changeant au cours du temps. Cet alphabet local pourrait être modifié au début de l'encodage de chaque règle comme dans l'exemple ci-dessous, où les règles présentées sont entourées de mille autres, et l'alphabet local est initialement : $\{\alpha = a, \beta = c, \gamma = g, \delta = t\}$.

$$(\alpha\beta)\beta1(\beta\alpha)(\beta\beta)\gamma2(\beta\gamma)\delta c\gamma3(\beta\gamma\alpha\delta)\ldots$$

Dans cet exemple, les règles sont mises entre parenthèses. Voici les étapes effectuées par le décodeur :

Symboles lus	nature	règle décodée	nouvel alphabet
$(\alpha\beta)$	règle	$1 \rightarrow ac$	
$\beta1$	changement		$\{\alpha = a, \beta = 1, \gamma = g, \delta = t\}$
$(\beta\alpha)$	règle	$2 \rightarrow 1a$	
$(\beta\beta)$	règle	$3 \rightarrow 11$	
$\gamma2$	changement		$\{\alpha = a, \beta = 1, \gamma = 2, \delta = t\}$
$(\beta\gamma)$	règle	$4 \rightarrow 12$	
δc	changement		$\{\alpha = a, \beta = 1, \gamma = 2, \delta = c\}$
$\gamma3$	changement		$\{\alpha = a, \beta = 1, \gamma = 3, \delta = c\}$
$(\beta\gamma\alpha\delta)$	règle	$5 \rightarrow 13ac$	

La séquence à encoder est certes plus longue que $ac|1a|11|12|13ac$, mais en sachant que les lettres greques peuvent être codées sur 2 bits, et les autres sur 10, et sans compter les parenthèses, cet encodage prend 104 bits contre 120 bits en utilisant un seul alphabet. Bien sûre, le gain dépend de l'ordre des règles et du choix des changements de l'alphabet local.

Cepedant, dans ce système, l'entretient de l'alphabet local est plus cher que l'encodage des règles elles mêmes, et on se demande si on ne peut pas fournir de l'information implicitement.

L'algorithme M.T.F. (Move To Front) est utilisé poour encoder des séquences qui comportent des suites de lettres répétées (parfois la sortie de transformations, comme celle de Burrows-Wheeler), comme dans *aaaccgggaattt*. L'ordre des symboles est modifié pendant toute la linéarisation. À chaque fois qu'un symbole est encodé, il est placé en première position et les suivants sont reculés. La séquence obtenue est la suite des positions des symboles de la séquence originale. Par exemple dans la séquence précédante, on a :

Alphabet initial	Symbole lu	Symbole codé	Alphabet final
$\{a,c,g,t\}$	a	0	$\{a,c,g,t\}$
$\{a,c,g,t\}$	a	0	$\{a,c,g,t\}$
$\{a,c,g,t\}$	a	0	$\{a,c,g,t\}$
$\{a,c,g,t\}$	c	1	$\{c,a,g,t\}$
$\{c,a,g,t\}$	c	0	$\{c,a,g,t\}$
$\{c,a,g,t\}$	g	2	$\{g,c,a,t\}$
$\{g,c,a,t\}$	g	0	$\{g,c,a,t\}$
$\{g,c,a,t\}$	g	0	$\{g,c,a,t\}$
$\{g,c,a,t\}$	a	2	$\{a,g,c,t\}$
$\{a,g,c,t\}$	a	0	$\{a,g,c,t\}$
$\{a,g,c,t\}$	t	3	$\{t,a,g,c\}$
$\{t,a,g,c\}$	t	0	$\{t,a,g,c\}$
$\{t,a,g,c\}$	t	0	$\{t,a,g,c\}$

On remarquera le nombre de 0 produits qui pourra être mis à profit par l'encodeur arithmétique qui favorise les plus petits nombres, ou tout autre codeur entropique.

Dans les codages à grammaires, on peut optimiser le M.T.F. en choisissant l'ordre des règles. On peut même imaginer une méthode de Backtracking pour améliorer la proximité spatiale des règles syntaxiquement proches (et pour obtenir un algorithme de complexité polynomiale) bien que la méthode impose d'encoder une structure d'arbre.

Cet algorithme est toujours meilleur que le précédent, et permet d'obtenir des résultats encourageants par rapport à l'encodeur arithmétique. Cependant, l'adaptativité de ce dernier, très important pour arriver à l'état de l'art, n'améliore pas assez l'algorithme M.T.F. pour le rendre intéressant. Cela est dû à l'inégalité du nombre d'occurences des règles. Certaines règles sont très souvent utilisées et les modèles d'encodages tirent fortement parti de leur fréquence. Certes, dans l'algorithme M.T.F., ces règles occupent souvent les premières positions, mais cette information est trop floue pour pouvoir l'utiliser efficacement.

4.3 Meilleure utilisation de l'A.A.C.

Bien que l'ordre d'encodage des règles reste un élément prometteur, l'essai expliqué précédemment montre la puissance de l'encodeur arithmétique s'il est utilisé avec un bon modèle de probabilités, même si la séquence à encoder peut être modifiée pour adapter les probabilités, soit pour pouvoir mieux les prédire, soit pour favoriser certains symboles (l'entropie est maximale lorsque toutes les probabilités sont égales). Pour cela, trois stratégie ont été imaginées :

Par niveaux : On peut chercher une structure à l'ensemble des règles d'une grammaire. En particulier, on peut les séparer par hauteur dans l'arbre de dérivation du mot unique du langage.

$$\begin{aligned} E_0 &= \Sigma \\ E_{i+1} &= \{\gamma \in \Gamma, rhs(\gamma) \in \textstyle\bigcup_{k=0}^{i} E_k\} \\ E'_0 &= E_0 \\ E'_{i+1} &= E_{i+1} - E_i \end{aligned}$$

Les règles contenues dans chaque niveau peuvent être codées en utilisant seulement des règles déjà codées dans les niveaux inférieurs. Cela réduit le

nombre de symboles possibles, et donc le nombre de bits nécessaires pour encoder un symbole.

Anti-dictionnaire : Si la plupart des compresseurs cherchent à utiliser au mieux les redondances d'un texte, ils ne peuvent s'appliquer à une grammaire dans laquelle les répétitions ont justement été éliminées. On peut par contre utiliser cette propriété qui interdit toute partie droite de règle d'être un sous mot de la partie droite d'une autre règle, sans quoi on peut la remplacer par le non-terminal correspondant, ou encore :

$$\forall \gamma = \gamma_0 \dots \gamma_{|\gamma|} \in \Gamma, \forall \delta = \delta_0 \dots \delta_{|\delta|} \in \Gamma, \forall 0 \leq i \leq |\delta| - |\gamma|,$$

$$(\delta_i = \gamma_0, \dots, \delta_{i+|\gamma|-1} = \gamma_{|\gamma|-1}) \Rightarrow P(\delta_{i+|\gamma|} = \gamma_{|\gamma|}) = 0$$

Ancienneté : On peut essayer de conjuguer les effets du M.T.F. avec ceux de l'adaptativité de l'encodeur arithmétique. Pour cela, on crée un nouveau modèle de probabilités qui prend en compte à la fois le nombre d'occurences de chaque symboles rencontrés et le nombre de symboles rencontrés depuis la dernière occurence du symbole. On peut choisir d'accentuer plutôt l'un ou l'autre aspect.

5 Structure de treillis

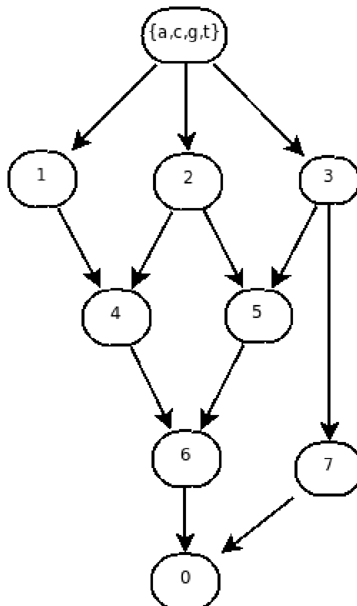
Pour pousser la recherche plus loin, on peut chercher à s'attaquer de face au problème du nombre de symboles, en prévoyant précisément les symboles qui doivent apparaître dans la partie droite de chaque règle. Pour étudier la structure de la grammaire, introduisons la relation $\prec$ sur l'ensemble des règles définie par : pour tout $\gamma_1, \gamma_2 \in \Gamma, \gamma_1 \prec \gamma_2$ si $\gamma_2 \in rhs(\gamma_1)$ ou $\gamma_1 = \gamma_2$, que l'on complète par transitivité. Cette règle est bien sûre réflexive et transitive. Elle est également antisymétrique car la grammaire est sans boucle. On a donc une relation d'ordre partiel. De plus, la séquence est l'unique élément maximal de cette règle. Si l'on ajoute un élément minimal, qui pourrait correspondre à l'ensemble des terminaux, on obtient une structure de treillis. Trouver une faÃ§on efficace d'encoder un treillis donnerait une faÃ§on efficace d'encoder une grammaire.

Prenons l'exemple sur la grammaire suivant :

$$\left\{ \begin{array}{lcl} 0 & \rightarrow & 616a524g7c337 \\ 1 & \rightarrow & ac \\ 2 & \rightarrow & gt \\ 3 & \rightarrow & ag \\ 4 & \rightarrow & 12 \\ 5 & \rightarrow & 23 \\ 6 & \rightarrow & 245 \\ 7 & \rightarrow & 33 \end{array} \right.$$

La figure 1 montre le graphe correspondant à cette grammaire. On retrouve l'idée de l'encodage par niveaux, mais on s'apperÃ§oit que cette remarque est plus générale. En particulier, il est inutile ici de prendre en compte les symboles 1 à 6 dans l'encodage de la règle 7.

Fig. 1 – Graphe dont la clôture transitive représente un treillis.



6 Conclusion

6.1 Extension

Tout le travail présenté a été effectué sur des séquences d'A.D.N., c'est à dire avec l'alphabet à quatre lettres A,C,G,T. Toutefois, nous n'utilisons pas de propriétés spécifiques à cet alphabet de terminaux, et l'encodage pourrait être fait sur tout autre alphabet. Dans ce cas, il faut encoder également l'alphabet. On se limite ici aux symboles ASCII.

Pour encoder cet alphabet, nous proposons de trier les symboles dans l'ordre alphabétique, et d'utiliser un alphabet de plus en plus petit. On pourrait faire mieux en autorisant des plages de symboles $(a - z)$ par exemple, ou même peut-être en encodant le premier, puis le dernier, puis le deuxième et ainsi de suite, car la répartition des symboles utilisés n'est généralement pas homogène, ni centrée sur la fin de l'alphabet.

6.2 Résultats

Les algorithmes de compression sont comparés sur des corpus standards. Le corpus généralement utilisé pour les compresseurs d'A.D.N. est le corpus historique de Manzini. les résultats sont donnés en bits par symbole, en se référant aux données fournies par les auteurs dans leurs articles.

Sequence	BioC	GenC	DNAC	DNAP
CHMPXX	1.6848	1.6730	1.6716	1.6602
CHNTXX	1.6172	1.6146	1.6127	1.6103
HEHCMVCG	1.8480	1.8470	1.8492	1.8346
HUMDYSTROP	1.9262	1.9231	1.9116	1.9088
HUMGHCSA	1.3074	1.0969	1.0272	1.039
HUMHBB	1.8800	1.8204	1.7897	1.7771
HUMHDAB	1.8770	1.8192	1.7951	1.7394
HUMHPRTB	1.9066	1.8466	1.8165	1.7886
MPOMTCG	1.9378	1.9058	1.8920	1.8932
MTPACG	1.8752	1.8624	1.8556	1.8535
VACCG	1.7614	1.7614	1.7580	1.7583
Moyenne	1.7837	1.7428	1.7254	1.7148

Sequence	CDNA	GeMNL	XM	Sequitur
CHMPXX	-	1.6617	1.6577	-
CHNTXX	1.65	1.6101	1.6068	2.12
HEHCMVCG	-	1.8420	1.8426	2.12
HUMDYSTROP	1.93	1.9085	1.9031	2.34
HUMGHCSA	0.95	1.0089	0.9828	1.86
HUMHBB	1.77	-	1.7513	-
HUMHDAB	1.67	1.7059	1.6671	-
HUMHPRTB	1.72	1.7639	1.7361	-
MPOMTCG	1.87	1.8822	8768	-
MTPACG	1.85	1.8440	1.8447	2.16
VACCG	1.81	1.7644	1.7649	2.11
Moyenne	-	-	1.6940	-

Voici les résultats obtenus avec les algorithmes développés pendant mon stage. La colonne M.T.F. montre les meilleurs résultats de l'algorithme Move To Front. La colonne A.A.C montre les résultats en encodant directement la séquence obtenue par la linéarisation avec un encodeur arithmétique adaptatif. Enfin, la dernière colonne comporte les trois améliorations à cet algorithme décrites plus haut.

Certains résultats manquent à cause d'un bogue découvert à la fin du stage dans l'encodeur arithmétique.

Dans ce premier tableau, les grammaires utilisées sont calculées avec la fonction de coût mc (maximal compression).

Sequence	M.T.F	A.A.C	A.A.C. amélioré
CHMPXX	2.0371	2.0931	-
CHNTXX	-	2.2179	2.1726
HEHCMVCG	-	2.2119	-
HUMDYSTROP	2.2054	2.2884	2.1900
HUMGHCSA	1.6374	1.6732	1.6146
HUMHBB	2.1192	2.1760	2.1222
HUMHDAB	2.1219	2.1948	2.1329
HUMHPRTB	2.1216	2.1990	2.13326
MPOMTCG	-	2.2012	-
MTPACG	2.0789	2.1457	2.0895
VACCG	-	2.1008	2.0609

Dans ce deuxième tableau, les grammaires utilisées sont calculées avec la fonction de coût ts (total entropy).

Sequence	M.T.F	A.A.C	A.A.C. amélioré
CHMPXX	1.8967	1.9004	1.8842
CHNTXX	2.0012	2.0048	1.9808
HEHCMVCG	2.0264	2.0295	2.0053
HUMDYSTROP	1.9720	1.9749	1.9558
HUMGHCSA	1.5756	1.4876	1.4417
HUMHBB	1.9687	1.9532	1.9258
HUMHDAB	1.9507	1.9513	1.9292
HUMHPRTB	1.9563	1.9585	1.9354
MPOMTCG	2.0155	2.0086	-
MTPACG	1.9263	1.9274	1.9055
VACCG	1.9306	1.9323	1.9074

Ce nouveau compresseur n'atteint pas les performances des compresseurs spécifiques à A.D.N. Cependant, les résultats sont meilleurs que ceux obtenus avec Sequitur, et on a donc amélioré la technique de compression à base de grammaires.

De plus, on remarque que les meilleures améliorations par rapport à l'encodeur arithmétique adaptatif sont obtenues avec humghcsa, qui est la séquence la plus compressible. Elle comporte en effet le plus grand nombre de répétitions. Ainsi, la grammaire la représentant est la plus fournie. L'effet est bien produit sur la grammaire, et non pas sur l'axiome, très long et très peu compressible.

6.3 Ouverture

Les différentes pistes de compression proposées ici ne sont pas optimales, c'est à dire qu'un autre code plus court peut être décodé avec le même compresseur pour donner la même séquence.

L'encodage par niveaux peut être assoupli en ajoutant des niveaux intermédiaires. On peut travailler plus en profondeur sur l'optimisation de l'ancienneté. Si la NP-complétude risque d'être facilement démontrable, on peut en revanche trouver de bien meilleures approximations. Quant à l'anti-dictionnaire, des méthodes statistiques plus précises peuvent permettre un léger gain de compression. Cependant, des gains significatifs dans l'encodage ne pourront être obtenus que par de nouvelles idées.

Dautres améliorations peuvent par contre être obtenues sur l'apprentissage de la grammaire en changeant la fonction de coût utilisée. Pour l'instant les meilleurs résultats sont obtenus en minimisant l'entropie, ce qui est moins intéressant au point de vue biologique que la compression maximale.

Le succès de l'encodage par niveaux serait renforcé si les grammaires étaient plus hautes. Les grammaires utilisées ont généralement autour de cinq niveaux. Pour cela, on pourrait introduire des grammaires à trous, avec des règles de la forme

$$1 \rightarrow ga * c$$

où le contenu de l'étoile pourrait n'être défini qu'au moment de l'utilisation du non-terminal. Prenons par exemple la séquence *gacgtcgcc*. Elle ne contient

qu'une seule répétition, et la grammaire minimale correspondante est donc

$$\left\{ \begin{array}{lcl} 0 & \rightarrow & ga1t1cc \\ 1 & \rightarrow & cg \end{array} \right.$$

Avec une grammaire à trous, on obtient la grammaire

$$\left\{ \begin{array}{lcl} 0 & \rightarrow & 1(a)1(t)1(c) \\ 1 & \rightarrow & g * c \end{array} \right.$$

Les grammaires à trous ont de plus une justification biologique. Les trous sont des mutations dans des duplications de séquences d'A.D.N.

Une autre amélioration uniquement réservé à l'A.D.N. est l'utilisation des complémentaires renversés. Les répétitions dans l'A.D.N. sont dues à des erreurs biologiques lors de la lecture d'une séquence. Cependant, l'A.D.N. est composé de deux brins complémentaires, c'est à dire qu'une adénine (a) est placée en face d'une thynine (t), une cythosine (c) en face d'une guanine (g) et inversement. De plus, le sens de lecture des deux brins est différent. Une dupplication d'un brin sur un autre crée donc une répétition en complément inversé. Dans l'exemple suivant, les quatre derniers symboles sont les compléments inversés des quatre premiers.

$$\begin{array}{cccccccccccccccccc} \rightarrow & a & g & t & c & a & g & a & t & c & g & a & g & a & c & t & \\ & | & | & | & | & | & | & | & | & | & | & | & | & | & | & | & \\ & t & c & a & g & t & c & t & a & g & c & t & c & t & g & a & \leftarrow \end{array}$$

L'utilisation des compléments inversés a tout de même un défaut, celui de doubler le nombre de règles de la grammaire.

Les améliorations proposées ici demandent des modifications dans le calcul des grammaires, mais aussi dans l'encodage qui doit en tirer le meilleur parti.

6.4 Synthèse

Toute tentative d'amélioration qui ajoute trop d'information est vouée à l'échec. Par exemple, on pourrait être tenté de donner le nombre d'occurences de chaque symbole pour pouvoir les supprimer de l'alphabet lorsqu'ils ne sont plus utiles, mais l'en-tête à ajouter est tout simplement trop longue. On pourrait aussi ajouter un symbole pour supprimer le dernier symbole utilisé, mais bien que l'information ajoutée soit beaucoup plus condensée, elle est toujours trop importante pour être rentabilisable. De même en est-il pour l'alphabet local du premier algorithme.

Quant au M.T.F., son défaut vient du fait qu'il s'effectue au détriment des propriétés naturelles de la grammaire.

Finalement, la grande règle à tirer de ce stage est qu'en compression, peut-être plus encore qu'ailleurs, ce qui est le plus simple fonctionne le mieux.

Bibliographie

1 *Grammar-Based Codes : A New Class of Universal Lossless Source Codes* [J. C. Kieffer, E.-H. Yang]

2 *Grammar-based Compression of DNA Sequences*[N. Cherniavsky, R. Ladner]

3 *Searching for Smallest Grammars on DNA Sequences*[R. Carrascosa, F. Coste, M. Gallé, G. Infante-Lopez]
4 *DNA Compression Challenge Revisited : A Dynamic Programming Approach* [B. Behzadi, F. le Fessant]
5 *Compression of DNA Sequences* [S. Grumbach, F. Tahi]
6 *A new Challenge for compression algorithms : genetic sequences* [S. Grumbach, F. Tahi]
7 *A Universal Algorithm for Sequential Data Compression, in IEEE Transactions on Information Theory* [J. Ziv, A. Lempel]
8 *Arithmetic coding* [J.J. Rissanen, G.G. Langdon]

7 Annexe pour la suite du stage

7.1 Encodage du treillis

Je ne sais pas s'il existe une faÃ§on d'encoder notre treilli sans utiliser de pointeur. Sinon, on doit pouvoir le montrer en utilisant des considérations entropiques.

On peut approcher la structure en ajoutant arbitrairement des éléments de hiérarchie entre certains éléments non-comparables. La meilleure approximation est celle qui modifie le moins l'ordre. Voici trois piste d'approximation :

Premièrement, la complétion en un ordre total (C'est ce que propose Raphael).

Deuxièmement, le découpage par niveaux. Dans l'exemple de la figure 1, cela donne $\{a, c, g, t\}, \{1, 2, 3\}, \{4, 5, 7\}, \{6\}, \{0\}$. On retrouve précisément les niveaux en dessinant les noeuds aussi haut que possible (ici, il faut remonter le 7).

La troisième faÃ§on est d'utiliser ce que j'ai appelé rivière (faute de mieux car les flèches me font penser à l'écoulement de l'eau dans une rivière pleine d'îlots). Une relation d'ordre $\prec$ sur E est une rivière si $\prec$ est un ordre total ou si il existe une partition H, B, D, G de E telle que H, B, D et G sont des rivières, et pour tout $h \in H$, pour tout $b \in B$, pour tout $d \in D$ et pour tout $g \in G$,

- $d \prec h$,
- $g \prec h$,
- $b \prec d$,
- $b \prec g$,
- d et g ne sont pas comparables.

Une telle rivière peut être encodée :

$$C(E) = C(H)\ \{\ C(G)\ ,\ C(D)\ \}\ C(B)$$

Exemple :

La rivière sur la figure 2 peut être encodée : $1/2/3(4/5(6/8, 7)9, 10)11$.

Il existe une rivière pour approcher n'importe quel treillis puisque les deux exemples ci-dessus en sont. Voici leurs codes pour ceux correspondant à la figure 1 :

L'ordre total : par exemple : $1/2/3/4/5/6/7/0$

L'encodage par niveaux : $(1, (2, 3))/(4, (5, 7))/6/0$

Le problème qui se pose est de trouver la rivière la plus proche du treillis, pour minimiser le nombre de noeuds inférieurs à chaque autre.

Fig. 2 – Exemple de rivière