Laboratoire d'Informatique
Fondamentale de Lille

THÈSE
Nouveau Régime

Présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

Par

OLIVIER DELGRANGE

UN ALGORITHME RAPIDE POUR UNE COMPRESSION MODULAIRE OPTIMALE APPLICATION À L'ANALYSE DE SÉQUENCES GÉNÉTIQUES

Soutenance le 25 juin 1997 devant le jury composé de :

Rapporteurs :	Maxime CROCHEMORE	Université de Marne-la-Vallée
	Jean-Marc STEYAERT	École Polytechnique de Palaiseau
Examinateurs :	Véronique BRUYÈRE	Université de Mons-Hainaut (Belgique)
	Max DAUCHET	Université de Lille I
	Jean-Paul DELAHAYE	Université de Lille I
	Pierre DUFOUR	Université de Mons-Hainaut (Belgique)
	Alain HÉNAUT	Université de Versailles Saint-Quentin
	Sophie TISON	Université de Lille I

Thèse défendue le 5 juin 1997 pour l'obtention du grade académique de
DOCTEUR EN SCIENCES de l'Université de Mons-Hainaut

Composition du jury :

Président :	Pr.	Maurice BOFFA	Université de Mons-Hainaut
Membres :	Dr.	Véronique BRUYÈRE	Université de Mons-Hainaut
	Pr.	Max DAUCHET	Université de Lille I
	Pr.	Jean-Paul DELAHAYE	Université de Lille I
	Pr.	Pierre DUFOUR	Université de Mons-Hainaut
	Pr.	Gaëtan LIBERT	Faculté Polytechnique de Mons
	Pr.	Guy NOËL	Université de Mons-Hainaut
	Mr.	Pierre ROUZÉ	Université de Gand

Thèse soutenue le 25 juin 1997 pour l'obtention du titre de
DOCTEUR EN INFORMATIQUE de l'Université des Sciences et Technologies de Lille

Composition du jury :

Président :	Pr.	Sophie TISON	Université de Lille I
Rapporteurs :	Pr.	Maxime CROCHEMORE	Université de Marne-la-Vallée
	Dr.	Jean-Marc STEYAERT	École Polytechnique de Palaiseau
Examinateurs :	Dr.	Véronique BRUYÈRE	Université de Mons-Hainaut
	Pr.	Max DAUCHET	Université de Lille I
	Pr.	Jean-Paul DELAHAYE	Université de Lille I
	Pr.	Pierre DUFOUR	Université de Mons-Hainaut
	Pr.	Alain HÉNAUT	Université de Versailles Saint-Quentin

À Aurore et Adrien

Remerciements

Je remercie Sophie Tison et Maurice Boffa qui ont présidé les jurys français et belge de la thèse.

Je remercie chaleureusement Max Dauchet, qui a dirigé ce travail, pour le temps qu'il a bien voulu me consacrer, pour ses nombreuses idées et pour ses encouragements continuels.

Je remercie Maxime Crochemore pour l'intérêt qu'il porte à mes travaux et pour avoir accepté d'être rapporteur.

Je remercie Jean-Marc Steyaert qui a accepté la lourde tâche d'être rapporteur.

Je remercie Alain Hénaut, Gaëtan Libert et Guy Noël pour l'attention qu'ils ont accordé au texte de la thèse et à ses défenses.

Je remercie Jean-Paul Delahaye pour son accueil chaleureux au sein de l'équipe du LIFL, et pour les collaborations et les réunions de travail enrichissantes.

Je remercie Eric Rivals pour sa sympathie, les collaborations nombreuses et fructueuses ainsi que pour son soutien administratif dans ce "grand pays où le petit belge est parfois un peu perdu".

Je remercie Pierre Dufour pour la confiance et l'autonomie qu'il m'a accordées durant ces années ainsi que pour le temps consacré à lire la thèse.

Je remercie Véronique Bruyère de m'avoir présenté à l'équipe du LIFL et d'avoir lu ce travail avec beaucoup d'attention.

Je remercie Pierre Rouzé pour les remarques judicieuses faites lors de la défense belge de la thèse.

Je remercie la communauté "Informatique et Génome" française de m'avoir accepté avec autant de naturel.

Je remercie Lyane Bouchez et Anne-Marie Struyf qui ont relu avec attention la version préliminaire de ce travail. Un merci particulier à Lyane pour ses conseils LaTeX.

Je remercie Jacques Lion pour l'ambiance des indispensables pauses café et pour le graphique de la page 215.

Je remercie tous les collègues qui m'ont soutenu durant ces quelques années de thèse, en particulier merci à Laurent Devos qui a longtemps été embarqué dans la "même galère".

Je remercie Marie-France Sagot qui m'a gentiment permis d'utiliser quelques une de ses illustrations pour la partie biologique.

Je remercie mon père et Jean-Claude Cordier pour la reprographie de dernière minute de la version belge de la thèse.

Enfin, je remercie les éternels oubliés : tous ces programmeurs du domaine public, pour les outils de grande qualité qu'ils ont conçus et sans lesquels cette thèse ne serait pas ce qu'elle est ("messieurs" LaTeX, Gnuplot, Xfig, Emacs,...)

Résumé

Une méthode de compression sans perte d'informations applique souvent le même schéma de codage d'un bout à l'autre de la séquence à comprimer. Certains facteurs de la séquence sont ainsi raccourcis mais malheureusement d'autres sont rallongés.

Dans ce travail, nous proposons un algorithme d'optimisation de compression qui rompt le codage là ou il n'est pas intéressant en recopiant des morceaux de la séquence initiale. La compression obtenue est dite *modulaire* : la séquence comprimée est une succession de morceaux comprimés et de morceaux recopiés tels quels. Sous certaines hypothèses, notre algorithme fournit une *compression modulaire optimale* en temps $O(n \log n)$ où n est la longueur de la séquence. Nous montrons que notre méthode de compression peut avantageusement être utilisée pour analyser des données et plus particulièrement des séquences génétiques. La théorie de la complexité de Kolmogorov éclaire l'idée d'analyse de séquences par compression.

Le travail comporte trois parties. La première introduit les concepts classiques de compression et de codage, ainsi que le concept nouveau de codage ICL d'entiers. La seconde développe l'algorithme d'*optimisation de compression par liftings* qui utilise les codes ICL. La dernière partie présente des applications de l'optimisation de compression par liftings, plus particulièrement dans le domaine de l'analyse de séquences génétiques. Nous montrons, à l'aide du problème spécifique de localisation de répétitions en tandem approximatives, comment l'algorithme d'optimisation par liftings peut être utilisé pour localiser précisément et de manière optimale les segments réguliers et les segments non réguliers des séquences. Il s'agit d'un retour à l'expérience qui permet l'analyse de séquences de plusieurs centaines de milliers de bases en quelques secondes.

Table des matières

Introduction

L'utilisation sans cesse croissante de l'informatique dans la société actuelle produit des quantités de données de plus en plus grandes. Ce sont par exemple des textes, des bases de données, des images, des sons, des animations, etc. Elles sont traitées, stockées et éventuellement transmises à distance. La compression informatique joue un rôle dans chacune de ces trois tâches. Son but premier est de réduire la taille des données. Dans le cadre du stockage et de la transmission de l'information, elle constitue une économie substantielle dans la mesure où le budget consacré à l'achat de disquettes, de disques durs ou au paiement des notes téléphoniques peut être consacré à d'autres choses. Il est courant, dans son utilisation de tous les jours, qu'elle réduise de moitié la taille occupée par les données. Par abus de langage, il est d'ailleurs commun d'entendre dire que la compression "double" la taille des disques durs.

L'utilisation que nous faisons ici de la compression dans le contexte du traitement des données est fondamentalement différente : nous l'utilisons pour analyser l'information. En effet, la compression exige une certaine compréhension des données à comprimer. Mieux les données sont comprises, mieux elles sont comprimées. La compression d'une séquence de symboles procède par détection de régularités dans la séquence et codage efficace des régularités détectées. La présence des régularités se traduit par une certaine redondance qui est éliminée ou du moins atténuée dans la version comprimée de la séquence. La réduction de longueur que la compression produit est une mesure de la quantité de régularités détectées. Par exemple, si une séquence est constituée de trois copies d'un même fragment, la compression évite le codage de deux des copies. Chacune d'elles est remplacée par un code qui spécifie que le fragment est une copie du morceau laissé intact. La longueur de la séquence est divisée approximativement par trois, ce qui exprime clairement que seulement un tiers de la séquence constitue la totalité de son contenu en information. Lorsque, comme c'est le cas dans cet exemple, la séquence comprimée permet de reconstruire intégralement la séquence initiale, la compression est dite conservative.

L'analyse de séquences génétiques est la motivation de notre travail. Les techniques automatisées de séquençage de génomes font qu'on dispose subitement de quantités énormes de séquences génétiques, donc "d'informations de base sur la vie", qui ouvrent un nouveau champ d'exploration. Motivés par un problème précis d'analyse de séquences génétiques, nous avons conçu un algorithme d'optimisation de méthodes de compression conservatives, algorithme dit *d'optimisation par liftings*. Il a la triple propriété de produire une compression optimale (en un sens que l'on précise), efficace et d'un intérêt général. Il prend, en entrée, le résultat de la compression d'une séquence et produit, en sortie, une optimisation de la compression. La présentation que nous en faisons ne retrace pas la démarche historique de son développement. Nous avons pu isoler une méthode générale qui constitue le cœur algorithmique du travail. Il est présenté dans la partie II.

Le deuxième point consistant du travail est le retour à l'expérimentation proposé dans la

partie III. Le problème général est, pour une propriété "locale" donnée, de déterminer sur de très grandes séquences génétiques, où cette propriété est vérifiée "par hasard" et où elle est vérifiée de façon significative. Nous entendons par propriété locale une propriété qui garde son sens indépendamment du contexte : par exemple, "avoir" une longueur paire n'est pas local, contenir six `A` consécutivement au lieu de deux est local. Par ailleurs, la théorie de la complexité de Kolmogorov permet d'identifier "par hasard" à "ne pouvant pas donner lieu à compression". Notre méthodologie consiste à une analyse par compression en deux phases :

Phase 1 : Compression aveugle par exploitation d'un type précis de régularité (c'est-à-dire de propriété).

Phase 2 : Optimisation globale de la compression par liftings.

L'optimisation globale par liftings fait le tri de façon optimale entre ce qui est significatif et ce qui ne l'est pas. Significatif ne prend pas ici de sens biologique, mais un sens général en terme de théorie algorithmique de l'information, ce qui revient à considérer les phénomènes sans connaissance, a priori, comme le montre la théorie.

Plus particulièrement, l'algorithme d'optimisation par liftings est appliqué ici à la localisation d'un type spécifique de régularité dans les séquences génétiques : les répétitions en tandem (c'est-à-dire côte à côte) approximatives. Ce problème n'est pas nouveau, il a été abordé à plusieurs reprises [BW94, FLSS92, Riv96, RDDD95, RDD+97]. Notre méthode peut être perçue comme une extension de la méthode d'alignement par programmation dynamique cyclique (Wraparound Dynamic Programming - WDP) de Fischetti, Landau, Schmidt et Sellers [FLSS92]. L'utilisation de la WDP représente la phase 1 de l'analyse.

Pour favoriser la compréhension intuitive de l'utilité de l'optimisation par liftings, nous allons l'introduire en nous appuyant sur nos motivations historiques. Il ne faut cependant jamais perdre de vue que l'algorithme développé est largement indépendant de la problématique que nous résolvons. Pour cette raison, la partie II lui est entièrement consacrée.

Avant de commencer le développement, il est important de situer clairement notre contribution personnelle dans l'ensemble des recherches évoquées au long de ce travail. La partie II, c'est-à-dire le développement de l'algorithme d'optimisation par liftings est essentiellement personnelle. La majorité des études présentées dans la partie III (la partie application) sont le fruit de la communauté "Informatique et Génome", et en particulier du groupement de recherche CNRS du même nom. Notre contribution personnelle dans cette partie se limite au chapitre 1, qui décrit d'une manière générale l'utilisation concrète de l'optimisation par liftings, et à la section 4.5 du chapitre 4 qui décrit l'utilisation que nous en faisons pour résoudre le problème précis qui nous intéresse. Dans cette section, nous présentons également la méthode de programmation dynamique cyclique sous la forme d'automates finis, plus précisément, sous la forme de transducteurs. Cette formalisation n'est pas nouvelle, elle s'inspire de [SM95, Lef96], mais dans le contexte, nous jugeons qu'elle constitue un travail de clarification important.

Le reste du chapitre 4 relève d'un travail coopératif réalisé au sein de l'équipe CAABAAL du Laboratoire d'Informatique Fondamentale de Lille (LIFL). Ce sont des études menées essentiellement par Eric Rivals dans le cadre de sa thèse de doctorat [Riv96]. Les articles joints dans les annexes B et C illustrent notre collaboration dans ces études. L'esquisse de problème que nous dressons dans le chapitre 5 résulte également d'une collaboration au sein de l'équipe CAABAAL, collaboration menée essentiellement avec Eric Rivals, dans laquelle nous avons joué un rôle prépondérant. En ce qui concerne la partie I, il s'agit de préliminaires portant

sur l'algorithmique des mots et la théorie de l'information. Nous y définissons cependant la nouvelle notion de code ICL de représentation d'entiers (chapitre 2). Cette notion est utilisée de manière intensive par notre algorithme d'optimisation. Le code ICL *PrefFibo* est également un développement personnel.

De manière simpliste, les séquences génétiques (plus précisément nucléiques), sont des mots sur l'alphabet à quatre lettres {A, C, G, T}. Une répétition est dite "en tandem" si plusieurs copies du même motif se suivent côte à côte. Une répétition en tandem approximative est une répétition en tandem qui a subi des altérations (des "mutations" en termes biologiques). Par exemple, la séquence suivante contient une Répétition en Tandem Approximative, ou RTA, de 9 copies du motif de base CATGG :

aattatcc CATGG CATaG CTGG CAtTGG ATGa CATGG CATGG CAaTGG CATcG aactac

Lorsqu'une RTA est détectée dans une séquence, une question importante qui se pose est : est-ce dû au hasard ? La question est essentielle pour ne pas donner de fausses interprétations au phénomène détecté. Des études statistiques ont en effet montré que tout motif apparaît dans une séquence suffisamment longue. Concrètement, une séquence aléatoire de longueur n contient, en moyenne, tout motif de longueur $\log n$. Elle a donc toutes les chances de contenir un certain nombre de RTA. L'idée d'analyse par compression propose une réponse à cette question : si le fait qu'une séquence S possède une RTA peut être exploité pour comprimer la séquence, alors ce n'est pas dû au hasard. Cette méthodologie d'analyse s'inspire de la théorie de la complexité de Kolmogorov [LV93] qui affirme que seules les véritables redondances des séquences permettent de les comprimer. Les redondances dûes au hasard n'apportent, en moyenne, pas de compression effective : la théorie de la complexité de Kolmogorov établit qu'il y a une chance sur 2^k qu'une séquence aléatoire se comprime de k bits. Donc, si l'on parvient à comprimer la séquence en utilisant la propriété, elle en est vraisemblablement la cause.

Si l'on tente de comprimer une séquence en exploitant la propriété d'"être une RTA du motif M donné", le long de toute la séquence, certains morceaux de la séquence sont rallongés et d'autres sont raccourcis. Les morceaux rallongés sont ceux qui ne ressemblent pas à une RTA de M, les autres sont ceux qui y ressemblent très fort.

Considérons par exemple le motif M = ATC et la séquence S suivante (les mots soulignés sont des copies approximatives du motif, les caractères en majuscules représentent des caractères qui sont en correspondance avec ceux du motif) :

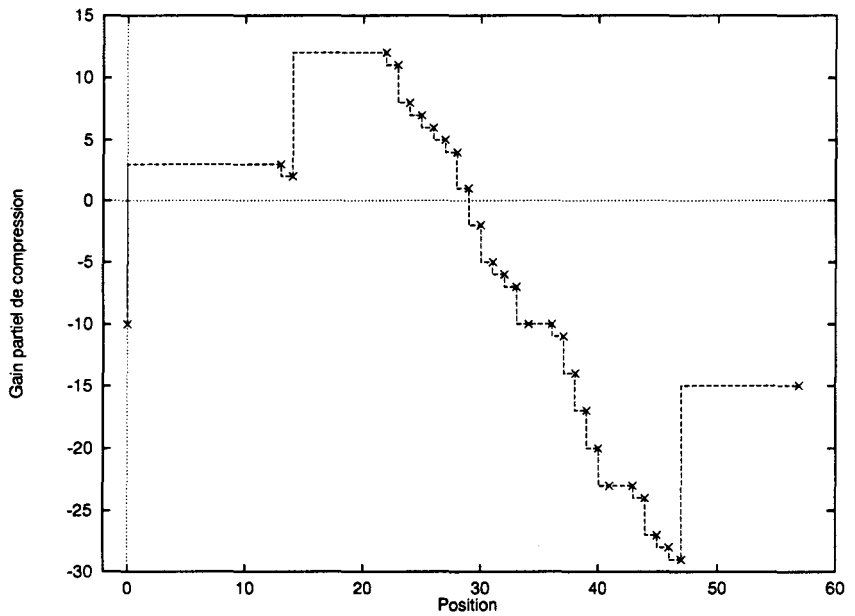
AT ATC ATC ATC ATgC ATC ATC A ggTtAgggcTggCAgaggtATggCc ATC ATC ATC A

Les morceaux constitués d'une suite de mots soulignés ressemblent à une répétition en tandem de M ; leur codage sera raccourci. Les autres morceaux n'y ressemblent pas, leur codage sera rallongé. Le problème auquel on s'intéresse est la détermination, de façon optimale (du point de vue de la compression de la séquence), des facteurs (morceaux de la séquence) où le phénomène de RTA a lieu. C'est-à-dire que l'on désire choisir, de façon optimale, les facteurs à comprimer en exploitant la propriété et les facteurs à coder sans exploiter la propriété. L'intérêt réside à la fois dans l'optimisation du gain global produit par la compression et la localisation précise des facteurs à qualifier de RTA.

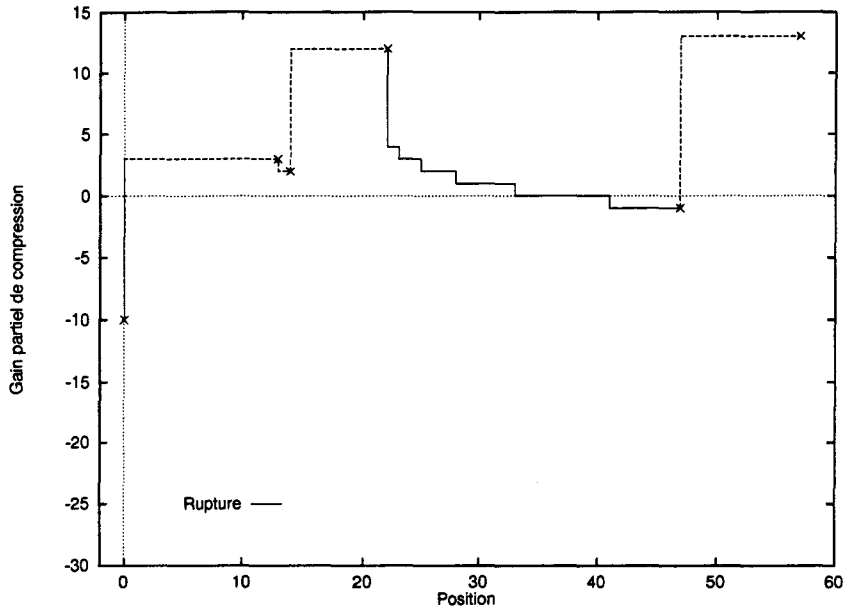
Nous obtenons un algorithme qui, pour une séquence de longueur n, fournit une compression optimale en temps $O(n \log n)$. La rapidité de l'algorithme le rend utilisable sans limitation pratique de la longueur de la séquence. De plus, on montre l'unicité de la solution (en un sens précisé par le théorème 3.9 du chapitre 3 de la partie II).

La principale difficulté du problème réside dans l'optimisation du codage de la délimitation des facteurs. Si l'on décide de ne pas exploiter la propriété le long d'un facteur, la séquence comprimée doit contenir la localisation précise du facteur. Le codage de cette localisation induit un coût qui tend à contrecarrer le bénéfice de notre choix. Intuitivement, il faut coder la longueur séparant deux facteurs consécutifs que l'on désire comprimer en exploitant la propriété. Le coût induit est égal au nombre de bits du codage de cette longueur.

Reprenons la séquence et le motif de l'exemple ci-dessus et traçons le graphique de la courbe de compression lorsque la propriété est exploitée partout. Ils s'agit de la courbe obtenue en plaçant, en abscisse, les positions de la séquence et en ordonnée, le nombre de bits économisés par exploitation de la propriété sur la partie gauche de la séquence. L'ordonnée correspondant à la plus grande abscisse est le gain total en compression. Les portions de courbe croissantes désignent les facteurs le long desquels la propriété est bien adaptée, la portion décroissante désigne le facteur le long duquel la propriété n'est pas adaptée :

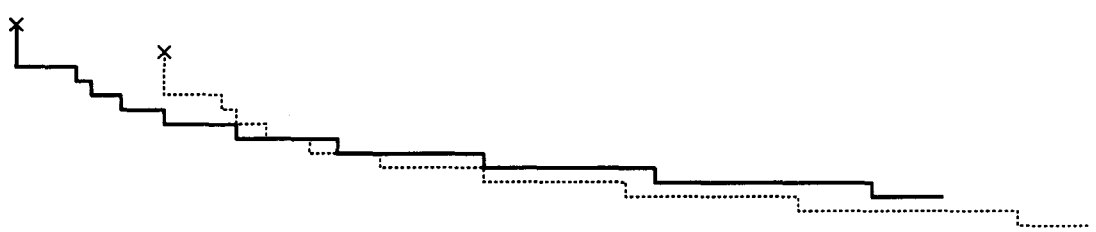


Idéalement, c'est-à-dire s'il ne fallait pas coder les délimitations des facteurs, la portion de courbe correspondant à un facteur le long duquel on décide de ne pas exploiter la propriété devrait être remplacée par un segment horizontal. En pratique, le coût de codage de la longueur l du facteur vaut de l'ordre de $\log l$ bits. La théorie de la complexité de Kolmogorov confirme cette intuition à ceci près que la longueur l du facteur doit être codée de manière auto-délimitée. Cela signifie que le nombre de bits sacrifiés pour coder la longueur l n'est pas connu d'avance, le code doit être tel que le décompresseur puisse le délimiter sans ambiguïté dans une suite de bits. Le coût de l'auto-délimitation est faible : la longueur du codage auto-délimité de l reste de l'ordre de $\log l$. La portion de courbe est donc remplacée par une portion de "courbe de rupture" qui possède une forme de "logarithme renversé discrétisé" :



Le caractère "logarithme renversé discrétisé" de la courbe de rupture constitue la propriété ICL du codage de la longueur (section 2.2.4.2.3, chapitre 2, partie I). Coder, à la fois de façon optimale, auto-délimitée, tout en préservant le caractère ICL du codage, ne va pas de soi. Aucune des techniques habituelles utilisées pour auto-délimiter les nombres entiers ne conjugue les trois propriétés. Nous développons un nouveau codage d'entiers, appelé *PrefFibo*. Nous démontrons (théorème 2.9, chapitre 2, partie I) que *PrefFibo* est à la fois asymptotiquement optimal, auto-délimité et ICL.

Il est évident que l'optimisation aveugle de la compression, par tentative de toutes les possibilités, est impraticable. Nous démontrons cela en établissant la borne supérieure exponentielle du nombre de cas à considérer (théorème 2.2, chapitre 2, partie II). L'efficacité de notre algorithme repose entièrement sur une exploitation minutieuse des propriétés des codes ICL qui fait qu'en tout point, il n'y ait qu'un nombre limité de choix à mémoriser. La principale propriété utilisée établit que lorsque deux courbes de ruptures potentielles se croisent en une abscisse, elles ne se croiseront plus pour des abscisses supérieures :



Toujours dans le cadre des répétitions en tandem approximatives, nous développons, dans la partie III, un algorithme d'alignement, formalisé en termes d'automates finis selon l'approche présentée par Searls et Murphy [SM95] et par Lefebvre [Lef96]. Cette approche est plus riche et plus élégante que les habituelles équations de récurrence dans lesquelles les méthodes classiques d'alignement par programmation dynamique sont exprimées. Il s'agit d'un algorithme permettant d'aligner une séquence avec une répétition en tandem exacte (c'est-à-dire qui n'a pas été sujette à des altérations) d'un motif. Notre approche rejoint celle de Fischetti, Landau, Schmidt et Sellers [FLSS92] issue de la programmation dynamique. La mise en œuvre

de l'application devient maintenant très claire :

1. Alignement, à l'aide de notre automate fini (plus précisément de notre transducteur), de la séquence avec une répétition en tandem exacte du motif : il s'agit de l'application aveugle de la propriété tout le long de la séquence.

2. Optimisation de la compression par liftings : on choisit de manière optimale les facteurs à comprimer en exploitant la propriété et les facteurs à ne pas comprimer.

Les répétitions en tandem approximatives ne sont qu'un exemple de propriété locale. Nous donnons, dans le chapitre 1 de la partie III, une définition générale de ce que peut être un codage modulaire. C'est le résultat, d'une méthode de compression appliquée de manière aveugle, auquel peut être appliquée notre optimisation par liftings. La compression résultant de l'optimisation est appelée une *compression modulaire optimale.* Dans ce cadre, notre algorithme permet d'une façon générale de localiser les régularités qui ne sont pas dues au hasard, c'est-à-dire qui permettent de comprimer. Le chapitre 1 de la partie III présente d'autres champs d'application de notre algorithme, par exemple pour optimiser des méthodes de compression classiques (Huffman, Ziv Lempel).

Dans le dernier chapitre du travail, nous esquissons rapidement une suggestion d'application de la méthode d'optimisation par liftings qui est très différente du reste. Il s'agit de localiser avec précision de longues répétitions approximatives dans une séquence. L'utilisation suggérée des liftings ne vise pas une optimisation globale du gain de compression mais plutôt une optimisation locale permettant de déterminer l'étendue à gauche et l'étendue à droite d'un facteur, possédant une propriété particulière, lorsqu'une position centrale du facteur est connue.

Parmi les perspectives de recherche liées à l'optimisation par liftings, citons :

- L'application pratique de la méthode à la localisation d'autres types de régularités que les répétitions en tandem approximatives. On peut par exemple envisager de comparer deux séquences génétiques à l'aide de la compression. L'algorithme d'optimisation par liftings nous permettrait de localiser les facteurs des deux séquences qui se ressemblent significativement.

- La recherche de plusieurs types de régularités dans une séquence. Lorsque l'optimisation par liftings a localisé les facteurs à comprimer par exploitation d'un type de régularité, une nouvelle analyse peut être entreprise, avec un autre type de régularité, sur les facteurs à ne pas comprimer.

  Par exemple, dans le contexte des RTA, lorsque les facteurs qualifiés de RTA d'un motif M ont été localisés dans la séquence, la méthode peut être appliquée, avec un second motif M' sur les facteurs qui n'ont pas été qualifiés de RTA.

  Cette démarche permet de localiser, en plusieurs étapes, les zones régulières de plusieurs types de régularités. Les résultats sont tributaires de l'ordre dans lequel les types de régularités sont envisagés.

- La généralisation théorique de notre méthode d'optimisation à l'optimisation de courbes continues par ruptures continues. Ce problème est présenté dans le chapitre 4 de la partie II.

En marge de notre travail de recherche, nous rappelons ici des notions utiles d'algorithmique sur les mots et de codage. Nous introduisons le minimum de notions génétiques nécessaire à la compréhension (biologie moléculaire, compression de séquences nucléiques, problématique biologique des répétitions en tandem approximatives, etc).

Nous dressons maintenant un plan des trois parties de la thèse.

Partie I : Préliminaires et Compression

La première partie de notre travail présente les concepts informatiques de base utilisés dans la suite. Le premier chapitre introduit formellement les notations utilisées dans le domaine de l'algorithmique des mots et les notations dites de Landau, à savoir O, Ω et θ, souvent utilisées en algorithmique pour évaluer l'ordre de grandeur du temps et de la mémoire nécessaires à l'exécution d'un algorithme. Le chapitre s'attarde sur la présentation de la structure d'arbre des suffixes, souvent mentionnée dans la suite du travail. C'est un outil très puissant mais la difficulté des méthodes qui permettent de le construire semblent limiter son utilisation au milieu de la recherche uniquement. Pour cette raison, l'annexe A contient une description, qui se veut pédagogique, de l'algorithme de construction de McCreight.

Le deuxième chapitre introduit les concepts de compression, de codage et de complexité de Kolmogorov. Le nouveau concept de code ICL y est notamment défini, il est utilisé de manière intensive dans la partie II. Actuellement, le seul code universel ICL permettant d'auto-délimiter les entiers est le code *Fibo* présenté par Apostolico et Fraenkel [AF87]. Malheureusement, il n'est pas asymptotiquement optimal. Nous prouvons qu'il est possible qu'un code permettant d'auto-délimiter les entiers peut être à la fois ICL et asymptotiquement optimal. Pour cela, nous présentons le nouveau code *PrefFibo*, auto-délimité, ICL et asymptotiquement optimal est présenté. Les méthodes classiques de compression conservative sont également rappelées dans ce chapitre.

Partie II : Optimisation de Courbes par Liftings

Cette partie constitue le cœur algorithmique de la thèse. Elle présente la méthode d'optimisation par liftings d'un point de vue général. La motivation principale est la compression de séquences mais la méthode est applicable, d'une manière générale, à des problèmes spécifiques d'optimisation de courbes. Le premier chapitre présente intuitivement la problématique, le second chapitre en présente la formalisation mathématique et le troisième développe l'algorithme d'optimisation TURBOOPTLIFT. La partie se termine par un chapitre qui mentionne quelques problèmes non résolus directement inspirés du problème principal d'optimisation.

Partie III : Application de l'Optimisation par Liftings à la Compression et à l'Analyse de Séquences Génétiques

La dernière partie concerne les applications de TURBOOPTLIFT, plus particulièrement dans le contexte de l'analyse de séquences génétiques. Dans le chapitre 1, nous montrons comment une méthode de compression existante peut être adaptée en vue d'une optimisation par liftings. Nous mettons en évidence les problèmes qui se posent, les compromis qui doivent être

trouvés et les méthodes qui peuvent être ainsi optimisées. Le but du second chapitre est d'inculquer quelques notions de biologie moléculaires aux lecteurs non biologistes. Il se contente de faire un vaste survol des quelques concepts et principes de base. Les concepts importants de séquence nucléique (ou séquence d'ADN) et d'alignements de séquences sont définis. La méthode classique d'alignement par programmation dynamique est présentée. L'analyse de séquences génétiques par compression n'est pas quelque chose de nouveau. Le chapitre 3 fait état des recherches menées dans ce domaine. Il met en évidence l'inadéquation des méthodes de compression classiques et effectue un large parcours des méthodes de compression dédiées aux séquences génétiques : la complexité linguistique de Trifonov [Tri90], la significativité algorithmique de Milosavljević et Jurka [MJ93, Mil93], BIOCOMPRESS et BIOCOMPRESS-2 de Grumbach et Tahi, CFACT et CPAL de Rivals [Riv94, Riv96, RDDD96, RDDD97] et enfin la méthode CBI de classification de séquences par compression de Loewenstern, Hirsh, Yianilos et Noordewier [LHYN95].

Le quatrième chapitre concerne le problème particulier de la localisation des répétitions en tandem approximatives. Nous présentons d'abord la problématique biologique des RTA, la programmation dynamique cyclique et la première méthode heuristique de Benson et Waterman [BW94]. Elle est basée sur un nombre de caractères minimal conservé dans chacune des copies approximatives du motif. Nous présentons alors la méthode heuristique de Eric Rivals utilisant la compression [Riv96]. Elle utilise une définition particulière des RTA, l'analyse complète d'une séquence d'ADN est effectuée par découpage de la séquence en fenêtres contiguës et analyses individuelles de chacune des fenêtres. Elle est limitée aux motifs très courts (de longueurs 1, 2 ou 3). Une large étude des chromosomes de levure est effectuée. Nous présentons en section 4.5 notre algorithme EXACTEMENTRTA qui utilise l'optimisation à l'aide de TURBOOPTLIFT. Il a l'avantage de ne souffrir d'aucun paramètre : les motifs sont de longueurs quelconques, la séquence est de longueur quelconque également et seule la compression permet de choisir les facteurs qui sont de vraies RTA et les facteurs qui n'en sont pas. La méthode est très rapide, elle permet de localiser des RTA dans des séquences de plusieurs centaines de milliers de nucléotides en quelques secondes. Elle semble bien adaptée pour localiser des régularités qui ne sont pas dues au hasard : des tests sont effectués sur des séquences engendrées aléatoirement et aucune RTA significative n'est détectée.

Le dernier chapitre esquisse un problème de biologie moléculaire qui semble pouvoir être résolu par une utilisation un peu particulière de l'algorithme d'optimisation par liftings. Il s'agit là, d'une perspective de recherche.

Première partie

Préliminaires et Compression

Chapitre 1

Préliminaires

Ce chapitre introduit formellement les concepts informatiques avec lesquels nous allons travailler dans la suite. Les éléments de base dans le domaine de l'algorithmique des mots sont tout d'abord définis : alphabet, mot (séquence), longueur de mot, facteur, préfixe, suffixe, concaténation, sous-mot,... Nous mentionnons alors brièvement les notations dites de Landau, à savoir O, Ω et θ, souvent utilisées en algorithmique pour évaluer l'ordre de grandeur du temps et de la mémoire nécessaires à l'exécution d'un algorithme. Ces notations permettront l'analyse des algorithmes décrits.

Le chapitre se termine par la présentation de l'arbre des suffixes ("Suffix Tree" dans la littérature). C'est un index compact qui permet d'accéder à tous les facteurs d'un mot en un temps raisonnable. Cet outil est très puissant, le nombre d'ouvrages qui en parlent est assez conséquent (voir par exemple [Wei73, McC76, MR80, Rod82, Apo85, AIL+88, Ukk92, Ukk93, CL94, CR94, Ste94, Ukk95, ALS96]). La section débute par la présentation du "trie" des suffixes, objet plus simple que l'arbre des suffixes mais moins compact. Vient ensuite le compactage du trie des suffixes pour obtenir l'arbre des suffixes. Nous terminons par l'utilisation de l'arbre des suffixes pour localiser les facteurs répétés d'une séquence. Une telle localisation nous sera très utile pour comprimer une séquence puisqu'elle met en évidence certaines redondances de la séquence qui méritent d'être supprimées. Le lecteur intéressé par un algorithme de construction de l'arbre des suffixes trouvera la description détaillée de celui de McCreight en annexe A.

1.1 Définitions formelles et notations

Soit $\mathcal{A} = \{a_1, a_2, \ldots, a_z\}$ un *alphabet*. C'est un ensemble fini, ses éléments $a_1, a_2, \ldots, a_z$ sont appelés des *lettres*, *symboles* ou *caractères*. Un *mot* x sur l'alphabet $\mathcal{A}$ est une suite de lettres de $\mathcal{A}$: $x = x_1 x_2 \ldots x_n$ avec $x_i \in \mathcal{A}, \forall i : 1 \leq i \leq n$. On dit également que x est une *séquence*. La *longueur* de x est notée $|x|$, c'est le nombre de ses lettres : $|x| = n$. Par convention, le *mot vide* est noté ϵ : $|\epsilon| = 0$.

L'*ensemble de tous les mots* que l'on peut former à partir de l'alphabet $\mathcal{A}$ est noté $\mathcal{A}^*$. On définit également l'*ensemble de tous les mots non vides* sur $\mathcal{A}$: $\mathcal{A}^+ = \mathcal{A}^* \backslash \{\epsilon\}$. L'*ensemble de tous les mots de longueur n* sur l'alphabet $\mathcal{A}$ est noté $\mathcal{A}^n$. Soit $X \subseteq \mathcal{A}^*$ un ensemble de mots, sa cardinalité est notée $\#X$ (toute cardinalité d'ensemble sera d'ailleurs notée de cette manière).

Dans la suite, $\mathcal{B} = \{0, 1\}$ désigne l'alphabet binaire. Ses éléments sont appelés des *bits*.

D'un point de vue terre à terre, tout objet informatique qui doit être mémorisé (programme, image, texte, nombre entier ou réel,etc) n'est rien d'autre qu'un mot sur l'alphabet $\mathcal{B}$.

La plupart du temps, les mots manipulés sont finis. Il nous arrivera cependant, dans le chapitre 4 de la partie III, de considérer des mots périodiques infinis. Dans ce cas, nous spécifierons que le mot est infini. Dans tous les autres cas, les mots sont finis.

Soit $x = x_1x_2 \dots x_n$ et $y = y_1y_2 \dots y_m$ deux mots de $\mathcal{A}^*$. Leur *concaténation*, notée $x.y$, est le mot de $\mathcal{A}^*$ obtenu en faisant suivre les lettres de x par les lettres de y: $x.y = x_1x_2 \dots x_ny_1y_2 \dots y_m$, sa longueur est $|xy| = |x| + |y| = n + m$. On omet souvent le point et on note xy pour $x.y$. La concaténation de k copies du même mot x est notée x^k.

Soit $x, y \in \mathcal{A}^*$. Le mot y est un *facteur* de x si il existe deux mots $u, v \in \mathcal{A}^*$ tel que $x = uyv$. De plus, si $u = \epsilon$, on dira que y est *préfixe* de x. De la même manière, y est *suffixe* de x si $v = \epsilon$. En d'autres termes, y est facteur de x s'il est égal à une suite contiguë de lettres de x. On note $y = x_{i..j}$ lorsque $y = x_ix_{i+1} \dots x_j$. Si la suite de lettre contiguës est extraite du début de x, alors y est préfixe de x: on note $y = x_{..i}$ si $y = x_1x_2 \dots x_i$. Lorsque y est extrait de la fin de x, y est suffixe de x: on note $y = x_{i..}$ lorsque $y = x_ix_{i+1} \dots x_n$. Le préfixe $y = x_{..i}$ de x est *préfixe propre* s'il n'est pas égal à x: $i < |x|$. De la même manière, un *suffixe propre* de x est un suffixe de x, différent de x.

On dit que le mot y possède une *occurrence* dans x à la position i si $y = x_{i..j}$. Ce concept permet de définir une *répétition* : le facteur y de x est *répété* s'il possède plusieurs occurrences dans x. En d'autres termes, y est une répétition si $y = x_{i_1..(i_1+l-1)} = x_{i_2..(i_2+l-1)} = \dots$, avec $1 \leq i_1 < i_2 \dots \leq n$ et $|y| = l$.

Le mot z est un *sous-mot* de x s'il peut être obtenu à partir de x en supprimant un certain nombre de lettres non nécessairement adjacentes: $z = x_{i_1}x_{i_2} \dots x_{i_m}$, avec $m \leq n$ et $1 \leq i_1 < i_2 < \dots i_m \leq n$.

Enfin, convenons que tout au long de ce travail, $\log i$ désigne le logarithme en base 2 de i. Pour une autre base (a par exemple), nous noterons $\log_a i$. La notation $\lfloor r \rfloor$ désigne le plus grand entier inférieur ou égal à r. De la même manière, $\lceil r \rceil$ désigne le plus petit entier supérieur ou égal à r.

Les concepts introduits dans cette section sont illustrés par l'exemple 1.1.

Exemple 1.1 *Considérons l'alphabet $\mathcal{A} = \{a, b, c, d, e\}$ et la séquence $x \in \mathcal{A}^*$:*

$$x = abceddaabaadeaaaccdabdeabaadeaadcee, |x| = 35$$

Le mot $y =$ abaade est un facteur répété de x, il possède deux occurrences aux positions 8 et 24 (ce sont les zones soulignées). Soit $u =$ abced. La concaténation de u et y est le mot $u.y =$ abcedabaade. Le mot u est préfixe de x : $u = x_{..5}$. Le mot $v =$ deaadcee est suffixe de x : $v = x_{28..}$. Le mot u^3 est abcedabcedabced. Le mot $z =$ edaaaadadc est un sous-mot de x : $z = x_4x_{6..7}x_{11}x_{14..15}x_{22}x_{31..33}$.

1.2 Notations de Landau

L'efficacité d'un algorithme se mesure en général en fonction de la taille du problème qu'il résout. Le temps réel mis par un programme pour s'exécuter n'est pas très intéressant parce qu'il dépend fortement de facteurs extérieurs tels que la machine utilisée, son occupation au moment de l'exécution, la qualité du compilateur utilisé,... L'intérêt porte plutôt sur l'ordre de grandeur du nombre d'opérations que le programme exécute lorsque la taille du problème

augmente. On parle dès lors d'algorithme linéaire (le nombre d'opérations augmente de façon linéaire avec la taille du problème), quadratique, logarithmique,...

Les *notations de Landau* permettent d'exprimer de manière concise cet ordre de grandeur. La plus fréquente, O donne une majoration de l'ordre de grandeur. La notation Ω en donne une minoration et la notation θ une borne inférieure et une borne supérieure.

Cette section est fortement inspirée de la présentation faite dans l'ouvrage "Éléments d'Algorithmique" de D. Beauquier, J. Berstel et Ph. Chrétienne [BBC92, Chapitre 2, pp.13–19].

1.2.1 Notation O

Soit $g : \mathbb{R} \to \mathbb{R}$ une fonction et $x_0 \in \mathbb{R} \cup \{-\infty, +\infty\}$. On désigne par $O(g)$ l'ensemble de toutes les fonctions f pour lesquelles il existe un voisinage V de x_0 et une constante $k > 0$ tels que $|f(x)| \leq k.|g(x)|, \forall x \in V$. Si la fonction g ne s'annule pas dans le voisinage V, cela revient au même de dire que le rapport $\left|\frac{f(x)}{g(x)}\right|$ est borné pour $x \in V$.

Pour l'évaluation de la complexité des algorithmes, nous nous intéressons uniquement à la comparaison de fonctions au voisinage de $+\infty$. On prend donc $x_0 = +\infty$ et tout voisinage de x_0 est un intervalle ouvert de la forme $]a, +\infty[$. Dès lors, $f \in O(g)$ au voisinage de $+\infty$ s'il existe deux nombres $k, a > 0$ tels que $|f(x)| \leq k|g(x)|, \forall x > a$.

Exemple 1.2 *Au voisinage de* $+\infty$, *on a* $x \in O(x^2)$, $\log x \in O(x)$, $x + 1 \in O(x)$.

Remarque 1.1 *On écrit* x *plutôt que* $f(x) : x \to x$ *pour alléger les notations. Il s'agit du premier abus d'écriture que nous utiliserons par la suite.*

Nous travaillerons toujours au voisinage de $+\infty$; ce sera toujours sous-entendu.

La définition de $O(g)$ reste valable lorsque le domaine des fonctions est restreint à l'ensemble $\mathbb{N}$ des entiers naturels. C'est le cas lorsque l'on étudie les performances d'algorithmes : la "taille" du problème résolu par l'algorithme est un entier. Dire que l'on se place au voisinage de l'infini revient à dire que l'on considère des tailles de problèmes très grandes. La notation O s'applique pour donner une majoration de l'ordre de grandeur du nombre d'opérations effectuées par l'algorithme.

Les conventions de notations de Landau permettent d'écrire, lorsque $f \in O(g)$: $f = O(g)$ ou $f(x) = O(g(x))$. De même, lorsque $O(f) \subset O(g)$, on peut écrire $O(f) = O(g)$ (ce sont de nouveaux abus d'écriture).

Il est important de remarquer que $f = O(g)$ n'implique pas $g = O(f)$. Pour s'en convaincre, prenons $f : x \to x$ et $g : x \to x^2$. On a $x = O(x^2)$ mais $x^2 \neq O(x)$.

Exemple 1.3 *Lorsqu'on écrit* $x^2 + 5x = O(x^2) = O(x^3)$, *cela signifie en fait* $x^2 + 5x \in O(x^2) \subset O(x^3)$.

En définissant

$$\begin{aligned} f + O(g) &= \{f + h | h \in O(g)\} \\ fO(g) &= \{fh | h \in O(g)\}, \end{aligned}$$

on a par exemple $h = f + O(g) \iff h - f \in O(g)$.

Si c est une constante, il est facile de montrer que :

$$\begin{aligned} c.O(f) &= O(f) \\ O(f)O(g) &= O(fg) \\ fO(g) &= O(fg) \end{aligned}$$

et si f et g sont toutes deux à valeurs positives, alors

$$O(f) + O(g) = O(f + g)$$

Exemple 1.4 *Pour tout polynôme* $P(x) = a_0x^k + a_1x^{k-1} + \cdots + a_k$, *il est facile de montrer que* $P(x) = O(x^k)$.

Exemple 1.5 *Considérons la fonction* TRI *qui trie le tableau* T *de longueur* n, *dont l'algorithme est le suivant (la fonction* SWAP *permet d'échanger deux valeurs) :*

```
TRI(T, n)
1  Pour i ← 1 Jusque n − 1
2      Faire Pour j ← i + 1 Jusque n
3          Faire Si T[i] > T[j]
4              Alors SWAP(T[i], T[j])
```

On dira que le temps d'exécution de TRI *est* $O(n^2)$. *En effet, on compte les opérations élémentaires effectuées par l'algorithme. On n'accorde aucune importance au temps d'exécution des différentes opérations élémentaires, cela n'apporte pas d'information substantielle [BBC92]. On suppose donc que* SWAP *s'exécute en temps constant (grâce aux notations de Landau, on écrit en temps* $O(1)$*).*

La boucle extérieure (d'indice i) est parcourue exactement $n - 1$ *fois. La boucle intérieure est parcourue* $n - 1$ *fois au maximum (*$n - 1$ *fois pour le premier tour de la boucle extérieure,* $n - 2$ *fois pour le second,...). Puisque chaque passage dans la boucle intérieure s'exécute en temps* $O(1)$, *l'algorithme s'exécute en temps* $O((n-1)^2) = O(n^2)$.

1.2.2 Notation Ω

La notation Ω est parfois utilisée pour décrire la complexité des algorithmes. Elle donne une minoration de l'ordre de grandeur d'une fonction.

On désigne par $\Omega(g)$ l'ensemble des fonctions f pour lesquelles il existe deux nombres $k, a > 0$ tels que $|f(x)| \geq k|g(x)|, \forall x > a$.

En d'autres termes, $f \in \Omega(g) \iff g \in O(f)$.

Comme pour la notation O, on écrit $f = \Omega(g)$.

Cette notation permet également d'établir la complexité minimale de tout algorithme devant résoudre un problème précis.

Exemple 1.6 *Le nombre de comparaisons de tout algorithme de tri d'un tableau de longueur* n *est* $\Omega(n \ln n)$ *[BBC92]. Cela signifie par exemple qu'il ne sera jamais possible de concevoir un algorithme général de tri fonctionnant en temps* $O(n)$.

1.2.3 Notation θ

La notation θ permet de spécifier que deux fonctions croissent "de façon comparable".

On désigne par $\theta(g)$ l'ensemble de fonctions f pour lesquelles il existe trois nombres $k_1, k_2, a > 0$ tels que $k_1|g(x)| \leq |f(x)| \leq k_2|g(x)|, \forall x > a$. Donc, si g ne s'annule pas, on a $k_1 \leq \frac{|f(x)|}{|g(x)|} \leq k_2, \forall x > a$. On écrit $f = \theta(g)$.

En d'autres termes, $f \in \theta(g) \iff f \in O(g)$ et $f \in \Omega(g)$.

Cela permet de donner un ordre de grandeur de façon précise.

Exemple 1.7 *Pour tout* $a, b > 1, \log_a n = \theta(\log_b n)$.

Il est facile de montrer que si $f = \theta(g)$, alors $g = \theta(f)$.

1.3 L'arbre des suffixes

Cette section est destinée aux lecteurs qui ne sont pas familiers avec le concept d'*arbre des suffixes*, elle en définit la structure. Nous commençons par la description de la structure de *trie des suffixes*, plus simple que celle d'arbre des suffixes mais également plus coûteuse en espace mémoire. Sa présentation nous permet de définir l'arbre des suffixes de manière progressive.

L'*arbre des suffixes* est une structure de données consistant en un index compact de tous les facteurs d'un texte. Cette structure peut être utilisée efficacement pour résoudre le problème classique de recherche de motifs dans un texte. Les méthodes habituelles de recherche d'un motif dans un texte effectuent un pré-traitement du motif et sont alors capables de rechercher le motif en $O(n)$ étapes où n est la longueur du texte [KMP77, BM77, CP91, CR94, Ste94]. Le pré-traitement du motif se fait en temps $O(m)$, m étant la longueur du motif. Si plusieurs motifs doivent être recherchés dans le même texte, chacun des motifs devra d'abord être pré-traité et l'ensemble de toutes les recherches prendra ensuite un temps $O(k.n)$ où k est le nombre de motifs.

Grâce à l'arbre des suffixes du texte, il est possible de rechercher la présence de n'importe quel motif dans le texte en un temps proportionnel à la longueur du motif. La recherche de plusieurs motifs peut donc se faire en un temps proportionnel à la somme des longueurs des motifs. De plus, l'arbre des suffixes permet de déterminer rapidement les facteurs répétés du texte.

Une idée similaire à celle de l'arbre des suffixes a d'abord été présentée implicitement par Morrison lorsqu'il a défini le *"Patricia Tree"* [Mor68],[Knu73b, pp. 490-493] (Practical Algorithm To Retrieve Information Coded In Alphanumeric); mais la description explicite de l'arbre des suffixes tel qu'il est utilisé à l'heure actuelle a été donnée par Weiner en 1973 [Wei73]. Dans cet article, un algorithme de construction de l'arbre des suffixes en $O(n)$ étapes est proposé. Cet algorithme procède par insertions, dans l'arbre des suffixes, de tous les suffixes du texte, en allant du plus court au plus long. La méthode de McCreight [McC76], possède une structure analogue à celui de Weiner mais il insère les suffixes en commençant par le plus long et en terminant par le plus court. Son temps d'exécution est également en $O(n)$ et son intérêt provient surtout de l'économie en espace réalisée par rapport à celui de Weiner.

Plusieurs efforts ont aussi été faits pour développer des algorithmes de construction *en ligne*, c'est-à-dire construisant l'arbre au fur et à mesure de la lecture, de gauche à droite,

des symboles du texte [MR80, Ukk92, Ukk95]. Parmi ceux-ci, celui de Ukkonen est le seul capable de construire l'arbre en temps $O(n)$.

Le but de cette section étant de présenter l'arbre des suffixes comme un outil informatique, nous ne donnons pas ici d'algorithme de construction. Le lecteur intéressé trouvera une description détaillée de l'algorithme de McCreight en annexe A.

De nombreux articles décrivent les propriétés et les applications de l'arbre des suffixes (voir par exemple [Rod82, Apo85, AIL+88, Ukk93, ALS96]). Nous parlerons ici de son utilisation comme outil de localisation des facteurs répétés d'une séquence.

1.3.1 Préliminaires sur les arbres

Soient p et f deux *nœuds* d'un arbre. Si un *arc* les relie, il est noté $arc(p, f)$. Cet arc est orienté de p vers f; on dira que p est le *père* et f le *fils* : $p = pere(f)$. Pour le nœud p, $arc(p, f)$ est un arc *sortant* mais pour f, c'est un arc *entrant*. La *racine* est le seul nœud n'ayant pas de père. Tous les autres nœuds ont un et un seul père mais un nœud peut avoir plusieurs fils. On note $fils(p)$ l'ensemble des nœuds fils de p : $f \in fils(p) \iff p = pere(f)$.

Par exemple, la figure 1.1 représente un arbre à 15 nœuds. Le nœud g_2 est le père des nœuds g_4, g_5 et g_6.

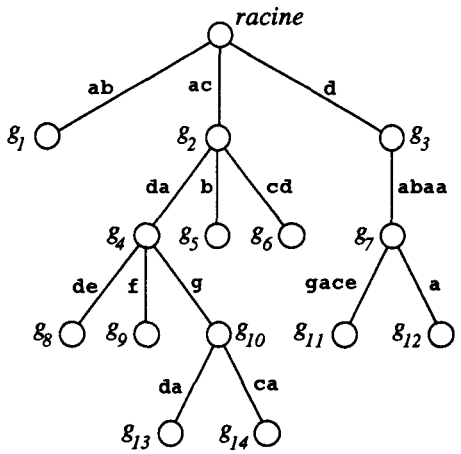


FIG. 1.1 - *Exemple d'arbre*

Le nœud g est *ancêtre* du nœud h si $h \in fils(g)$ ou si $h \in fils(f)$ avec g ancêtre de f (définition récursive). De cette manière, la racine est ancêtre de tous les autres nœuds de l'arbre. Pour notre exemple, g_2 est ancêtre de $g_4, g_5, g_6, g_8, g_9, g_{10}$, g_{13} et g_{14}. Si g est ancêtre de h, alors on dit également que h est *descendant* de g. Tout nœud est descendant de la racine excepté la racine elle-même.

Le *degré* d'un nœud p est le nombre de ses fils, on le note $degre(p)$. Un nœud f est une *feuille* s'il ne possède aucun fils : $degre(f) = 0$. Les nœuds de degré supérieur ou égal à 1 sont appelés des *nœuds internes*. Sur l'exemple, $g_1, g_5, g_6, \ldots$ sont des feuilles et $degre(g_4) = 3$.

Un *chemin* est une suite d'arcs consécutifs qui relient, pour chacun, un père avec un de ses fils. Ainsi, la suite $(a_1, a_2, \ldots a_n)$ d'arcs d'un arbre forme un chemin si $a_1 = arc(p_1, p_2), a_2 = arc(p_2, p_3), \ldots a_{n-1} = arc(p_{n-1}, p_n)$ et $a_n = arc(p_n, p_{n+1})$ avec $p_1, p_2, \ldots p_{n+1}$ des nœuds de l'arbre tels que $p_1 = pere(p_2), p_2 = pere(p_3), \ldots p_n = pere(p_{n+1})$. Par définition d'un arbre (il ne contient pas de "boucles"), s'il existe un chemin entre un nœud g et un nœud h, alors

ce chemin est unique. On le notera $chemin(g, h)$. Le nœud g sera donc ancêtre de h si et seulement si $chemin(g, h)$ existe. Sur notre exemple, $chemin(g_2, g_{14})$ existe mais ce n'est pas le cas pour $chemin(g_2, g_7)$.

Chacun des arcs de l'arbre peut être étiqueté par un mot. L'étiquette de l'arc $arc(p, f)$ est notée $etiq(arc(p, f))$. Dans notre exemple, les étiquettes des arcs sont indiquées juste à côté des arcs. Par extension, l'étiquette d'un chemin est la concaténation des étiquettes de tous les arcs qui le composent : $etiq(chemin(g, h)) = etiq(arc(g, g')).etiq(arc(g', g'')) \ldots etiq(arc(g^{(i)}, h))$. Le chemin $chemin(g_2, g_{14})$ de notre exemple est étiqueté **dagca**.

Puisqu'un nœud ne possède qu'un seul père, que la racine ne possède pas de père et que chaque arc représente un lien *père→fils*, le nombre d'arcs d'un arbre est égal au nombre de nœuds moins 1.

1.3.2 Le trie des suffixes

Étant donné un ensemble E de mots sur un alphabet $\mathcal{A}$, son *trie* (pour "information re*trie*val", sans traduction en français) [Knu73b] est l'arbre possédant les six propriétés suivantes.

1. Chaque arc est étiqueté par un symbole de $\mathcal{A}$.
2. Les étiquettes des arcs menant d'un nœud père vers ses nœuds fils sont toutes différentes : pour tout nœud p, si $f_1, f_2 \in fils(p)$ et $etiq(arc(p, f_1)) = etiq(arc(p, f_2))$, alors $f_1 = f_2$.
3. Il existe deux types de nœuds : les nœuds *terminaux* et les nœuds *non terminaux*. Les feuilles de l'arbre sont toujours des nœuds terminaux.
4. L'étiquette de tout chemin menant de la racine à un nœud terminal est un mot de E : pour tout g, nœud terminal, $etiq(chemin(racine, g)) \in E$.
5. Tout mot de E est l'étiquette d'un chemin allant de la racine à un nœud terminal de l'arbre : pour tout mot w de E, il existe un nœud terminal g de l'arbre tel que $etiq(chemin(racine, g)) = w$.
6. L'étiquette de tout chemin menant de la racine à un nœud non terminal est un préfixe d'un mot de E : pour tout g, nœud non terminal, il existe un mot $w \in E$ tel que $etiq(chemin(racine, g))$ est préfixe de w.

La figure 1.2 représente le trie de $E = \{\texttt{A}, \texttt{CA}, \texttt{GC}, \texttt{GG}, \texttt{GGA}, \texttt{GGGCA}, \texttt{GTG}, \texttt{GTT}, \texttt{TC}, \texttt{TCA}, \texttt{TCT}\}$ sur l'alphabet $\mathcal{N} = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. Les nœuds terminaux sont représentés par des cercles grisés.

Étant donné le mot $y = y_1 y_2 \ldots y_m$, on vérifie son appartenance à E en vérifiant s'il est l'étiquette d'un chemin de la racine à un nœud terminal. Pour cela, il suffit, à partir de la racine, de "se déplacer" dans l'arbre en suivant les arcs imposés par les symboles de y, lus de gauche à droite. Si le chemin n'existe pas parce qu'un nœud intermédiaire ne possède pas le fils adéquat, alors y n'appartient pas à E. C'est le cas, dans l'exemple, avec $y =$ **GTA**. Si le chemin conduit à un nœud non terminal, alors y n'appartient pas à E mais est préfixe d'un mot de E. Par exemple, $y =$ **GGG** est préfixe du mot **GGGCA** de E. Si le chemin atteint un nœud terminal, alors y appartient à E. La vérification d'appartenance se fait donc en temps $O(m)$.

On remarque facilement que si tous les mots de E se terminent par le même symbole n'apparaissant nulle part ailleurs dans les mots de E, alors tous les nœuds terminaux sont des feuilles. En effet, aucun mot de E n'est le préfixe d'un autre mot de E.

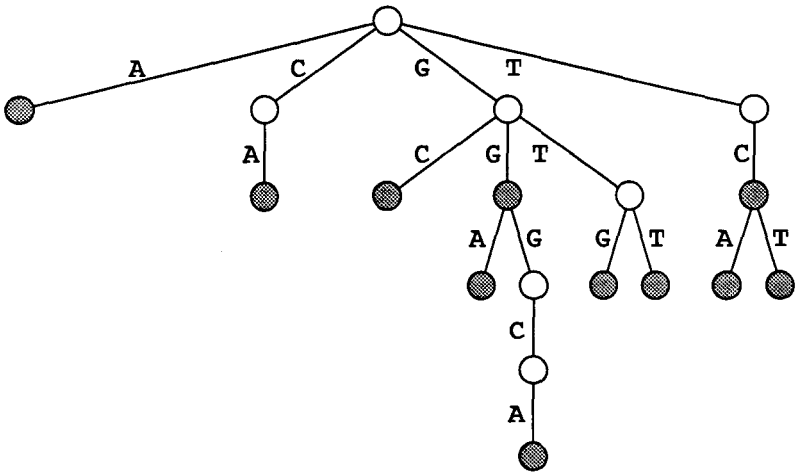


FIG. 1.2 - *Trie de l'ensemble E*

Soit $x = x_1 x_2 \dots x_n \in \mathcal{A}^+$ un mot. Supposons que son dernier symbole soit différent de tous les autres. Si ce n'est pas le cas, il suffit d'étendre l'alphabet à $\mathcal{A} \cup \{\$\}$, avec $\$ \notin \mathcal{A}$, et de considérer le mot x auquel on concatène le symbole \$. Le *trie des suffixes* de x, noté $STrie(x)$, est le trie de l'ensemble S_x constitué de tous les suffixes de x.

Si $x =$ `ACTACT$`, alors $S_x = \{$`ACTACT$, CTACT$, TACT$, ACT$, CT$, T$, $`$\}$ et $STrie(x)$ est l'arbre représenté à la figure 1.3.

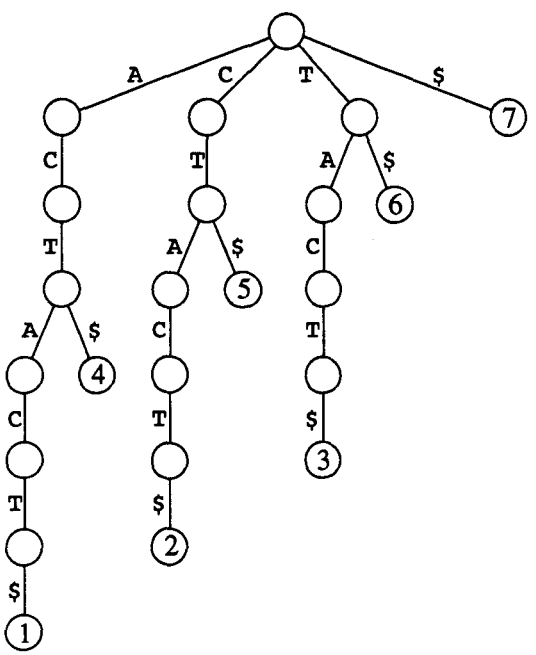


FIG. 1.3 - *STrie(ACTACT$)*

Puisque le dernier symbole de x est différent de tous les autres, aucun suffixe n'est le préfixe d'un autre suffixe et donc les nœuds terminaux de $STrie(x)$ sont tous situés sur les feuilles. Dès lors, plutôt que de représenter les nœuds terminaux par des cercles grisés, nous

plaçons, dans chaque feuille, la position de début du suffixe concerné.

Étant donné un mot $y = y_1 y_2 \dots y_m$, on peut tester, en maximum m étapes, s'il est facteur de x. En effet, s'il correspond à un chemin partant de la racine et se terminant sur une feuille, c'est un suffixe de x. Si le chemin se termine sur un nœud interne, alors y est préfixe d'un suffixe de x et donc facteur de x. Si le chemin n'existe pas, alors y n'est pas facteur de x.

Donc, il y a non seulement correspondance bi-univoque entre les feuilles et les suffixes de x mais, de plus, les nœuds internes correspondent aux facteurs de x. Pour tout nœud g, on peut donc définir $fact(g)$ comme l'étiquette du chemin menant de la racine à g. Par définition, $fact(racine) = \epsilon$. Symétriquement, étant donné un mot w, son *locus* est le nœud g du trie des suffixes pour lequel $etiq(chemin(racine, g)) = w : locus(fact(g)) = g$. Il est important de remarquer que le locus n'existe dans le trie que pour les facteurs du mot.

La proposition 1.1 montre que l'information d'un nœud de degré 1 peut être immédiatement déduite de l'information de son unique fils. Par contre, la proposition 1.2 montre qu'un nœud interne de degré supérieur ou égal à 2, ou une feuille, apporte de l'information supplémentaire. De ce fait, ces derniers nœuds sont communément appelés *nœuds importants* [CR94, Riv96].

Proposition 1.1 *Soit x un mot. Soient g et f deux nœuds de $STrie(x)$ tels que $degre(g) = 1$ et $pere(f) = g$. Supposons que $fact(f)$ soit connu, alors grâce à $etiq(arc(g, f))$ on déduit immédiatement $fact(g)$.*

Preuve. Par construction de $STrie(x)$, on a $fact(g).etiq(arc(g, f)) = fact(f)$. □

Proposition 1.2 *Soit $x = x_1 x_2 \dots x_n$ un mot, avec $x_n \neq x_i, \forall i : 1 \leq i < n$.*

1. *Soit h une feuille de $STrie(x)$. Alors $fact(h)$ est un suffixe de x.*
2. *Soit g un nœud de $STrie(x)$ avec $degre(g) \geq 2$ et $l = |fact(g)|$. Soient $f_1, f_2 \in fils(g)$ avec $etiq(arc(g, f_1)) \neq etiq(arc(g, f_2))$. Il existe deux suffixes $x_{i..}$ et $x_{j..}$ de x tels que :*
   - *$fact(g)$ est le préfixe de longueur l à la fois de $x_{i..}$ et de $x_{j..}$.*
   - *$fact(f_1)$ est le préfixe de longueur $l + 1$ de $x_{i..}$ mais n'est pas préfixe de $x_{j..}$.*
   - *$fact(f_2)$ est le préfixe de longueur $l + 1$ de $x_{j..}$ mais n'est pas préfixe de $x_{i..}$.*

Preuve.

1. $fact(h)$ est un suffixe de x par définition de $STrie(x)$.
2. Les mots $fact(g), fact(f_1)$ et $fact(f_2)$ sont tous trois préfixes de suffixes de x par construction de $STrie(x)$. Donc il existe $i \leq j \leq n$ tels que $fact(f_1) = x_{i..i+l}$ et $fact(f_2) = x_{j..j+l}$. Puisque $fact(f_1) = fact(g).etiq(arc(g, f_1))$, $fact(g)$ est le préfixe de longueur l de $x_{i..}$. De la même façon, $fact(g)$ est le préfixe de longueur l de $x_{j..}$.

   Puisque $l = |fact(g)|$, on a $x_{i..i+l-1} = x_{j..j+l-1}$. Puisque $x_{i+l} = etiq(arc(g, f_1)) \neq etiq(arc(g, f_2)) = x_{j+l}$, $x_{i..i+l}$ n'est pas préfixe de $x_{j..}$ et inversement. □

Chaque nœud interne, de degré supérieur ou égal à 2 correspond donc à un facteur répété *maximal à droite*. On entend par là que chacune des occurrences de la répétition possède un contexte droit différent : $x_{i..i+l-1} = x_{j..j+l-1}$ mais $x_{i+l} \neq x_{j+l}$. La répétition ne peut donc pas être étendue à droite pour chacune des deux occurrences.

Le trie des suffixes est un outil très puissant mais malheureusement, coûteux en espace mémoire. En effet, dans le pire des cas, par exemple lorsque tous les symboles de x sont différents, il n'existe aucun préfixe commun à deux suffixes quelconques de x. Dès lors, le nombre d'arcs du trie sera égal à la somme des longueurs de tous les suffixes de x, c'est-à-dire $n + (n-1) + (n-2) \ldots + 1 = \frac{n^2+n}{2}$ (voir figure 1.4).

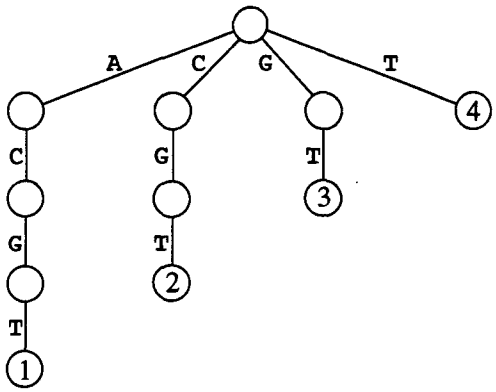


FIG. 1.4 - *STrie(ACGT)*

1.3.3 Du trie des suffixes à l'arbre des suffixes

La séparation des nœuds du trie des suffixes en nœuds *importants* et autres nœuds permet d'en réduire le nombre de nœuds.

Soit g un nœud de degré 1 de $STrie(x)$. Soit $p = pere(g)$. Soit h le plus proche nœud important descendant de g. Ce nœud existe et est unique, dans le pire des cas, c'est une feuille. Le chemin $chemin(p,h)$ ne contient que des nœuds de degré 1 : $chemin(p,h) = (arc(p,g), arc(g,f_1), arc(f_1,f_2), \ldots, arc(f_i,h))$ et $degre(g) = degre(f_1) = \ldots = degre(f_i) = 1$ (voir figure 1.5).

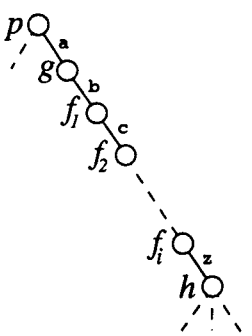


FIG. 1.5 - *Chemin composé de nœuds de degré 1*

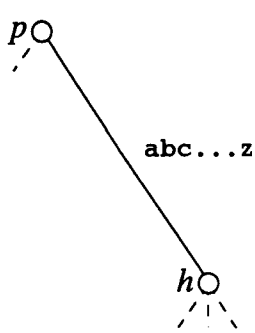


FIG. 1.6 - *Chemin de la figure 1.5 compacté*

Grâce à la proposition 1.1, l'information de g peut être déduite de l'information de f_1 qui peut elle-même être déduite de l'information de $f_2 \ldots$ l'information de f_i qui peut être déduite de l'information de h. Dès lors, on compacte $chemin(p,h)$ en un seul arc en supprimant tous les

nœuds intermédiaires de degré 1. L'étiquette du nouvel arc est la concaténation des étiquettes des arcs supprimés : $etiq(arc(p,h)) = etiq(arc(p,g)).etiq(arc(g,f_1))\ldots\ etiq(arc(f_i,h))$ (voir figure 1.6).

L'itération de ce compactage sur tous les chemins entièrement composés de nœuds de degré 1 produit l'*arbre des suffixes*, noté $ST(x)$. Dans $ST(x)$, tous les nœuds internes sont de degré supérieur ou égal à 2 (excepté éventuellement la racine d'un arbre des suffixes trivial). Chaque arc de $ST(x)$ est étiqueté par un facteur de x. Étant donné un nœud p de $ST(x)$, $fact(p)$ est égal à la concaténation des étiquettes des arcs constituant le chemin $chemin(racine,p)$.

Par construction, pour un nœud interne p de $ST(x)$, les étiquettes des arcs menant à chacun de ses fils commencent toutes par un symbole différent.

Bien entendu, la diminution du nombre de nœuds a provoqué un accroissement de la quantité d'information à stocker pour chaque arc (l'étiquette peut contenir plusieurs lettres). Si on suppose la connaissance de x, l'étiquette de chaque arc $arc(p,h)$ peut être remplacée par le couple (i,j) tel que $etiq(arc(p,h)) = x_{i..j}$.

Remarque 1.2 *Puisqu'un nœud de degré supérieur ou égal à 2 correspond à un facteur répété, le couple (i,j) est choisi arbitrairement parmi les positions de début et de fin des occurrences de la répétition.*

La figure 1.7 présente l'arbre des suffixes correspondant au trie des suffixes de la figure 1.3. Les étiquettes des arcs figurent sur le graphique tout comme les couples de positions (i,j) qui délimitent les facteurs.

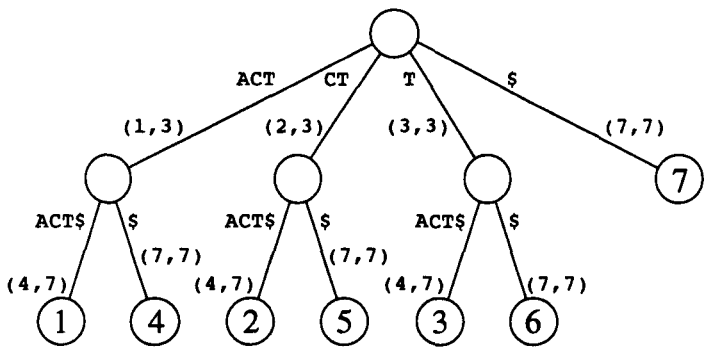


FIG. 1.7 - $ST(ACTACT\$)$

La proposition suivante établit une borne au nombre de nœuds de l'arbre des suffixes.

Proposition 1.3 *Soit $x = x_1x_2\ldots x_n$. le nombre maximal de nœuds de $ST(x)$ est $2n-1$.*

Preuve. Puisqu'il y a n suffixes, $ST(x)$ contient exactement n feuilles. Par compactage de $STrie(x)$, tous les nœuds internes de $ST(x)$ ont un degré supérieur ou égal à 2. Il y a donc au plus $\lfloor \frac{n}{2} \rfloor$ pères de feuilles. Il y a au plus $\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor$ "grand-pères" de feuilles... L'arbre $ST(x)$ contient donc au maximum $\lfloor \frac{n}{2} \rfloor + \left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \right\rfloor}{2} \right\rfloor \ldots + 1 \leq n-1$ nœuds internes.

La comptabilisation des feuilles donne $2n-1$ nœuds au maximum. □

Soit $y = x_{i..j}$ un facteur de x. Si, dans $STrie(x)$, $locus(y)$ était un nœud de degré 1, alors $locus(y)$ n'existe pas dans $ST(x)$. On définit alors le *locus contracté* du facteur y dans $ST(x)$

comme étant le locus du plus long préfixe de y représenté dans $ST(x)$. Sur l'exemple de la figure 1.7, le locus contracté de $y =$ `ACTA` est $locus($`ACT`$)$. Le *locus étendu* de y est le locus du plus court facteur de x, représenté dans $ST(x)$, dont y est un préfixe. Sur l'exemple de la figure 1.7, le locus étendu de $y =$ `CTA` est $locus($`CTACT$`$)$, c'est-à-dire la feuille 2.

Tout comme pour le trie des suffixes, vérifier si $y = y_1 y_2 \dots y_m$ est un facteur de x, se fait en temps $O(m)$. Il suffit, à partir de la racine, de descendre parmi les fils en suivant les arcs imposés par les symboles de y, lus de gauche à droite. Chaque descente d'un nœud père p vers son fils f impose la lecture, dans y de $|etiq(arc(p, f))|$ symboles. À chaque nœud, six cas peuvent se produire :

1. Il y a concordance entre le facteur $etiq(arc(p, f))$ et les $|etiq(arc(p, f))|$ symboles courants de y. Dans ce cas, **la descente doit continuer** vers le fils f. C'est le cas du parcours de `ACT` pour $y =$ `ACTA` dans l'exemple de la figure 1.7.

2. Le nœud courant est une feuille et les symboles de y ont tous été parcourus. On conclut que y **est un suffixe de** x. C'est le cas de $y =$ `ACT$` après la descente de deux arcs pour notre exemple.

3. Le nœud courant n'est pas une feuille mais tous les symboles de y ont été parcourus. Alors y **est un facteur répété maximal à droite de** x. C'est le cas de $y =$ `ACT` après la descente d'un arc pour notre exemple.

4. Tous les symboles de y ont été parcourus mais seuls les premiers symboles de l'étiquette de l'arc courant ont été lus. Dans ce cas, y **est un facteur de** x, le dernier nœud parcouru dans $ST(x)$ est le locus contracté de y. C'est le cas de $y =$ `ACTAC` après descente d'un arc et parcours des deux symboles `AC` qui terminent y.

5. La comparaison des symboles courants de y avec ceux du facteur $etiq(arc(p, f))$ provoque une discordance. Le mot y **n'est alors pas un facteur de** x. C'est le cas de $y =$ `ACA` lors de la descente dans le premier arc.

6. Le nœud courant ne possède pas le fils correspondant au caractère courant de y. Le mot y **n'est alors pas un facteur de** x. C'est le cas de $y =$ `ACTCA` après la descente dans le premier arc : $locus($`ACT`$)$ ne possède pas de fils pour le symbole `C`.

L'information de l'arbre des suffixes permet donc de tester si un mot est un facteur de x en un temps équivalent au test utilisant le trie des suffixes.

1.3.4 Structure de données pour le stockage de l'arbre des suffixes

Puisqu'on travaille avec des arbres, chaque nœud (excepté la racine) est le fils d'un et d'un seul père. Dès lors, l'étiquette d'un arc (en fait les deux indices, dans x, qui délimitent l'étiquette de l'arc) peut être stockée dans le nœud fils de chaque arc *père→fils*.

Pour le stockage des arcs eux-mêmes, deux cas sont à distinguer :

1er cas : L'alphabet contient un nombre indéterminé ou un grand nombre de lettres. Dès lors, le nombre de fils peut varier fortement d'un nœud à l'autre.

Une structure possible est celle dans laquelle chaque nœud possède un pointeur vers son premier fils et un pointeur vers son frère droit. La figure 1.8 montre cette structure de données pour l'arbre des suffixes de la figure 1.7 dans le cas où l'alphabet n'est pas

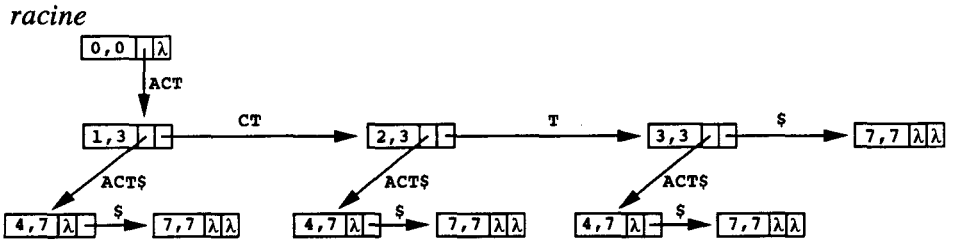


FIG. 1.8 - *Structure de données pour un alphabet inconnu*

connu ou comporte trop de lettres. Chaque pointeur *nul* est représenté par λ. L'accès à un fils se fait en temps $O(\#\mathcal{A})$ puisqu'il faut rechercher le fils adéquat dans la liste des fils. Dans ce cas, le temps maximal de recherche d'un facteur dans l'arbre des suffixes est à multiplier par $O(\#\mathcal{A})$.

Une autre possibilité est de placer tous les fils d'un même nœud dans une structure de recherche permettant l'accès à un fils en temps $O(\log \#\mathcal{A})$.

2ème cas : L'alphabet est déterminé, il contient un petit nombre de lettres. C'est le cas par exemple des séquences d'ADN : $\mathcal{N}_\$ = \mathcal{N} \cup \{\$\} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \$\}$. Dans ce cas, chaque nœud peut contenir un pointeur pour chacun de ses fils potentiels. Puisque l'étiquette de chaque arc commence par une lettre différente, il faut prévoir un pointeur par lettre de l'alphabet. On suppose alors que l'accès à un fils se fait en temps constant.

La figure 1.9 montre la structure de données associée à l'arbre des suffixes de la figure 1.7 dans le cas où l'alphabet est déterminé et contient peu de lettres (en l'occurrence, $\mathcal{N}_\$ = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \$\}$).

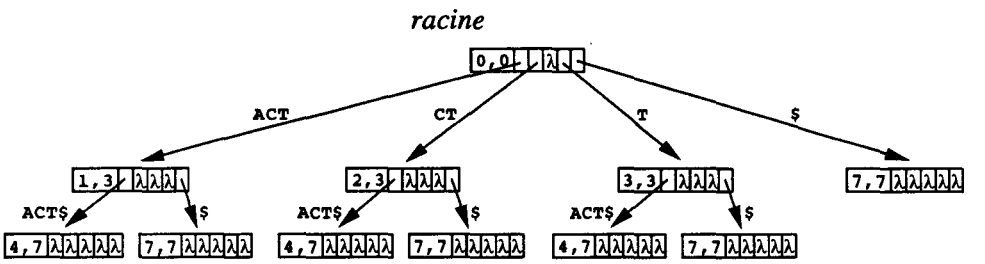


FIG. 1.9 - *Structure de données pour un alphabet petit et déterminé*

1.3.5 Utilisation pour localiser les facteurs répétés dans une séquence

Puisque chaque nœud interne de l'arbre des suffixes $ST(x)$ est de degré supérieur ou égal à deux, c'est un facteur répété maximal à droite : chacune des occurrences de la répétition possède un contexte droit différent. La répétition ne peut pas être étendue à droite.

Grâce à l'arbre des suffixes, il est aisé de localiser toutes les occurrences d'un facteur maximal à droite.

Soit f un facteur répété maximal à droite, il possède un locus dans l'arbre : $g = locus(f)$. Soient $g_1, g_2, \ldots g_k$ toutes les feuilles du sous-arbre dont g est la racine :

- Ce sont des feuilles : $\forall g_i : 1 \leq i \leq k : degre(g_i) = 0$.

- g est leur ancêtre commun : $\forall g_i : 1 \leq i \leq k : g$ est ancêtre de g_i.
- Toutes les feuilles sont considérées : il n'existe pas de feuille h de l'arbre telle que g est ancêtre de h et $h \notin \{g_1, g_2, \ldots g_k\}$.

Chaque nœud g_i est le locus d'un suffixe de x. Par construction de l'arbre des suffixes, $fact(g)$ est préfixe de chacun des suffixes $fact(g_1), fact(g_2), \ldots fact(g_k)$ et par conséquent, les positions d'occurrence du facteur répété $fact(g)$ sont les positions de début des suffixes $fact(g_i)$. Étant donné un nœud interne, les positions de ses occurrences sont trouvées par recherche récursive des positions des occurrences de toutes les feuilles dont il est ancêtre.

Supposons qu'un facteur f répété ne soit pas maximal à droite, alors les positions de chacune de ses occurrences sont égales aux positions des occurrences du facteur de son locus étendu.

Les facteurs répétés maximaux sont importants dans le contexte de la compression. On code en effet la deuxième (ou une des suivantes) occurrence d'un facteur répété par un pointeur qui identifie la première occurrence (pour plus de précisions, voir le chapitre suivant). Plus long est le facteur répété, meilleur est le gain produit par le codage du pointeur. Un facteur répété maximal à droite est donc potentiellement plus intéressant qu'un des facteurs répétés qu'il contient comme facteur.

Il est possible de déterminer la liste des positions de toutes les occurrences des facteurs répétés maximaux à droite par simple parcours récursif de l'arbre. Chaque nœud est visité une seule fois, le processus donc linéaire en temps.

Chapitre 2

Compression et complexité de Kolmogorov

Notre vie deviendrait un cauchemar
si nous ne négligions pas
les petites probabilités défavorables
dans nos algorithmes quotidiens

Andreï Nikolaïevitch KOLMOGOROV (1903-1987)

Il n'est pas de semaines sans que l'on découvre de nouvelles applications à la compression informatique [Del95]. Elle fait partie intégrante du monde des utilisateurs de micro-ordinateurs, stations de travail ou mainframe que ce soit volontairement (en actionnant un compresseur spécifique) ou involontairement (certains formats de fichiers tels que GIF ou JPEG sont automatiquement comprimés; des programmes "compresseurs de disque dur" compriment systématiquement toutes les informations que l'on vient écrire sur les disques).

Bien que les prix des mémoires et des disques diminuent sans cesse, la compression est de plus en plus utilisée car la quantité d'informations que l'on désire stocker sur son ordinateur grandit de jour en jour. La compression permet également de réduire les temps de communication de l'information par ligne téléphonique, satellite ou par réseau. Grâce à la compression, l'information devient moins coûteuse à transmettre.

Une deuxième utilisation de la compression, moins commune, est l'analyse de données. En effet, la compression exige une certaine compréhension de ce qui doit être comprimé. Mieux les données sont comprises, mieux elles sont comprimées. Les régularités des données sont exploitées pour comprimer. La réduction de longueur obtenue sur une séquence est une mesure de la régularité des données. Cette utilisation de la compression est basée sur la théorie algorithmique de l'information.

Dans ce chapitre, nous nous intéressons à la problématique générale de la compression de séquences. Nous nous restreignons aux méthodes conservatives (sans perte d'informations). Dans la première section, les contextes d'utilisation de la compression sont présentés. Les limitations pratiques évidentes de la compression sont également discutées. Viennent alors les définitions de gain et taux de compression permettant l'évaluation des performances d'un compresseur pour une séquence donnée. Une méthode de compression commence par détecter les régularités des données et procède alors au codage efficace des régularités. Le but de la section 2.2 est de présenter les techniques de codage. Le concept central de cette section est

le code. Ce n'est rien d'autre qu'une fonction de traduction de mots source en mots code. L'accent est placé sur l'auto-délimitation d'un code qui permet un décodage rapide et non ambigu. Les bases de la théorie de l'information de Shannon [Sha48, SW49] sont exposées : entropie, longueur moyenne des mots code, "Noiseless Coding Theorem"... Les propriétés générales des codes, telles que l'optimalité et l'universalité sont introduites. La section 2.2 se termine par un vaste parcours des méthodes de codage de nombres entiers. Des méthodes classiques telles que le codage de Fibonacci sont développées. Nous présentons alors une nouvelle propriété que peuvent posséder des méthodes de codage de nombres entiers : la propriété ICL (*Increasing, Concave and Limited*). Cette propriété concerne les codes à longueur variable qui peuvent coder des entiers arbitrairement grands. Elle impose des contraintes spécifiques sur l'accroissement de la longueur des mots code en fonction de l'accroissement des entiers. L'utilité de cette propriété n'apparaît pas clairement dans le contexte de la compression de séquences mais elle est à la base de l'algorithme d'optimisation que nous présentons dans la deuxième partie de ce travail : elle permet de créer des fonctions de rupture qui sont DCL (*Décroissantes, Convexes et Limitées*). On montre que le codage de Fibonacci est auto-délimité et ICL. Malheureusement il n'est pas asymptotiquement optimal. Les codes, présentés dans la littérature [Eli75, ER78, AF87, Sto88, BCW90, LV93], ne sont jamais auto-délimités, ICL et asymptotiquement optimaux à la fois. La section 2.2 se termine par la construction du code *PrefFibo* qui possède les trois propriétés.

La section 2.3 présente rapidement les méthodes classiques de compression conservative telles que le codage de Huffman, le codage arithmétique et les méthodes par substitutions de facteurs LZ77 et LZ78. Ce sont des méthodes polyvalentes qui sont implémentées dans beaucoup de programmes disponibles dans le commerce et dans le domaine public.

Le chapitre se termine par une brève introduction du concept central de la théorie algorithmique de l'information : la complexité de Kolmogorov. Le but de cette dernière section est de présenter les idées de base de cette théorie pour justifier l'utilisation de la compression comme outil d'analyse de données. La présentation est réalisée à l'aide d'exemples.

2.1 Généralités sur la compression

Nous présentons brièvement la problématique de la compression de séquence. Nous commençons par une description générale des différents types de compresseurs et des différents contextes d'utilisation de la compression. Les limitations théoriques de la compression sont brièvement exposées en référence à la théorie de la complexité de Kolmogorov. Les principes de cette théorie sont exposés plus profondément à la section 2.4. Pour évaluer les performances de compresseurs, nous définissons les notions de gain de compression et taux de compression. Nous terminons la section par un schéma général d'un processus de compression.

2.1.1 Utilisations et limitations de la compression de séquences

La compression a pour but de réduire la quantité de symboles nécessaires à la représentation de l'information.

Le processus chargé de réduire la taille d'une séquence est appelé un *algorithme de compression* ou une *méthode de compression* ou encore un *compresseur*. De la même manière, un *algorithme de décompression* ou une *méthode de décompression* ou enfin un *décompresseur* effectue le travail inverse : il reconstruit la séquence initiale à partir de la séquence comprimée.

Soit S une séquence à comprimer et $\mathcal{C}$ une méthode de compression. Soit S' la séquence S comprimée à l'aide de $\mathcal{C}$. On note $S \xrightarrow{\mathcal{C}} S'$ ou encore $S' = \mathcal{C}(S)$. La méthode de compression $\mathcal{C}$ ne présente de l'intérêt que s'il existe une méthode de décompression $\mathcal{D}$ permettant, à partir d'une séquence comprimée, de reconstruire la séquence initiale. Il faut distinguer deux cas :

1. $\mathcal{D}(S') = S$: la compression $\mathcal{C}$ est dite *conservative*. Après décompression, la séquence initiale est reconstruite.
2. $\mathcal{D}(S') = S''$ avec S'' "proche" de S : la compression $\mathcal{C}$ est dite *non-conservative*. Après décompression, une approximation de la séquence initiale est construite.

Si, lorsque l'on comprime un texte ou une base de données, il est indispensable que la décompression retrouve avec exactitude le texte ou la base de données (la compression est conservative ou "lossless" en anglais), il n'en est pas de même pour tous les types de fichiers. Ainsi, on peut se permettre, pour le son, les images, les animations, etc, de perdre un peu d'information pour autant que la réduction de taille lors de la compression soit très grande. Cela se traduit en général par une perte de qualité, dans le son, l'image ou l'animation. Une telle compression est dite non-conservative ou avec perte d'information (ou enfin "lossy" en anglais). Une méthode de compression non-conservative est dédiée à un type de séquences bien précis et ne peut aucunement être appliquée à d'autres types de séquences. Le seuil de tolérance de la perte de qualité est en effet intimement lié au type de données que contient la séquence.

Aucune des séquences manipulées au cours de ce travail ne peuvent être dégradées par la compression. Il s'agit principalement de séquences de bits codant pour des données d'un type inconnu ou des séquences nucléiques qui ne peuvent pas être sujettes à des pertes d'information. Dorénavant, toutes les méthodes de compression que nous mentionnerons sont conservatives.

La compression est utilisée dans deux contextes précis :

1. Elle est habituellement utilisée pour **réduire l'espace de stockage** des fichiers informatiques et pour **diminuer les temps de transmission** de l'information sur un réseau. Dans ce contexte, la compression constitue une économie substantielle dans la mesure où le budget consacré à l'achat de disques magnétiques, optiques, de bandes magnétiques ou au paiement des notes de téléphone peut être utilisé à autre chose.
2. Elle s'avère également très intéressante dans le contexte de l'**analyse de séquences** de symboles. La compression d'une séquence procède par détection de régularités dans la séquence et codage efficace des régularités détectées. La présence des régularités se traduit par une certaine redondance qui est éliminée ou du moins atténuée dans la version comprimée de la séquence. Par définition, une séquence *sans régularité* est une séquence aléatoire (au sens défini par Martin-Löf). De manière équivalente, elle est incompressible (au sens de la complexité de Kolmogorov) [LV93]. Les régularités trop insignifiantes à l'échelle de la taille de la séquence ne permettent pas de réduire sa longueur. Une compression effective d'une séquence exige donc une bonne compréhension de sa structure. Le taux de compression, c'est-à-dire la mesure de réduction de taille obtenue par la compression (voir section 2.1.2), donne une évaluation de la quantité des régularités détectées.

Expliquons pourquoi de gros efforts doivent être réalisés pour développer des méthodes de compression qui réduisent au mieux les tailles des séquences.

Dans le premier contexte d'utilisation, c'est évident puisqu'il y a de l'argent en jeu !

Dans le deuxième contexte, les efforts doivent aussi être réalisés dans ce sens. En effet, la présence d'un type de régularités qu'une méthode de compression exploite traduit que la séquence possède une propriété bien précise. Plus la méthode comprime la séquence, plus la propriété est pertinente pour la séquence. On essaie donc d'obtenir un taux de compression le plus élevé possible pour analyser au mieux l'adéquation d'une propriété pour la séquence.

La théorie de la complexité de Kolmogorov (voir section 2.4) établit qu'il existe une borne inférieure de la longueur de la plus courte version comprimée d'une séquence. Cette borne théorique n'est malheureusement pas calculable mais il est facile de montrer que toute séquence ne peut pas être comprimée.

Étant donné une longueur n, montrons qu'aucun algorithme de compression conservative n'est capable de comprimer toutes les séquences de n bits [Del95].

Il y a 2^n séquences à considérer. Pour parler de compression effective, il est nécessaire que l'algorithme, appliqué à une séquence de n bits, fournisse une séquence qui comporte $n-1$ bits au maximum. D'autre part, si deux séquences distinctes S_1 et S_2 de n bits se compriment en S_1' et S_2', alors S_1' et S_2' doivent être distinctes. Donc, l'algorithme devrait fournir 2^n séquences de longueur maximale $n-1$. Or il n'en existe que $2^n - 1$ (voir tableau 2.1).

Nombre de séquences	Longueur
1	0
2	1
4	2
...	...
2^{n-1}	$n-1$
$2^n - 1$	$< n$

TAB. 2.1 - *Nombre de séquences de longueur maximale* $n-1$

Au moins une des séquences de longueur n ne peut donc pas être comprimée.

2.1.2 Gain et taux de compression

Pour évaluer et comparer différents compresseurs, il faut définir précisément une mesure de performance. Nous définissons le *gain* et le *taux* de compression.

Soit C un compresseur, $S \in \mathcal{B}^*$ la séquence à comprimer et $S' \in \mathcal{B}^*$ la séquence comprimée : $S' = C(S)$.

Définition 2.1 *Le* gain de compression, *de C appliqué à S, est le nombre de bits économisés par la compression : gain* $= |S| - |S'|$.

Définition 2.2 *Le* taux de compression, *de C appliqué à S, évalue le pourcentage de réduction de la longueur de S : taux* $= \frac{gain}{|S|}$

Exemple 2.1 *Supposons que la séquence $S1$ contienne 50 bits. Si la séquence comprimée $S1' = C(S1)$ contient 40 bits, alors :*

- *Le gain de compression est de 10 bits.*
- *Le taux de compression est de $\frac{10}{50} = 20\%$.*

Le gain et le taux de compression peuvent être négatifs. Cela se produit lorsque le compresseur rallonge la séquence : les propriétés qu'il exploite ne sont pas adaptées à cette séquence.

Exemple 2.2 *Supposons que la séquence $S2$ contienne 50 bits. Si la séquence comprimée $S2' = C(S2)$ contient 66 bits, alors :*

- *Le gain de compression est de -16 bits.*
- *Le taux de compression est de $\frac{-16}{50} = -32\%$.*

Remarque 2.1 *D'après les définitions 2.1 et 2.2, le gain et le taux de compression ne peuvent être calculés que si les séquences S et S' sont toutes deux des séquences binaires : $S, S' \in \mathcal{B}^*$. En effet, il faut comparer des choses comparables ! Il est exclu de compter le nombre de symboles de S et de S' si les séquences ne sont pas formées sur le même alphabet. Puisqu'un compresseur tente de réduire au maximum la taille des séquences, il est amené à utiliser des codages très fins. L'entité informative de base pour de tels codages est le bit, les séquences comprimées sont donc, dans la majorité des cas, des séquences binaires.*

Lorsque la séquence initiale $S \in \mathcal{A}^*$ n'est pas une séquence binaire, le gain et le taux de compression peuvent être calculés en effectuant la conversion suivante. Tout symbole de l'alphabet $\mathcal{A}$ peut être traduit sur l'alphabet binaire en utilisant exactement $\lceil \log \#\mathcal{A} \rceil$ bits. La séquence S peut donc être traduite sur l'alphabet $\mathcal{B}$ en utilisant exactement $|S| \times \lceil \log \#\mathcal{A} \rceil$ bits. On calcule le gain de compression comme la différence entre la longueur de la séquence S, traduite sur $\mathcal{B}$, et la longueur de la séquence comprimée S' : $gain = |S| \times \lceil \log \#\mathcal{A} \rceil - |S'|$. De la même manière, le taux de compression est donné par la formule : $taux = \frac{gain}{|S| \times \lceil \log \#\mathcal{A} \rceil}$. À titre d'exemple, prenons l'alphabet des séquences d'ADN : $\mathcal{N} = \{A, C, G, T\}$ (voir partie III de ce travail). On a $\#\mathcal{N} = 4$. Chaque base d'une séquence d'ADN peut être traduite sur 2 bits en traduisant par exemple A par 00, C par 01, G par 10 et T par 11.

Exemple 2.3 *Soit $S = AACTGAGTACTC \in \mathcal{N}^*$ une séquence d'ADN et $S' \in \mathcal{B}^*$, tel que $|S'| = 15$, sa version comprimée par le compresseur C.*

- *Le gain de compression est de $|S| \times 2 - |S'| = 9$ bits.*
- *Le taux de compression est de $\frac{9}{|S| \times 2} = 37.5\%$.*

L'omission de cette conversion préalable conduit à de fâcheuses erreurs d'interprétations concernant les performances de la compression.

2.1.3 Schéma général d'une méthode de compression

Un compresseur réalise toujours son travail en deux phases :

1. La phase de **détection de régularités** : elle parcourt la séquence et localise les régularités qui seront exploitées pour réduire sa longueur. Par exemple, un compresseur qui exploite les facteurs répétés d'une séquence commence par localiser les facteurs répétés.

   La qualité de cette phase relève de son aptitude à détecter les régularités intéressantes en un temps raisonnable. Si les régularités, qui permettent de bien comprimer, sont trop difficiles à détecter, le compresseur est inutilisable. Souvent, il faut trouver un juste milieu pour obtenir une compression satisfaisante tout en conservant un temps de calcul raisonnable.

2. La phase de **codage** : elle utilise les informations collectées tout au long de la première phase pour construire la version comprimée de la séquence. Le but est de supprimer les redondances produites par les régularités. Dans le cas du compresseur qui exploite les facteurs répétés, la première occurrence d'un facteur est copiée telle quelle dans la séquence de sortie, les autres occurrences sont remplacées par un pointeur référençant la première. Tout facteur non répété est recopié tel quel.

   Cette phase s'exécute bien souvent très rapidement car il lui suffit de parcourir la séquence initiale, la liste des régularités détectées et de produire le codage de la séquence comprimée.

   Dans la suite, l'ensemble de règles qui déterminent les suites de bits à écrire dans la séquence comprimée lorsqu'une information précise doit être codée, constitue ce que l'on appelle le *schéma de codage*. Ces règles doivent vérifier des contraintes spécifiques décrites à la section 2.2.

Dans beaucoup d'algorithmes de compression, ces deux phases sont entremêlées. Il s'agit principalement d'algorithmes en ligne qui compriment la séquence au fur et à mesure de sa lecture. Régulièrement, la phase de détection passe la main à la phase de codage pour écrire une partie de la séquence comprimée en sortie. Après avoir écrit ces bits, la phase de codage repasse la main à la phase de détection qui continue son travail sur la suite de la séquence initiale.

Puisque le compresseur C tente de supprimer les redondances provoquées par la présence d'un type précis de régularités, si $S \in \mathcal{B}^*$ est une séquence et $S' = C(S)$ sa version comprimée, alors S' n'est plus, ou est très peu, compressible par C. En effet, supposons que $S'' = C(S')$ avec $|S''| < |S'|$. Cela signifie que le compresseur C a encore détecté, dans S', des régularités du même type qui provoquent des redondances permettant de réduire sa longueur. Lors de la compression de S, le compresseur C n'a donc pas supprimé toutes les redondances qu'il aurait pu supprimer. Le compresseur peut encore être amélioré.

2.2 Techniques de codage

La conception d'un schéma de codage doit être réalisée soigneusement, elle doit respecter les deux contraintes majeures suivantes.

1. La séquence comprimée doit être **déchiffrable de manière non ambiguë**. On travaille en effet dans un contexte de compression conservative.

2. Le schéma de codage doit être **performant** : le nombre de bits de la séquence comprimée doit être aussi petit que possible.

Beaucoup d'ouvrages s'étendent largement sur les méthodes de codage de l'information [Sha48, SW49, Eli75, ER78, BP85, AF87, LH87, Sto88, BCW90, Zip90, Bru91, HM91, LV93, Nel93, CR94, CL97]. Il s'agit d'un large domaine d'étude en informatique théorique et appliquée. Nous nous restreignons ici aux définitions et aux propriétés de base pertinentes dans le cadre de la compression de séquences. Nous nous étendons un peu plus sur les codages auto-délimités de nombres entiers, qui sont très importants lorsque les nombres à coder sont arbitrairement grands [BCW90]. Un nouveau concept de codage d'entiers, appelé *code ICL*, est défini. Il confère une continuité naturelle aux longueurs des codages des entiers. Il sera intensément utilisé dans les parties II d'optimisation de courbes et III d'applications à la compression.

2.2.1 Codes

La définition d'un code que nous donnons ici est celle qui est habituellement utilisée en compression. Un code est une fonction injective qui applique un mot source sur un mot code [Sto88]. Remarquons que cette définition généralise celle utilisée en théorie des codes algébriques où un code est un ensemble de mots tel que tout mot obtenu par concaténation des mots du code est déchiffrable de manière unique [BP85, Bru91].

Définition 2.3 *Étant donné un* ensemble source E_S *fini ou infini et un* alphabet de sortie $\mathcal{A}$, *un* code c *de* E_S *dans* $\mathcal{A}^+$ *est une fonction injective qui applique chaque élément de* E_S *sur un mot de* $\mathcal{A}^+$.

Les éléments de E_S *sont appelés des* mots source. *L'ensemble* $E_C = \{c(a) : a \in E_S\} \subseteq \mathcal{A}^+$ *est appelé l'*ensemble code, *ses éléments sont les* mots code.

Définition 2.4 *Puisque* c *est injectif, il existe une fonction réciproque* $c^{-1} : E_C \to E_S$ *que l'on appelle la* fonction de décodage. *Elle est définie par* $\forall w \in E_C : c^{-1}(w) = a \iff c(a) = w$.

L'ensemble source est un ensemble quelconque, chaque mot source représente une entité que l'on désire coder. Ici, "mot source" est à comprendre comme une entité informative de base : un mot source n'est pas décomposable en entités plus petites. L'ensemble source E_S peut, par exemple, être l'ensemble $\mathbb{N}$ des entiers naturels, l'alphabet des nucléotides $\mathcal{N}$, l'alphabet binaire $\mathcal{B}$ ou encore l'alphabet $\mathcal{B}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ contenant les mots binaires de longueur 3.

On étend la définition d'un code à l'homomorphisme d'un code.

Définition 2.5 *Soit* $c : E_S \to \mathcal{A}^+$ *un code. Soit* $c^* : E_S^* \to \mathcal{A}^*$ *l'*homomorphisme *de* c *défini par* $c^*(a_1 a_2 \dots a_p) = c(a_1)c(a_2)\dots c(a_p), \forall i, a_i \in E_S$. *L'homomorphisme* c^* *est un code si et seulement si un mot* $w = c(a_1)c(a_2)\dots c(a_p)$ *est déchiffrable de manière unique c'est-à-dire qu'il existe au plus une suite* $(a_1, a_2, \dots, a_p)$ *d'éléments de* E_S *tels que* $w = c(a_1)c(a_2)\dots c(a_p)$.

On dit alors que c^* *est le* code homomorphe *de* c.

Cette définition du code homomorphe c^* rejoint la définition d'un code algébrique [BP85, Bru91].

Exemple 2.4 *Soit* $E_S = \{a, b, c, d, e\}$. *L'alphabet de sortie est l'alphabet binaire* $\mathcal{B}$. *Le code* $f : E_S \to \mathcal{B}^+$ *est défini par:*

$$\begin{array}{ll} f(a) = 00 & f(b) = 110 \\ f(c) = 10 & f(d) = 11 \\ f(e) = 100 & \end{array}$$

L'ensemble $E_C \subset \mathcal{B}^+$ *est* $\{00, 110, 10, 11, 100\}$.

L'homomorphisme f^* *est un code (la preuve est omise). Soit* $w = 0011010 \in E_C^+$, *il est déchiffrable de manière unique:* $w = f(a)f(b)f(c)$.

Le fait que $c : E_S \to \mathcal{A}$ soit un code n'implique pas systématiquement que l'homomorphisme c^* soit un code. L'exemple 2.5 donne un exemple de code qui s'étend en un homomorphisme qui n'en est pas un.

Exemple 2.5 *Soit* $E_S = \{a, b, c\}$. *L'alphabet de sortie est l'alphabet binaire* $\mathcal{B}$. *Le code* $f : E_S \to \mathcal{B}^+$ *est défini par:*

$$f(a) = 0 \quad f(b) = 1 \quad f(c) = 01$$

L'homomorphisme f^* *n'est pas un code car* $f(a)f(b) = f(c)$.

2.2.2 Codes auto-délimités (préfixes)

Un code *auto-délimité* (parfois appelé *code préfixe* ou *code instantané*) s'étend en un code homomorphe.

Définition 2.6 *Un code* $c : E_S \to \mathcal{A}^+$ *est* auto-délimité *si aucun mot code n'est le préfixe d'un autre mot code. C'est-à-dire*

$$\forall a, a' \in E_S : c(a) \text{ préfixe de } c(a') \Rightarrow a = a'$$

La proposition 2.1 illustre l'intérêt d'un tel code : il s'étend en un homomorphisme qui est un code.

Proposition 2.1 *Soit* $c : E_S \to \mathcal{A}^+$ *un code. Si* c *est auto-délimité, alors l'homomorphisme* $c^* : E_S^* \to \mathcal{A}^*$ *est un code.*

Preuve. Soient $a_1, a_2, \ldots a_p, b_1, b_2, \ldots b_q \in E_S$ tels que

$$c^*(a_1 a_2 \ldots a_p) = c^*(b_1 b_2 \ldots b_q)$$

Alors $c(a_1)c(a_2)\ldots c(a_p) = c(b_1)c(b_2)\ldots c(b_q)$. Supposons $|c(a_1)| \leq |c(b_1)|$ (le raisonnement est identique si $|c(b_1)| \leq |c(a_1)|$). Donc $c(a_1)$ est préfixe de $c(b_1)$. Puisque c est auto-délimité, $a_1 = b_1$. En utilisant le même argument pour $c(a_2), c(b_2), \ldots$, on obtient $p = q$ et $a_i = b_i, \forall i \leq p$. Donc c^* est un code puisque le décodage est non ambigu. □

Un code auto-délimité est également appelé un code instantané car le décodage d'une séquence de mots code se fait instantanément. Il suffit de commencer le décodage au début de la séquence et de décoder un mot code à la fois. Quand on arrive à la fin d'un mot code, on sait qu'il est terminé puisqu'aucun mot code n'est préfixe d'un autre mot code.

Le code de l'exemple 2.4 n'est pas auto-délimité car $f(d)$ est préfixe de $f(b)$ et $f(c)$ est préfixe de $f(e)$. C'est même un *code à délai de déchiffrage infini* [BP85, p. 128] car

si l'on prend la séquence infinie périodique de mots code $f(c)f(a)f(a)f(a)f(a)\ldots$ qui est $100000000\ldots$, elle peut également être décodée en $f(e)f(a)f(a)f(a)f(a)\ldots$. C'est un code malgré tout car toute séquence finie de mots code est déchiffrable de manière non ambiguë.

Exemple 2.6 *Soit $E_S = \{0,1,2,3\}$ l'ensemble source et $f : E_S \to \mathcal{B}^+$ le code défini par :*

$$f(0) = 0 \qquad f(1) = 10$$
$$f(2) = 110 \qquad f(3) = 111$$

La séquence 0310 *est codée en* 0111100 *qui peut être facilement décodé de manière non ambiguë de la gauche vers la droite. C'est un code auto-délimité.*

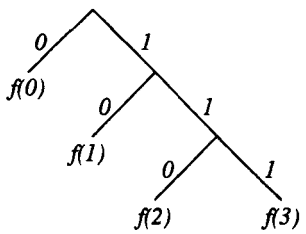


FIG. 2.1 - *Trie de E_S*

On représente habituellement un code auto-délimité par le trie (voir définition, section 1.3.2) de son ensemble code. Puisque cet ensemble ne contient aucun mot préfixe d'un autre mot, tous les mots code sont situés sur les feuilles. La figure 2.1 représente le code f. La reconnaissance d'un mot code lors du décodage d'une séquence se fait en suivant le chemin, partant de la racine, étiqueté par les symboles lus. Lorsque le chemin atteint une feuille, un mot code a été complètement lu.

Les codes auto-délimités sont très intéressants dans le domaine de la compression de séquences d'une part parce qu'ils assurent un décodage non ambigu, d'autre part ils assurent que ce décodage est linéaire en temps : un seul parcours de la séquence codée suffit.

L'exemple suivant permet de coder n'importe quelle séquence binaire de manière auto-délimitée.

Exemple 2.7 *Soit $c_1 : E_S = \mathcal{B}^* \to \mathcal{B}^+$ le code défini par $c_1(x) = 1^{|x|}0x$. Ainsi, $c_1(0110) =$* 1111 0 0110.

Par construction, c_1 est un code car x est retrouvé sans ambiguïté lorsque $c_1(x)$ est connu. Il est évident que $\forall x \in \mathcal{B}^*, |c_1(x)| = 2|x| + 1$.

C'est un code auto-délimité. En effet, supposons que $c_1(x)$ soit préfixe de $c_1(x')$, c'est-à-dire que $1^{|x|}0x$ préfixe de $1^{|x'|}0x'$.

1^{er} *cas : $|x| = |x'|$.*

Dans ce cas, $|c_1(x)| = |c_1(x')|$ et donc $c_1(x) = c_1(x')$.

$2^{ème}$ *cas : $|x| < |x'|$.*

Dès lors, $1^{|x|}0$ est préfixe de $1^{|x'|}0$, ce qui est impossible.

$3^{ème}$ *cas : $|x| > |x'|$.*

Donc, $|c_1(x)| > |c_1(x')|$, ce qui n'est pas possible si $c_1(x)$ est préfixe de $c_1(x')$.

2.2.3 Théorie de l'information

La théorie de l'information a été mise sur pied à la fin des années quarante par Shannon [Sha48, SW49]. Elle s'intéresse au problème de communication d'un message entre un émetteur et un récepteur, elle ne considère pas l'aspect "signification" du message comme dans la *théorie algorithmique de l'information* (voir section 2.4). Nous exposons ici les concepts de base de cette théorie tels que l'entropie, l'universalité et l'optimalité d'un code. Le lecteur intéressé trouvera la théorie exposée plus en profondeur dans les ouvrages [LH87, LV93].

2.2.3.1 Entropie et longueur moyenne

Supposons que l'on ait un ensemble E_S de mots source à coder sur l'alphabet binaire $\mathcal{B}$ (toute cette théorie est facilement adaptable à un autre alphabet de sortie). La théorie de l'information suppose que les paramètres statistiques de l'ensemble E_S sont connus. Notons p la distribution de probabilité d'apparition des mots source dans les messages à coder: $\forall a_i \in E_S : p(a_i)$ est la probabilité d'apparition du mot a_i dans un message. Bien entendu, $\sum_{a_i \in E_S} p(a_i) = 1$.

La mesure de la quantité d'information, apportée par le mot source a_i, est $h_i = -\log p(a_i)$. L'explication intuitive de cette mesure est que si $p(a_i) = 1$, alors l'apparition de a_i dans un message n'apporte aucune information puisqu'on savait qu'il allait apparaître. Plus petite est la valeur de $p(a_i)$, moins il est probable que a_i apparaisse et donc plus son contenu en information est grand. Une discussion technique sur cette façon de présenter les choses se trouve dans [Abr63, pp.6–13]. Le contenu moyen en information de l'ensemble source E_S est obtenu en additionnant les contenus en informations des mots a_i, pondérés par leur probabilité d'apparition. Ce contenu moyen est appelé *entropie*, et représente le nombre minimum de bits nécessaires, en moyenne, pour coder les mots source.

Définition 2.7 *Soit E_S un ensemble source et p la distribution de probabilité des mots de E_S. L'*entropie *de E_S relativement à p est:*

$$H(p) = \sum_{a_i \in E_S} p(a_i) h_i = -\sum_{a_i \in E_S} p(a_i) \log p(a_i)$$

Si l'on doit concevoir un code $c : E_S \to \mathcal{B}^+$, l'entropie impose une borne inférieure aux longueurs $|c(a_i)|$ des mots code. Définissons la *longueur moyenne des mots code.*

Définition 2.8 *Soit $c : E_S \to \mathcal{B}^+$ un code et p la distribution de probabilité des mots source. La* longueur moyenne des mots code, *notée $E(c,p)$ est:*

$$E(c,p) = \sum_{a_i \in E_S} p(a_i)|c(a_i)|$$

Puisque la longueur $|c(a_i)|$ doit être suffisante pour transporter le contenu en information de a_i, il n'est pas possible d'avoir une longueur moyenne des mots code inférieure à l'entropie. C'est ce qu'exprime la proposition 2.2.

Proposition 2.2 *Soit E_S un ensemble source et p la distribution de probabilité des mots source. Alors pour tout code $c : E_S \to \mathcal{B}^+$ on a $H(p) \leq E(c,p)$.*

La preuve est omise. Elle se trouve dans [LV93, p. 71].

2.2.3.2 Code optimal, code universel et code asymptotiquement optimal

L'*optimalité* et l'*universalité* des codes sont des propriétés qui concernent les performances. Un code optimal possède une longueur moyenne des mots code qui est minimale.

Définition 2.9 *Soit E_S un ensemble source et p la distribution de probabilité des mots code. Soit $L = min\{E(f,p)$ tel que $f : E_S \to \mathcal{B}^+$ est un code$\}$ le minimum de la longueur moyenne des mots code pour E_S et p. Un code $c : E_S \to \mathcal{B}^+$ est* optimal *si $L = E(c,p)$.*

Exemple 2.8 *Considérons à nouveau le code auto-délimité f de l'exemple 2.6. Supposons que la distribution de probabilité p des mots source soit :*

$$p(0) = \frac{1}{2}, p(1) = \frac{1}{4}, p(2) = \frac{1}{8} \text{ et } p(3) = \frac{1}{8}$$

Alors l'entropie est $H(p) = \sum_{i=0}^{3} -p(i) \log p(i) = 1.75$. La longueur moyenne des mots code de f est $E(f,p) = \sum_{i=0}^{3} p(i)|f(i)| = 1.75$. Dans ce cas, l'entropie est atteinte. Le code f est optimal car un autre code ne peut jamais avoir une longueur moyenne des mots code inférieure à l'entropie.

Exemple 2.9 *Reprenons l'ensemble source E_S et la distribution de probabilité de l'exemple précédent. Le code $g : \{0,1,2,3\} \to \mathcal{B}^+$ défini par :*

$$g(0) = 00 \qquad g(1) = 01$$
$$g(2) = 10 \qquad g(3) = 11$$

n'est pas optimal car $E(g,p) = \sum_{i=0}^{3} p(i)|g(i)| = 2 > 1.75 = E(f,p)$.

Shannon a démontré que la longueur moyenne des mots code d'un code optimal est à une distance maximale de 1 bit [Sha48, SW49] de l'entropie. Cette propriété, conjuguée avec la proposition 2.2, est connue sous le nom de "Noiseless Coding Theorem".

Théorème 2.3 (Noiseless Coding Theorem) *Soit E_S un ensemble de mots source distribués avec la probabilité p. Soit $c : E_S \to \mathcal{B}^+$ un code optimal. Alors $H(p) \leq E(c,p) \leq H(p)+1$.*

La preuve est omise. Elle se trouve dans [LV93, p. 71].

Un code *universel* est un code qui optimise, à une constante multiplicative près, la longueur moyenne des mots code indépendamment de la distribution de probabilité des mots de l'ensemble source.

Définition 2.10 ([Eli75]) *Le code $c : E_S \to \mathcal{B}^+$ est* universel *s'il existe deux constantes K_1, K_2 telles que $E(c,p) \leq K_1 H(p) + K_2$ pour toute distribution de probabilité p.*

Il n'est donc pas nécessaire de connaître la distribution de probabilité exacte des mots source pour concevoir un code universel. L'importance de la compression, produite par l'utilisation d'un code universel, dépend bien entendu de la grandeur des constantes K_1 et K_2.

Un code *asymptotiquement optimal* est un code universel dont la longueur moyenne des mots code approche l'entropie lorsque celle-ci devient grande.

Définition 2.11 *Le code universel $c : E_S \to \mathcal{B}^+$ est* asymptotiquement optimal *si*

$$\lim_{H(p) \to +\infty} \frac{E(c,p)}{H(p)} = 1$$

D'après les définitions 2.10 et 2.11, un code universel avec la constante $K_1 = 1$ est un code asymptotiquement optimal.

Exemple 2.10 ([LV93, p. 74]) *On montre que le code $c_1 : \mathcal{B}^* \to \mathcal{B}^+$ de l'exemple 2.7 est universel mais pas asymptotiquement optimal.*

L'exemple 2.11 définit un code, permettant de coder toute séquence binaire de manière auto-délimitée, qui est asymptotiquement optimal. Pour le définir, nous avons besoin des deux fonctions INT et BIN permettant de convertir une séquence binaire en entier et inversement.

Définition 2.12 *Soit $INT : \mathcal{B}^* \to \mathbb{N}$ la fonction permettant de convertir une séquence binaire en entier selon la conversion binaire usuelle. Par exemple, $INT(0) = 0, INT(1) = 1, INT(10) = 2, ..., INT(010011) = 19$,... Ce n'est pas un code. Par exemple, $INT(10011) = 19 = INT(010011) = INT(000010011)$.*

Définition 2.13 *Soit $BIN : \mathbb{N} \to \mathcal{B}^+$ la fonction permettant de convertir un entier en séquence binaire de la façon suivante :*

- *Si $i > 0$, alors $BIN(i) = w \in \mathcal{B}^+$ tel que $INT(w) = i$ et le premier chiffre de w est 1.*
- *$BIN(0) = 0$.*

C'est un code, sa fonction réciproque est INT.

Pour $i \in \mathbb{N}$, nous notons $l(i)$ la longueur de sa représentation binaire : $l(i) = |BIN(i)| = \lceil \log(i+1) \rceil$ pour $i > 0$ et $l(0) = 1$.

Exemple 2.11 ([LV93, p. 74]) *Soit $c_2 : \mathcal{B}^* \to \mathcal{B}^+$ le code qui applique x sur $c_2(x) = c_1(BIN(|x|))\, x$. Prenons $x = 0110 \Rightarrow |x| = 4 \Rightarrow BIN(|x|) = 100 \Rightarrow c_2(x) = 111\,0\,100\,0110$.*

Le code c_2 est auto-délimité (la preuve est similaire à la preuve de l'auto-délimitation de c_1 dans l'exemple 2.7). La longueur d'un mot code est $|c_2(x)| = 2l(|x|) + |x| + 1$.

Il est possible de montrer que c_2 est asymptotiquement optimal. Intuitivement, c'est parce que $2l(|x|)$ devient négligeable par rapport à $|x|$ lorsque $|x|$ devient grand.

2.2.4 Codage de nombres entiers

Lors de la phase de codage d'une méthode de compression, il est souvent nécessaire de coder des entiers. C'est le cas par exemple dans les algorithmes classiques de Ziv et Lempel (section 2.3.2) lorsque les indices dans le dictionnaire doivent être codés. Pour assurer un décodage unique, les nombres doivent être codés de manière auto-délimitée. Remarquons que BIN, défini à la section précédente, n'est pas auto-délimité car, par exemple, le mot 10111 est soit $BIN(5)BIN(3)$, soit $BIN(2)BIN(7)$, soit $BIN(2)BIN(1)BIN(3)$,... Il convient donc de définir d'autres codes de $\mathbb{N}$ dans $\mathcal{B}^+$.

Le codage d'entiers de manière auto-délimitée est également très important dans la mesure où il permet d'auto-délimiter n'importe quelle séquence binaire. Il suffit de la faire précéder du codage auto-délimité de sa longueur.

Lorsque l'éventail de nombres à coder est petit et que les nombres ont des probabilités semblables, alors on peut utiliser un code qui donne une longueur fixe à tous les mots code. Si, par contre l'éventail de nombres est grand ou si les probabilités des nombres sont très dissemblables, il convient d'utiliser des codes à longueur variable.

Nous définissons le nouveau concept de code à longueur variable *ICL*. C'est un code permettant de coder des nombres entiers de telle sorte que l'accroissement des longueurs des mots code vérifie certaines contraintes. Ces codes permettront de créer des fonctions de rupture *DCL* (voir proposition 2.1, de la partie II) pour l'optimisation de courbes de compression (voir partie III).

Nous montrons que le code *Fibo*, introduit par Apostolico et Fraenkel [AF87], est ICL. Nous définissons le nouveau code *PrefFibo* qui est non seulement ICL mais également asymptotiquement optimal.

2.2.4.1 Codes à longueur fixe

Soit $E_S = [0, n] \subset \mathbb{N}$ l'ensemble source ($n > 0$). Supposons que la répartition des mots source soit uniforme: $\forall i \in E_S : p(i) = \frac{1}{n+1}$.

Alors $H(p) = -\sum_{i=0}^{n} p(i) \log p(i) = -\sum_{i=0}^{n} \frac{1}{n+1} \log \frac{1}{n+1} = -\log \frac{1}{n+1} = \log(n+1)$.

Soit $k = \lceil \log(n+1) \rceil$. Il est possible de coder tous les nombres de E_S sur exactement k bits en utilisant le codage fix_k limité à l'ensemble source E_S.

Définition 2.14 *Étant donné $k \geq 1$, $fix_k : [0, 2^k - 1] \to \mathcal{B}^k$ est le code préfixe permettant de coder tous les nombres de $[0, 2^k - 1]$ sur k bits en utilisant la conversion binaire usuelle: $\forall i \in [0, 2^k - 1] : INT(fix_k(i)) = i$ et $|fix_k(i)| = k$. Par exemple, $fix_4(0) = 0000, fix_4(5) = 0101, \ldots$*

La figure 2.2 représente le trie des mots code de $f : [0, 10] \to \mathcal{B}^+ : x \mapsto f(x) = fix_4(x)$.

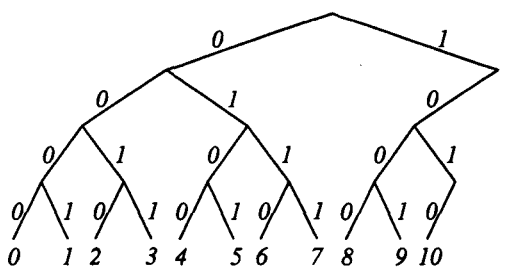


FIG. 2.2 - *Code $f : [0, 10] \to \mathcal{B}^+ : x \mapsto f(x) = fix_4(x)$*

La longueur moyenne des mots code de $f : [0, n] \to \mathcal{B}^+ : x \mapsto f(x) = fix_k(x)$, est $E(f, p) = \sum_{i=0}^{n} p(i)|fix_k(i)| = \sum_{i=0}^{n} \frac{1}{n+1} \lceil \log(n+1) \rceil = \lceil \log(n+1) \rceil$.

La condition $H(p) \leq E(f, p) \leq H(p) + 1$ du théorème 2.3 est satisfaite. La longueur moyenne est donc très proche de l'entropie. Cela ne signifie pas que f soit optimal! En effet, quelle que soit la distribution de probabilité p, le code préfixe f' correspondant au trie de la figure 2.3 donnera une longueur moyenne des mots code inférieure à $E(f, p)$.

Toutefois, si $n + 1$ est une puissance de 2 alors $E(f, p) = H(p)$ et donc le code f est optimal.

Tant que les probabilités $p(i)$ restent semblables, le code fix_k est performant.

2.2.4.2 Codes à longueur variable

Dans un premier temps, supposons que l'ensemble source E_S soit fini. Soit $E_S = [0, n] \subset \mathbb{N}$ avec ($n > 0$). Supposons que la répartition des mots source soit très éloignée d'une répartition

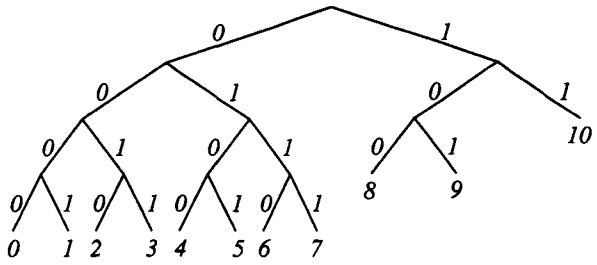


FIG. 2.3 - *Code* $f' : [0, 10] \to \mathcal{B}^+$

uniforme. Dans ce cas, un code à longueur fixe n'est pas adapté : la longueur moyenne des mots serait trop éloignée de l'entropie. C'est le cas de l'exemple 2.9.

On utilise donc un code à longueur variable. Étant donné la distribution de probabilités des mots source, la méthode de Huffman (section 2.3.1.2.2) permet de construire un tel code. Le code décrit dans l'exemple 2.8 était optimal pour sa distribution de probabilités.

Plaçons-nous maintenant dans le cas d'ensembles sources infinis.

Parfois, la valeur maximale des entiers à coder n'est pas connue : $E_S = \mathbb{N}$. Il n'est alors pas possible d'utiliser un code à longueur fixe ou un code de Huffman. De plus, il est souvent préférable d'avoir des mots code courts pour de petits entiers et des mots code plus longs pour de plus grands entiers. Nous présentons ici des codes auto-délimités permettant de coder des entiers arbitrairement grands.

Exemple 2.12 *Soit* $f_0 : \mathbb{N} \to \mathcal{B}^+$ *le code défini par* $f_0(x) = 1^x0$. *Puisque le symbole* 0 *ne peut apparaître qu'à la fin d'un mot code, le code est auto-délimité. La longueur du mot code* $f_0(x)$ *est* $|f_0(x)| = x + 1$. *Ce code n'est pas universel.*

2.2.4.2.1 Limite des longueurs des mots code

Si tous les entiers doivent être codés de manière auto-délimitée, la proposition 2.4 établit une limite inférieure théorique à la longueur des mots code.

Proposition 2.4 ([LV93, p. 75]) *Soit* $c : \mathbb{N} \to \mathcal{B}^+$ *un code auto-délimité. Alors, lorsque* $x \to +\infty : |c(x)| > \ell^*(x)$, *avec :*

$$\ell^*(x) = \begin{cases} l'(x) + \ell^*(l'(x)) & \text{si } l'(x) > 1 \\ l'(x) & \text{si } l'(x) \leq 1 \end{cases} \quad \text{et } l'(x) = l(x+1) - 1$$

La preuve est omise. Elle se trouve dans [LV93, p. 75]. Elle est basée sur l'inégalité de Kraft [Kra49].

Remarque 2.2 *Dans cette proposition,* $l'(x)$ *est utilisé à la place de* $l(x)$ *pour se placer exactement dans le même contexte que celui de [LV93]. Dans ce livre, les auteurs utilisent un autre code que BIN pour les entiers, ce qui justifie la différence de longueur des mots code :* $l'(x) = l(x+1) - 1$.

En pratique, il est possible de coder x de manière auto-délimitée en utilisant $\theta(\log x)$ bits comme le montre l'exemple suivant.

Exemple 2.13 *Soit $f_1 : \mathbb{N} \to \mathcal{B}^+$ le code défini par $f_1(x) = c_1(BIN(x))$. C'est un code auto-délimité, la longueur du mot code $f_1(x)$ est $|f_1(x)| = 2l(x) + 1$. Puisque $l(x) = \lceil \log(x+1) \rceil$, $|f_1(x)| \in \theta(\log x)$.*

Ce code sera souvent utilisé par la suite, nous noterons également $f_1(x) = \bar{x}$.

Remarque 2.3 *Intuitivement, lorsqu'un code tel que f_1 permet d'auto-délimiter tous les entiers x en $\theta(\log x)$ bits, il est universel.*

On peut également préfixer le code $BIN(x)$ par sa longueur auto-délimitée à l'aide du code f_1. On obtient le code f_2.

Exemple 2.14 *Soit $f_2 : \mathbb{N} \to \mathcal{B}^+$ le code défini par $f_2(x) = \overline{l(x)}BIN(x)$. C'est un code auto-délimité, la longueur du mot code $f_2(x)$ est $|f_2(x)| = 2l(l(x)) + l(x) + 1 \in \theta(\log x)$.*

On peut répéter le préfixage par la longueur comme le montre l'exemple 2.15.

Exemple 2.15 *Soit $f_3 : \mathbb{N} \to \mathcal{B}^+$ le code défini par $f_3(x) = \overline{l(l(x))}BIN(l(x))BIN(x)$. C'est un code auto-délimité, la longueur du mot code $f_3(x)$ est $|f_3(x)| = 2l(l(l(x))) + l(l(x)) + l(x) + 1 \in \theta(\log x)$.*

Remarque 2.4 *Intuitivement, un code asymptotiquement optimal $f : \mathbb{N} \to \mathcal{B}^+$ possède des longueurs de mots code $|f(x)|$ qui ne sont pas beaucoup plus grandes que $l(x)$. Ainsi, on démontre que si on code un nombre x par un mot de longueur de l'ordre de $l(x) + 2l(l(x))$ (cas de l'exemple 2.14), alors il est asymptotiquement optimal. Par contre, si la longueur du mot code $f(x)$ est de l'ordre de $2l(x)$, alors il n'est pas asymptotiquement optimal. Dans la suite, pour démontrer qu'un code est asymptotiquement optimal, nous nous ramènerons à des codes asymptotiquement optimaux connus, par la propriété évidente que si f' est asymptotiquement optimal et si, pour tout x assez grand: $|f(x)| \leq |f'(x)|$ alors f est aussi asymptotiquement optimal.*

2.2.4.2.2 Codes de Fibonacci

Les codes à longueur variable permettant d'auto-délimiter tous les entiers ont été abondamment étudiés [Eli75, ER78, AF87, Sto88, BCW90, LV93]. Ils sont bien souvent obtenus par combinaison de plusieurs méthodes et de préfixages de mots code par leurs longueurs. Nous nous intéressons maintenant à des codes un peu plus originaux, appelés *codes de Fibonacci* qui utilisent la représentation des nombres entiers en base de Fibonacci. Ces codes ont été introduits par Apostolico et Fraenkel [AF87]. Ils ont l'avantage supplémentaire d'être robustes aux erreurs: les conséquences néfastes, provoquées par une erreur, seront bien souvent limitées au décodage erroné d'un seul mot code. Nous ne mentionnons ici que le code de Fibonacci d'ordre 2, il utilise le fait qu'un nombre de Fibonacci est égal à la somme des 2 nombres de Fibonacci précédents. Toutes les propriétés restent valables pour les ordres supérieurs lorsque l'on définit un nombre de Fibonacci d'un ordre supérieur comme l'addition de plus des deux nombres précédents comme cela est fait dans [AF87].

Rappelons que les nombres de Fibonacci sont les nombres F_n, avec $n \geq 0$, tels que:

$$F_0 = 0, F_1 = 1 \text{ et } F_{i+2} = F_i + F_{i+1} \text{ pour } i \geq 0$$

Tout entier $x > 0$ a exactement une représentation de la forme $R(x) = \sum_{i=1}^{k} d_i F_{i+1}$ (où $d_i \in \mathcal{B}$ et $F_{i+1} \leq x$) telle que deux coefficients adjacents d_i, d_{i+1} ne sont pas tous les deux égaux à 1 et $d_k = 1$ [Knu73a, LH87].

Définition 2.15 *Soit Fibo* : $\mathbb{N} \to \mathcal{B}^+$ *le* code de Fibonacci *qui applique l'entier* x *sur le mot code* $d_1 d_2 \ldots d_k 1$ *où* $d_1, d_2, \ldots, d_k$ *sont les coefficients de la représentation de Fibonacci* $R(x+1)$.

Le code utilise la représentation de $R(x+1)$ plutôt que celle de $R(x)$ pour autoriser le codage de 0.

Le tableau 2.2 donne quelques exemples de mots code de Fibonacci.

x	$R(x+1)$							$Fibo(x)$
	21	13	8	5	3	2	1	
0	0	0	0	0	0	0	1	11
1	0	0	0	0	0	1	0	011
2	0	0	0	0	1	0	0	0011
4	0	0	0	1	0	0	0	00011
6	0	0	0	1	0	1	0	01011
8	0	0	1	0	0	0	1	100011
10	0	0	1	0	1	0	0	001011
11	0	0	1	0	1	0	1	101011
15	0	1	0	0	1	0	0	0010011
25	1	0	0	1	0	0	0	00010011

TAB. 2.2 - *Exemples de mots code de Fibonacci*

La proposition 2.5 montre que le code de Fibonacci est auto-délimité.

Proposition 2.5 *Le code Fibo est auto-délimité.*

Preuve. Tout mot code se termine par la séquence 11 qui n'apparaît nulle part ailleurs dans le mot. Il n'est donc pas possible d'avoir $x, y \in \mathbb{N}$ tels que $Fibo(x)$ soit préfixe de $Fibo(y)$ et $x \neq y$. □

Dans [AF87] les auteurs montrent que *Fibo* est universel. Intuitivement on le vérifie en montrant que $|Fibo(x)| \in \theta(\log x)$.

Proposition 2.6 $|Fibo(x)| \in \theta(\log x)$

Preuve. Soit $n = |Fibo(x)|$. Donc F_n est le plus grand nombre de Fibonacci inférieur ou égal à $x+1$. Or $F_n = \frac{\phi^n}{\sqrt{5}}$ arrondi à l'entier le plus proche [Knu73a] avec $\phi = \frac{1+\sqrt{5}}{2}$ le "nombre d'or". On conclut alors que $|Fibo(x)| \in \theta(\log x)$ □

2.2.4.2.3 Codes ICL

Un *code ICL* est un code, de $\mathbb{N}$ dans $\mathcal{B}^+$, qui possède des propriétés bien précises concernant l'accroissement de la longueur des mots code. C'est un nouveau concept dont l'utilité n'apparaît pas immédiatement mais il sera intensément utilisé dans les parties II et III pour la création de fonctions de ruptures DCL.

Les codes ICL ont des propriétés naturelles dans la mesure où elles expriment que la fonction de longueur des mots code ressemble à une fonction logarithmique discrétisée (voir figure 2.4).

Définition 2.16 *Un code $f : \mathbb{N} \to \mathcal{B}^+$ est un* code ICL *(croissant, concave et limité: "Increasing, Concave and Limited" en anglais) si la fonction de longueur $l_f : \mathbb{N} \to \mathbb{N} : x \mapsto l_f(x) = |f(x)|$ possède les trois propriétés suivantes:*

1. *l_f est une fonction croissante, c'est-à-dire que $\forall x, y \in \mathbb{N}$ avec $x < y$, $l_f(x) \leq l_f(y)$. C'est une propriété naturelle. Elle établit que les mots codes sont plus longs pour les plus grands entiers.*

2. *l_f est une fonction concave, c'est-à-dire que pour deux longueurs l_1, l_2 avec $l_1 < l_2$, il existe au moins autant de mots code de longueur l_2 que de mots code de longueur l_1 :*

$$\#\{x \in \mathbb{N} : l_f(x) = l_1\} \leq \#\{x \in \mathbb{N} : l_f(x) = l_2\}$$

3. *L'accroissement de l_f est limité à 1 entre deux entiers consécutifs. C'est-à-dire que $\forall x \in \mathbb{N}, l_f(x+1) \leq l_f(x) + 1$. Le mot code correspondant à $x+1$ est au maximum un bit plus long que celui correspondant à x.*

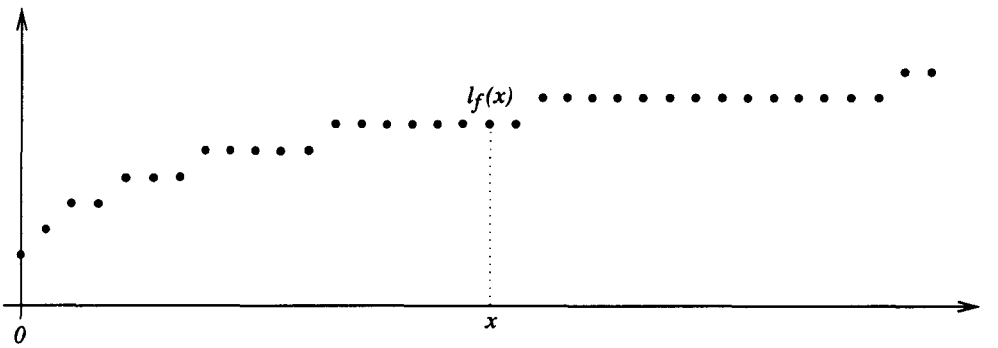


FIG. 2.4 - *Fonction de longueur l_f du code ICL f*

Les "marches" du graphique de la fonction l_f ont toutes une hauteur de 1 et elles sont de plus en plus longues lorsque les entiers deviennent grands (voir figure 2.4).

Exemple 2.16 *Le code BIN est ICL car:*

1. *La longueur $l(x)$ est d'autant plus grande que x est grand.*

2. *Avec la longueur de mot code l_1, on code 2^{l_1} entiers. Avec la longueur $l_2 > l_1$, on code $2^{l_2} > 2^{l_1}$ entiers.*

3. *On a toujours $l(x+1) \leq l(x) + 1$.*

Malheureusement, *BIN* n'est pas un code auto-délimité. Montrons que le code *Fibo* est non seulement auto-délimité, universel mais également ICL.

Proposition 2.7 *Le code Fibo : $\mathbb{N} \to \mathcal{B}^+$ est un code ICL.*

Preuve. Soit $l_{Fibo} : \mathbb{N} \to \mathbb{N}$ la fonction de longueur des mots code de Fibonacci. Nous savons que $l_{Fibo}(x)$ est l'indice du plus grand nombre de Fibonacci inférieur ou égal à $x+1$.

Le code *Fibo* est ICL car les trois propriétés suivantes sont vérifiées :

1. l_{Fibo} est une fonction croissante. En effet, pour $x, y \in \mathbb{N} : x < y$ tels que $l_{Fibo}(x) = n_x$ et $l_{Fibo}(y) = n_y$, on a $F_{n_x} \leq x+1 < F_{n_x+1}$ et $F_{n_y} \leq y+1 < F_{n_y+1}$. Dès lors, $F_{n_x} < F_{n_y+1}$. Puisque $F_i < F_j \iff 1 < i < j$, on a $n_x < n_y + 1$ et donc $n_x \leq n_y$.

2. l_{Fibo} est une fonction concave. En effet, $l_{Fibo}(x) = l \iff F_l \leq x+1 < F_{l+1}$. Dès lors, on peut représenter $F_{l+1} - F_l = F_{l-1}$ entiers avec la longueur de mot code l. Puisqu'étant donné $l_1 < l_2$ deux longueurs de mots code, on a $F_{l_1-1} \leq F_{l_2-1}$, on peut coder au moins autant d'entiers avec l_2 bits que le nombre d'entiers que l'on peut coder avec l_1 bits.

3. L'accroissement de l_{Fibo} est limité à 1 entre deux entiers consécutifs. En effet, lorsque $l_{Fibo}(x) = l$, on a $x+1 < F_{l+1}$ et dès lors $x+2 \leq F_{l+1}$. Supposons $l_{Fibo}(x+1) > l+1$. Alors $F_{l+2} \leq x+2$ ce qui est impossible vu que $F_{l+1} < F_{l+2}$. Donc $l_{Fibo}(x+1) \leq l+1$. □

2.2.4.2.4 Code *PrefFibo*

Les codes présentés dans la littérature [Eli75, ER78, AF87, Sto88, BCW90, LV93], permettant de coder tous les entiers de manière auto-délimitée, sont rarement ICL. Les mots code sont très souvent composés de plusieurs parties (par exemple, $BIN(x)$ préfixé de $\overline{l(x)}$ dans le cas du code f_2 de l'exemple 2.14). Pour certains entiers consécutifs $x-1$ et x, il y a une augmentation de longueur d'au moins 1 bit dans plus d'une des parties composant le mot code. Dès lors, il y a une augmentation de longueur d'au moins deux bits entre $x-1$ et x. Par exemple, bien que l'on ait $|f_2(3)| = 7$ et $|f_2(4)| = 8$, on a aussi $|f_2(7)| = 8$ et $|f_2(8)| = 11$!

Les codes f_0 (exemple 2.12) et *Fibo* sont ICL. Malheureusement, le code f_0 n'est pas universel, il n'est donc pas intéressant dans le contexte de la compression. De la même manière, Apostolico et Fraenkel ont montré que *Fibo* n'est pas asymptotiquement optimal.

Dès lors, une question très intéressante autant d'un point de vue théorique que d'un point de vue pratique est : **existe-t-il un code, de $\mathbb{N}$ dans $\mathcal{B}^+$, qui soit auto-délimité, asymptotiquement optimal et ICL ?** Nous répondons ici par l'affirmative et proposons le code *PrefFibo* basé sur un préfixage de BIN à l'aide du code de Fibonacci.

Commençons par définir le code fb dont les mots code $fb(x)$ sont obtenus par simple préfixage de $BIN(x)$ par la longueur $l(x)$ auto-délimitée à l'aide du code *Fibo*.

Définition 2.17 *Soit* $fb : \mathbb{N} \to \mathcal{B}^+ : x \mapsto fb(x) = Fibo(l(x))BIN(x)$.

C'est un code auto-délimité. En effet, supposons pour $x, y \in \mathbb{N}$, *que* $fb(x)$ *soit préfixe de* $fb(y)$.

1^{er} *cas :* $Fibo(l(x)) \neq Fibo(l(y))$.

C'est impossible car cela sous-entend que $Fibo(l(x))$ *est préfixe de* $Fibo(l(y))$ *ou que* $Fibo(l(y))$ *est préfixe de* $Fibo(l(x))$. *Cela ne peut pas se produire car Fibo est un code auto-délimité.*

$2^{ème}$ *cas :* $Fibo(l(x)) = Fibo(l(y))$.

Dès lors $l(x) = l(y)$ *car Fibo est un code. Donc* $|BIN(x)| = |BIN(y)|$ *et donc* $x = y$.

Étant donné le mot code $fb(x)$, il est facile de retrouver x : on commence par décoder $Fibo(l(x))$ et on sait alors combien de bits comporte $BIN(x)$.

Le lemme 2.8 montre que le code fb est asymptotiquement optimal.

Lemme 2.8 *Le code auto-délimité fb est asymptotiquement optimal.*

Preuve. Rappelons-nous que le code $f_2 : \mathbb{N} \to \mathcal{B}^+ : x \mapsto f_2(x) = \overline{l(x)}BIN(x)$ défini dans l'exemple 2.14, est asymptotiquement optimal. Pour prouver que fb est asymptotiquement optimal, nous allons montrer qu'il existe $x_0 \geq 0$ tel que $\forall x \geq x_0 : |fb(x)| \leq |f_2(x)|$ (suite à la remarque 2.4).

Recherchons un entier x_0 qui satisfasse la condition.

$$\begin{array}{rcl} |fb(x)| & \leq & |f_2(x)| \\ & \Updownarrow & \\ l_{Fibo}(l(x)) + l(x) & \leq & 2l(l(x)) + l(x) + 1 \\ & \Updownarrow & \\ l_{Fibo}(l(x)) & \leq & 2l(l(x)) + 1 \end{array}$$

Puisque l est une fonction croissante, effectuons le changement de variable $y = l(x)$.

Nous devons rechercher $y_0 \geq 1$ tel que $\forall y \geq y_0 : l_{Fibo}(y) \leq 2l(y) + 1$.

Puisque $l(y) = \lceil \log(y+1) \rceil$ on a $\log(y+1) \leq l(y)$. Nous allons rechercher y_0' tel que $\forall y \geq y_0' : l_{Fibo}(y) \leq 2\log(y+1) + 1$.

Soit $n = l_{Fibo}(y)$. Par construction du code de Fibonacci, n est le plus grand indice tel que $F_n \leq y + 1$. Puisque $F_n = \frac{\phi^n}{\sqrt{5}}$ arrondi à l'entier le plus proche, on a $\frac{\phi^n}{\sqrt{5}} - \frac{1}{2} \leq y + 1$. En passant par le logarithme en base ϕ, on obtient $n \leq \log_\phi(\sqrt{5}\, y + \frac{3\sqrt{5}}{2})$.

Il nous reste à trouver y_0'' tel que $\forall y \geq y_0'' : \log_\phi(\sqrt{5}\, y + \frac{3\sqrt{5}}{2}) \leq 2\log(y+1) + 1$. Il est facile de vérifier que la condition est vérifiée pour $y_0'' = 3$.

Dès lors, en effectuant le changement de variable dans l'autre sens, $l(x) \geq 3$ signifie $x \geq 4$.

Donc, $\forall x \geq 4 : |fb(x)| \leq |f_2(x)|$. □

Le tableau 2.3 compare les longueurs des mots code des codes BIN, f_0, f_2, $Fibo$ et enfin fb. La dernière colonne concerne le code ICL *PrefFibo* que nous développons ci-dessous.

On remarque immédiatement que f_2, $Fibo$ et fb ont des longueurs de mots code comparables. Lorsque x devient grand, fb est le code le plus intéressant, ensuite c'est f_2 et enfin $Fibo$ qui lui n'est pas asymptotiquement optimal.

Dans ce tableau, on peut voir que fb n'est pas ICL car il y a une différence de longueur de 2 bits entre $fb(x-1)$ et $fb(x)$ pour certains entiers x ($|fb(7)| = 7$ et $|fb(8)| = 9$ par exemple). Cette situation se produit pour l'entier x lorsque les deux conditions suivantes sont vérifiées. Nous disons alors que x est une *forte puissance*.

1. x est une puissance de 2 : $x = 2^p$, avec $p > 0$.

   Dans ce cas, $BIN(x)$ est 1 bit plus long que $BIN(x-1)$: $p + 1 = l(x) = l(x-1) + 1$.

2. $l(x) + 1$ est un nombre de Fibonacci : $l(x) + 1 = p + 2 = F_n$ avec $n > 2$.

x	$\|BIN(x)\|$	$\|f_0(x)\|$	$\|f_2(x)\|$	$\|Fibo(x)\|$	$\|fb(x)\|$	$\|PrefFibo(x)\|$
0	1	1	4	2	4	4
1	1	2	4	3	4	5
2	2	3	7	4	6	6
3	2	4	7	4	6	6
4	3	5	8	5	7	7
5	3	6	8	5	7	7
6	3	7	8	5	7	8
7	3	8	8	6	7	8
8	4	9	11	6	9	9
9	4	10	11	6	9	9
10	4	11	11	6	9	9
11	4	12	11	6	9	9
12	4	13	11	7	9	9
13	4	14	11	7	9	9
14	4	15	11	7	9	9
15	4	16	11	7	9	9
16	5	17	12	7	10	10
17	5	18	12	7	10	10
18	5	19	12	7	10	10
19	5	20	12	7	10	10
20	5	21	12	8	10	10
21	5	22	12	8	10	10
22	5	23	12	8	10	10
23	5	24	12	8	10	10
24	5	25	12	8	10	10
25	5	26	12	8	10	10
...						
50	6	51	13	9	11	12
100	7	101	14	11	13	13
500	9	501	18	14	15	15
1000	10	1001	19	16	16	16
10000	14	10001	23	20	21	21
100000	17	100001	28	25	24	24
1000000	20	1000001	31	30	28	28
10000000	24	10000001	35	35	32	32
1000000000	30	1000000001	41	44	38	38

TAB. 2.3 - *Comparatif des longueurs des mots code*

Les premières fortes puissances sont $2, 8, 64, 2048, 524288 \ldots$. Elles sont de plus en plus espacées. Les sauts de longueur de 2 bits pour les mots codes de fb ne peuvent se produire que pour ces entiers.

Nous allons maintenant modifier légèrement le code fb pour le rendre ICL. Appelons *PrefFibo* le code résultant de ces modifications.

Remarque 2.5 *Bien que fb ne soit pas ICL, il possède les deux autres propriétés des codes ICL :*

- *$|fb(x)|$ est une fonction croissante. C'est facile à vérifier puisque $l(x)$ est croissante et $l_{Fibo}(x)$ également.*
- *Si $h_1 < h_2$ alors il y a plus de mots code de longueur h_2 que de mots code de longueur h_1. En effet, comme fb est construit à partir du code BIN, le nombre de mots code de longueur h est le double du nombre de mots code de longueur $h-1$ (ou $h-2$ s'il n'existe aucun mot code de longueur $h-1$ à cause du codage d'une forte puissance). Les longueurs des "marches" de la représentation graphique de fb doublent à chaque étape.*

Considérons trois puissances successives de 2 :

$$x_0 = 2^i, x_1 = 2^{i+1} \text{ et } x_2 = 2^{i+2} \text{ avec } i \geq 1$$

Ces trois entiers correspondent à trois augmentations successives de la longueur des mots code de fb :

$$|fb(x_0 - 1)| < |fb(x_0)| = |fb(x_1 - 1)| < |fb(x_1)| = |fb(x_2 - 1)| < |fb(x_2)|$$

Nous allons définir les mots codes $PrefFibo(x)$ pour $x_0 \leq x < x_1$ de telle sorte que les trois conditions d'un code ICL soient vérifiées sur $[x_0, x_2]$:

1. $|PrefFibo(x)|$ est une fonction croissante sur $[x_0, x_2]$.
2. $|PrefFibo(x)|$ est une fonction concave, c'est-à-dire que les "marches" de la représentation graphique de $|PrefFibo(x)|$ sont de plus en plus longues (au sens large) sur $[x_0, x_2]$.
3. Les marches de la représentation graphique de $|PrefFibo(x)|$ aient une hauteur de 1 sur $[x_0, x_2]$.

Distinguons deux cas :

1er cas : x_1 n'est pas une forte puissance : $|fb(x_1)| = |fb(x_1 - 1)| + 1$.

Dans ce cas, les trois conditions sont vérifiées pour le code fb sur $[x_0, x_2]$ puisqu'il n'y a pas de saut de longueur de 2 bits entre $fb(x_1 - 1)$ et $fb(x_1)$.

On code 2^i entiers avec la longueur $h = |fb(x_0)|$ et 2^{i+1} entiers avec la longueur $h+1 = |fb(x_1)|$ (voir figure 2.5).

Dans ce cas, on définit $PrefFibo(x) = fb(x)$ pour $x \in [x_0, x_1[$.

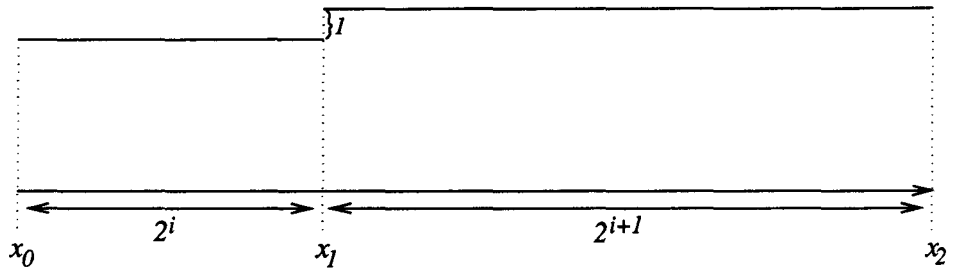


FIG. 2.5 - $|fb(x)|$ *sur* $[x_0, x_2]$ *lorsque* x_1 *n'est pas une forte puissance*

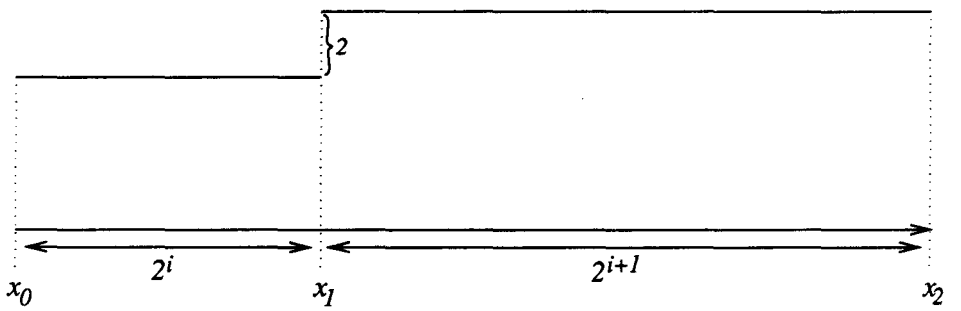


FIG. 2.6 - $|fb(x)|$ *sur* $[x_0, x_2]$ *lorsque* x_1 *est une forte puissance*

$2^{\text{ème}}$ cas : x_1 est une forte puissance : $|fb(x_1)| = |fb(x_1 - 1)| + 2$.

Dans ce cas, fb n'est pas ICL sur $[x_0, x_2]$ car il y a un saut de longueur 2 bits entre $fb(x_1 - 1)$ et $fb(x_1)$ (voir figure 2.6).

Dans ce cas, on va couper la marche $[x_0, x_1[$ en deux marches ayant chacune une longueur de 2^{i-1}. Soit $x'_0 = x_0 + 2^{i-1}$. On va s'arranger pour avoir les longueurs de mots code suivantes (voir figure 2.7) :

- $\forall x \in [x_0, x'_0[: |PrefFibo(x)| = |fb(x)|$.
- $\forall x \in [x'_0, x_1[: |PrefFibo(x)| = |fb(x)| + 1$.

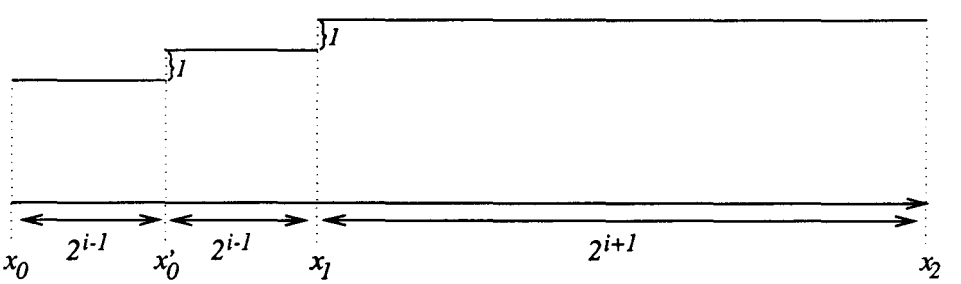


FIG. 2.7 - $|PrefFibo(x)|$ *sur* $[x_0, x_2]$ *lorsque* x_1 *est une forte puissance*

Pour cela, on définit :

- $\forall x \in [x_0, x'_0[: PrefFibo(x) = fb(x)$.
- $\forall x \in [x'_0, x_1[: PrefFibo(x) = Fibo(l(x))\,0\,BIN(x)$.

On a augmenté artificiellement les longueurs des mots code sur $[x'_0, x_1[$ en intercalant un bit 0 entre $Fibo(l(x))$ et $BIN(x)$. Puisque les mots code $BIN(x)$ commencent toujours par un 1 (à l'exception de $BIN(0)$), si le processus de décodage rencontre un 0 juste après le codage de $Fibo(l(x))$, il lui suffit tout simplement de l'ignorer (sauf dans le cas $x = 0$ évident à détecter).

Cette modification rend *PrefFibo* ICL sur l'intervalle $[x_0, x_2]$. En effet :

1. $|PrefFibo|$ est une fonction croissante sur $[x_0, x_2]$: par construction de $|PrefFibo|$.
2. Les marches de la représentation graphique sont de plus en plus longues au sens large : $x'_0 - x_0 = x_1 - x'_0 < x_2 - x_1$.
3. Toutes les marches ont une hauteur de 1.

Remarquons que le nombre d'entiers représentés par fb avec la longueur de mot code $|fb(x_0 - 1)|$ est 2^{i-1}. La représentation du code *PrefFibo* comporte donc trois marches successives de longueur 2^{i-1} :

$$[x_0 - 2^{i-1}, x_0[, [x_0, x'_0[et[x'_0, x_1[$$

Le code *PrefFibo* est complètement construit, à partir de fb, par application de ce processus à tous les triplets (x_0, x_1, x_2) de puissances successives de 2. On l'applique également au triplet $(0, 2, 4)$ pour rectifier la première marche. Il est facile de voir que cette itération ne produit jamais une marche de longueur plus courte (au sens strict) que la marche précédente. Certaines marches consécutives sont de même longueur. Chaque marche de longueur h, qui précède un saut de longueur de 2 bits pour le code fb, est divisée en 2 marches de longueur $\frac{h}{2}$ dans *PrefFibo*, le saut de deux bits est ainsi réparti sur les 2 marches.

Nous avons donc construit le code *PrefFibo* qui est auto-délimité et ICL. Le théorème 2.9 montre qu'il est aussi asymptotiquement optimal.

Théorème 2.9 *Le code auto-délimité PrefFibo* : $\mathbb{N} \to \mathcal{B}^+$ *est asymptotiquement optimal.*

Preuve. Puisque fb est un code asymptotiquement optimal et que, $\forall x \in \mathbb{N}$, $|PrefFibo(x)| \leq |fb(x)| + 1$, le rapport $\frac{|PrefFibo(x)|}{l(x)}$ tend bien vers 1 lorsque x tend vers $+\infty$ (remarque 2.4). □

En pratique, le tableau de la figure 2.3 montre que pour les petits entiers, le code ICL *Fibo* est plus intéressant que le code ICL *PrefFibo*. Ce n'est qu'à partir de 10^5 environ que *PrefFibo* devient plus intéressant que *Fibo*. Si la majorité des entiers à coder, dans la phase de codage d'un compresseur, sont inférieurs à 10^5, on préfèrera donc utiliser le code *Fibo*.

2.3 Méthodes classiques de compression conservative

Dans cette section, nous décrivons les méthodes les plus courantes qui permettent de comprimer l'information. Nous nous focalisons sur les méthodes polyvalentes, c'est-à-dire les méthodes qui compriment indifféremment les textes, images, sons... Bien entendu, pour un type de données spécifique, une méthode dédicacée serait plus appropriée mais elle est difficilement adaptable à d'autres types de données.

Nous n'entrons pas dans les détails des méthodes, de nombreux ouvrages les étudient en profondeur et les comparent ([LH87, Sto88, BCW90, Zip90, HM91, Nel93, CR94, Del96a, CL97]).

Nous commençons par les méthodes statistiques pour en venir ensuite aux méthodes par substitution.

2.3.1 Codages statistiques

Les méthodes de compression par codages statistiques traitent les textes symbole par symbole. Elles exploitent la "non-uniformité" des probabilités d'apparition des symboles du texte pour en diminuer la taille.

L'idée est de choisir un codage adéquat permettant de coder les caractères les plus courants de manière beaucoup plus courte que les caractères les moins courants.

Deux cas sont possibles: soit la distribution de probabilité des symboles est connue, soit elle n'est pas connue. Le premier cas se produit par exemple lorsque l'on comprime un texte de langue française. Les études linguistiques permettent de connaître les fréquences attendues des lettres. Dans le deuxième cas, le nombre d'occurrences de chaque symbole est comptabilisé dans le texte à comprimer et divisé par la longueur totale du texte, ce qui nous donne la *fréquence observée* du symbole.

Le codage de Huffman code chaque symbole sur un nombre déterminé de bits tandis que le codage arithmétique utilise un codage du texte grâce à des intervalles de nombres réels, calculés grâce aux fréquences d'apparition des symboles.

2.3.1.1 L'entropie

Nous avons vu en 2.2.3.1 que l'entropie est la borne inférieure de la moyenne des longueurs des mots code. Dans le cas des codages statistiques, les mots source sont les symboles de l'alphabet et les mots code leurs traductions.

Soient $p_a, p_b, p_c, p_d \ldots$ les fréquences observées des lettres de l'alphabet pour S:

$$\forall i \in \mathcal{A} : p_i = \frac{\#i}{n} \Rightarrow 0 \leq p_i \leq 1$$

où $\#i$ est le nombre d'occurrences du symbole i et n est la longueur de la séquence.

L'entropie H se calcule par la formule:

$$H = -\sum_{i \in \mathcal{A}} p_i \log p_i$$

Il n'est pas possible de comprimer la séquence, à l'aide d'un codage statistique, pour obtenir une taille inférieure à $n \times H$.

Exemple 2.17
Soit $\mathcal{N} = \{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$ *l'alphabet et*
$S =$ `TTTCATTGTACCTGTTATTAGATTAGTCTAGATTTAGTACTATATCTT` *une séquence.*
On a $|S| = 48$, $p_{\texttt{A}} = \frac{1}{4}, p_{\texttt{C}} = \frac{1}{8}, p_{\texttt{G}} = \frac{1}{8}, p_{\texttt{T}} = \frac{1}{2}$

$$\Longrightarrow H = -\left(\frac{1}{4} \times (-2) + \frac{1}{8} \times (-3) + \frac{1}{8} \times (-3) + \frac{1}{2} \times (-1)\right) = 1.75$$

Chaque symbole nécessitera donc (en moyenne) au moins 1.75 bits pour être codé. Au mieux, en utilisant un codage statistique, on obtiendra une séquence comprimée de 84 bits, ce qui donnera un taux de compression de 12.5%.

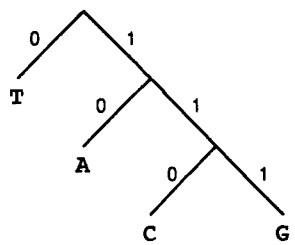
Cette notion d'entropie est fréquemment utilisée pour calculer, d'un point de vue statistique, le contenu en information d'une séquence, mais elle ne prend en compte que les notions de fréquences des lettres.

2.3.1.2 Codage de Huffman

2.3.1.2.1 Principe

L'algorithme de Huffman lit les symboles du texte les uns après les autres et les code chacun par une suite de bits d'autant plus courte que la fréquence d'apparition du symbole est grande [Huf52]. Pour que la décompression soit possible et non ambiguë, le code doit être auto-délimité.

La figure 2.8 montre un exemple de code pour la séquence de l'exemple 2.17. L'arbre représenté est appelé l'*arbre de Huffman*.



Traduction de S: 000110100011101011011001110010000101111 0... qui contient 84 bits.

FIG. 2.8 - *Exemple de codage de Huffman pour la séquence d'ADN de l'exemple 2.17*

Le décompresseur doit bien entendu connaître l'arbre de Huffman utilisé. Si le compresseur et le décompresseur connaissent le modèle statistique des données (exemple de modèle: texte français), ils connaissent l'arbre de Huffman. Si ce n'est pas le cas, le compresseur code les fréquences des mots source en tête du fichier comprimé. Cela augmente légèrement la taille mais garantit que le compresseur et le décompresseur travaillent avec le même arbre de Huffman.

2.3.1.2.2 Construction de l'arbre de Huffman

L'algorithme de construction de l'arbre de Huffman considère un ensemble de "sous-arbres" à relier entre eux pour construire l'arbre final. À chaque étape, les deux sous-arbres de poids minimaux T_1 et T_2 sont reliés pour n'en former qu'un seul: l'arbre T. Pour cela, une nouvelle racine est créée, elle possède deux fils: le sous-arbre T_1 et le sous-arbre T_2. L'arc qui relie la racine à T_1 est étiqueté 0 et l'arc qui relie la racine à T_2 est étiqueté 1. Le poids de T est égal à la somme des poids des deux sous-arbres T_1 et T_2. À l'origine, chaque symbole de l'alphabet correspond à un arbre trivial ne contenant qu'un seul nœud. Son poids est égal à la probabilité (ou le nombre d'occurrences) du symbole. Le processus se termine lorsqu'il ne reste plus qu'un seul arbre.

Au point de départ, dans l'exemple de la figure 2.8, on a 4 sous-arbres, ne contenant qu'une seule feuille chacun, de poids respectifs 12, 6, 6 et 24 (on utilise les nombres d'occurrences). On regroupe alors les deux sous-arbres correspondant à C et G, ce qui donne un sous-arbre de poids 12. On continue jusqu'à obtention de l'arbre final (voir figure 2.9).

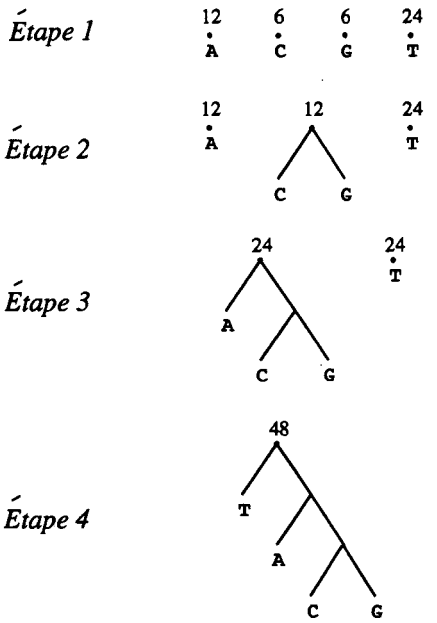


FIG. 2.9 - *Construction de l'arbre de Huffman*

Par construction, l'arbre est complet: le degré de chaque nœud est soit 0, soit 2. Il est possible de montrer que le code construit est optimal [Huf52].

2.3.1.2.3 Une variante: le codage de Huffman adaptatif

L'algorithme de Huffman nécessite la connaissance des fréquences des mots source avant la compression et avant la décompression. Cela implique une perte de place suite au codage de ces informations. De plus, du temps sera perdu lors de la compression pour déterminer les fréquences. Il n'est pas possible, avec cette méthode, d'imaginer un modem "en ligne" comprimant automatiquement les informations envoyées puisque le modem ne connaîtra les fréquences qu'après envoi total du message.

Dans l'algorithme de Huffman adaptatif [Fal73, Gal78], un arbre déterminé est imposé au point de départ de la compression. Il s'agit en général d'un arbre dans lequel tous les mots code ont la même longueur. Au fur et à mesure de la compression, l'algorithme modifie son arbre en fonction des fréquences observées dans la partie lue. L'arbre final sera donc le même que dans le cas de l'algorithme de Huffman non-adaptatif. Chaque symbole n'est pas codé de la même façon d'un bout à l'autre. La décompression utilise le même arbre imposé au point de départ et le modifie également en fonction des fréquences observées dans ce qui a été décodé. De ce fait, pour une position déterminée dans la suite considérée, la compression et la décompression travailleront sur le **même arbre** et utiliseront donc la même traduction du symbole à cette position.

2.3.1.2.4 Efficacité

Il est possible de montrer que, dans le cas ou les fréquences observées sont toutes des puissances de 2 ($\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16} \cdots$), l'entropie est atteinte [Zip90]. C'est le cas de la séquence de l'exemple 2.17 (figures 2.8 et 2.9). Si les fréquences ne sont pas des puissances de 2, alors l'entropie ne pourrait être atteinte que si le codage de Huffman permettait de coder chaque symbole sur un nombre fractionnaire de bits. Ce qui n'est pas possible.

2.3.1.3 Codage arithmétique

Le codage de Huffman ne sera jamais capable de coder un symbole sur moins de 1 bit. Lorsque la probabilité d'apparition d'un symbole devient très grande, la taille optimale de son mot code devient très petite par rapport à 1 bit. Le codage s'éloigne donc très fort de l'optimalité de l'entropie. Le codage arithmétique remédie à cet inconvénient [RL79, Gua80].

2.3.1.3.1 Principe

Le codage arithmétique utilise un intervalle de nombres réels pour traduire une séquence. Chaque symbole réduit l'intervalle courant à un intervalle qui lui est inclus. L'importance de la réduction dépend de la probabilité (ou de la fréquence observée) du symbole traduit.

Supposons que la répartition de la probabilité des quatre symboles `A`, `C`, `G` et `T` soit celle donnée par l'exemple 2.17. On divise l'intervalle $[0, 1[$ en 4 sous-intervalles de largeurs proportionnelles aux probabilités des 4 symboles (figure 2.10).

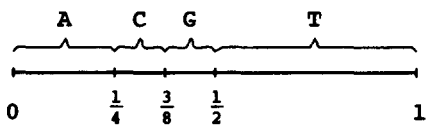


FIG. 2.10 - *Division de l'intervalle* $[0, 1[$ *en sous-intervalles selon la distribution de probabilité* $p_A = \frac{1}{4}, p_C = \frac{1}{8}, p_G = \frac{1}{8}, p_T = \frac{1}{2}$

Le texte est codé de gauche à droite, symbole par symbole. Au point de départ, lorsque rien n'a été codé, l'intervalle de sortie est $[0, 1[$. À chaque nouveau symbole, l'intervalle est réduit au sous-intervalle déterminé, proportionnellement, par la répartition des probabilités. Ainsi, si la séquence à comprimer est `ATCGC` et que le modèle est celui illustré à la figure 2.10, le codage de `A` donne l'intervalle $[0, 0.25[$. Le codage de `AT` correspond à la réduction de $[0, 0.25[$ à sa moitié supérieure : $[0.125, 0.25[$. On continue avec les symboles `C`, `G` et `C`, ce qui nous donne l'intervalle final $[0.162597656, 0.162841796[$ (figure 2.11).

La compression consiste alors simplement à écrire le nombre de symboles codés (5) suivi de la borne inférieure de l'intervalle (0.162597656).

Etant donné le nombre de symboles (5) et la borne (0.162597656), la décompression est relativement simple. La borne appartient à $[0, 0.25[$. Le premier symbole est donc un `A`. Pour décoder le deuxième symbole, on applique la transformation dilatant $[0, 0.25[$ en $[0, 1[$ à la borne 0.162597656. On obtient 0.6503906 qui est inclus à $[0.5, 1[$. On produit donc le symbole `T`. On continue ainsi jusqu'à ce que tous les symboles aient été décodés.

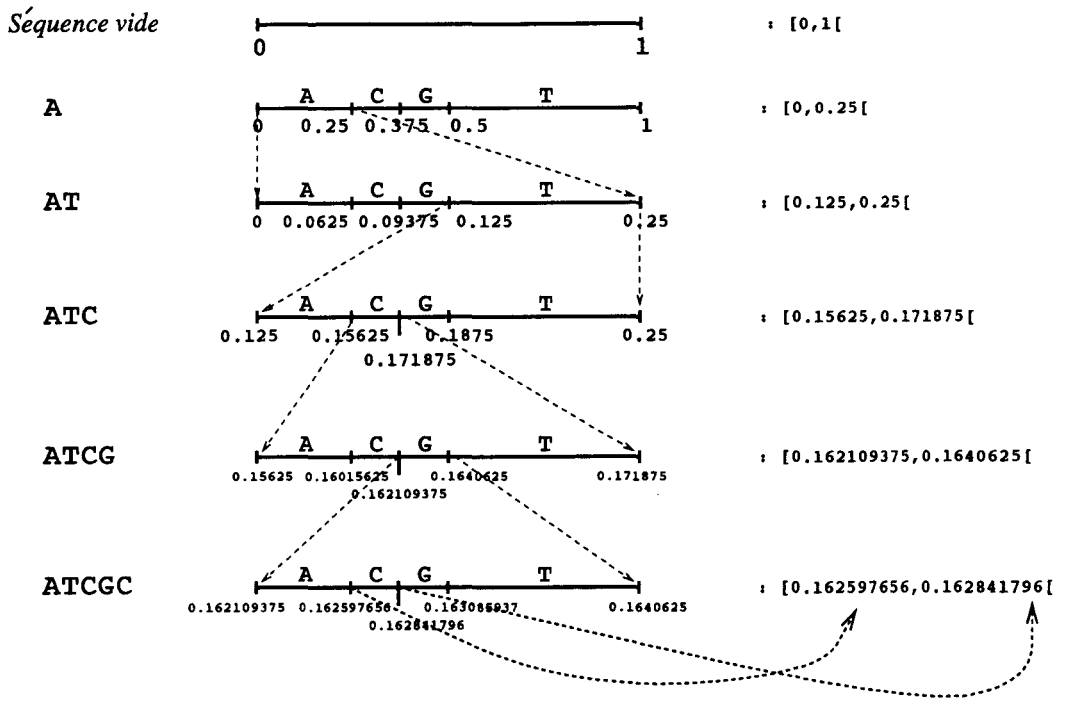


FIG. 2.11 - *Compression arithmétique de* ATCGC *étant donné le modèle de la figure 2.10*

2.3.1.3.2 Implémentation à l'aide de nombres entiers

L'intervalle se restreignant à chaque étape, on atteint très vite les limites de précision du codage des nombres réels sur ordinateur. Seules les séquences très courtes peuvent être comprimées de la sorte.

La méthode peut être adaptée pour n'utiliser que des nombres entiers. En effet, lorsque les deux bornes réelles se rapprochent, il arrive un moment où les premiers chiffres deviennent identiques. Par exemple, dans l'intervalle $[0.15625, 0.171875[$ codant pour ATC à la figure 2.11, la première décimale 1 est identique. Elle ne changera donc plus jamais puisque les deux bornes ne peuvent que se rapprocher. Dès lors, on peut coder "1" dans le fichier de sortie et travailler avec l'intervalle $[0.5625, 0.71875[$ dont la largeur est plus grande.

Dans l'adaptation en nombres entiers de la méthode, les deux bornes de l'intervalle sont entières. Elles se rapprochent de plus en plus. A chaque fois que le premier chiffre de la borne inférieure est identique au premier chiffre de la borne supérieure, il est codé en sortie. Il est alors supprimé des deux bornes qui sont décalées de un chiffre vers la gauche. Les réductions d'intervalles sont proportionnelles aux fréquences des symboles. Bien entendu, les calculs en entiers nécessitent des arrondis qui feront perdre de la précision à l'algorithme mais, si les mêmes arrondis sont utilisés à la compression et à la décompression, la version décomprimée du texte est identique à la version de départ. Seul le taux de compression sera un peu moins bon.

2.3.1.3.3 Version adaptative

Les fréquences de symboles doivent être codées dans le texte de sortie. Il est possible de concevoir un modèle adaptatif comme dans le cas de l'algorithme de Huffman [WNC87]. La

répartition des sous-intervalles de $[0, 1[$ est fixée au commencement du codage. Elle change au cours de l'exécution de l'algorithme.

2.3.1.3.4 Efficacité

L'algorithme de codage arithmétique produit une meilleure compression que celui de Huffman. En effet, un symbole peut être codé sur un nombre fractionnaire de bits.

Il est possible de montrer que le codage arithmétique atteint asymptotiquement (c'est-à-dire lorsque la taille de la séquence tend vers l'infini) l'entropie [Zip90]. Certains détails empêchent de l'atteindre pour une séquence finie :

- Besoin de coder, dans le texte de sortie, la liste des fréquences de symboles.
- "Gaspillage", à la fin du codage, d'une partie de l'intervalle qui aurait permis de coder encore d'autres symboles.

En pratique, la méthode est tout de même la plus satisfaisante des compressions statistiques.

2.3.1.4 Ordres supérieurs

Les codages statistiques tels que l'algorithme de Huffman ou le codage arithmétique peuvent être étendus pour exploiter les biais de répartition des *t-uples* (un t-uple est une suite de symboles de longueur fixée). On parle alors d'algorithme d'ordre 1, d'ordre 2,... en fonction de la longueur choisie pour les t-uples. L'algorithme habituel est d'ordre 0.

Le texte à comprimer est décomposé en une séquence de t-uples non chevauchant. Une probabilité (ou fréquence observée) est associée à chaque t-uple. Grâce à ces probabilités, un arbre de Huffman (ou une division de $[0, 1[$ en sous-intervalles pour le codage arithmétique) peut être construit et l'algorithme fonctionne exactement de la même manière que pour l'ordre 0.

Il s'agit donc simplement d'un changement d'alphabet. Imaginons un texte à comprimer sur l'alphabet binaire $\mathcal{B}$. Considérons l'extension de l'algorithme de Huffman à l'ordre 2. Cela revient à faire fonctionner l'algorithme de Huffman sur l'alphabet $\mathcal{B}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ et donc à construire un arbre de Huffman comportant une feuille pour chacun des huit éléments de $\mathcal{B}^3$. Sur l'exemple de la figure 2.12, le codage du triplet 000 est 11110, celui de 010 est 00, etc.

2.3.2 Substitutions de facteurs

Les méthodes de compression par substitutions de facteurs substituent, à des facteurs de longueurs variables, des mots code plus courts. Chaque mot code représente l'indice du facteur dans un dictionnaire. Le dictionnaire peut être construit à l'avance ou au fur et à mesure de la compression. On peut par exemple se servir du dictionnaire Larousse si le texte à comprimer est en français. Dans le cas d'un dictionnaire construit au fur et à mesure, il contient un sous-ensemble des facteurs de la séquence. Nous ne nous intéresserons pas au cas des dictionnaires connus d'avance. Nous supposerons qu'aucune information ne permet de connaître, a priori, l'ensemble des mots qui apparaissent dans la séquence à comprimer.

La majorité des programmes de compression à base de dictionnaires utilisent une des deux méthodes développées par Jacob Ziv et Abraham Lempel en 1977 et 1978 [ZL77, ZL78]. Ces

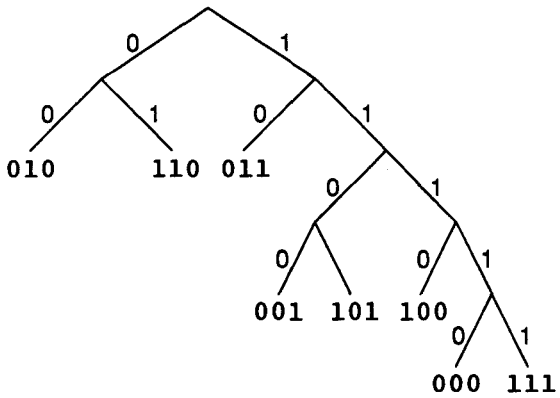


FIG. 2.12 - *Exemple d'arbre de Huffman sur l'alphabet* $\mathcal{B}^3$

deux algorithmes, appelés LZ77 et LZ78, parcourent la séquence à comprimer de gauche à droite. Ils remplacent les facteurs répétés par des pointeurs vers l'endroit où ils sont déjà apparus précédemment dans le texte. Bien qu'elles soient souvent confondues dans la littérature, ces deux méthodes diffèrent fondamentalement en ce qui concerne la construction du dictionnaire.

Ce n'est qu'après la sortie d'un article de Terry Welch [Wel84], en 1984, expliquant comment implémenter pratiquement la méthode LZ78, que des compresseurs performants ont vu le jour. C'est le cas de ARC sous MS-DOS, de COMPRESS sous UNIX, le format graphique .GIF inclut une compression LZ78,... De nouveaux compresseurs, utilisant des implémentations particulières de LZ77, ont ensuite été créés. On trouve par exemple PKZIP et ARJ sous MS-DOS, GZIP sous UNIX,.... On peut dire qu'à l'heure actuelle, LZ77 et LZ78 sont les deux références en matière de compression à usage général.

2.3.2.1 Compression avec fenêtre coulissante : LZ77

La méthode LZ77 comprime le texte de gauche à droite et utilise la fin de la partie de texte déjà comprimée comme dictionnaire. La principale structure de données de LZ77 est une fenêtre de N caractères divisée en deux parties. Les $N-T$ premiers caractères constituent le dictionnaire. Les T derniers caractères constituent le "tampon de lecture" à comprimer grâce au dictionnaire.

La fenêtre est dite coulissante parce qu'elle glisse sur le texte de gauche à droite du début à la fin du texte. Initialement, elle est située $N-T$ caractères avant le début du texte de telle sorte que le tampon de lecture commence au début du texte. Les $N-T$ caractères du dictionnaire sont alors des espacements.

A tout moment, l'algorithme recherche, dans les $N-T$ premiers caractères de la fenêtre, le plus long facteur qui se répète au début du tampon de lecture. Il doit être de taille maximale T. Cette répétition est alors codée par le triplet (p, l, c). L'entier p est la distance entre le début du tampon et la position de la répétition dans le dictionnaire, l est la longueur de la répétition et c est le premier caractère du tampon qui diffère du caractère correspondant dans le dictionnaire.

Après avoir codé cette répétition, la fenêtre coulisse de $l+1$ caractères vers la droite.

Le codage du caractère c ayant provoqué la différence est indispensable dans le cas où aucune répétition n'est trouvée dans le dictionnaire. Par convention, on code alors $(0, 0, c)$.

Supposons que le texte à comprimer soit `Le magicien dit "abracadabra"`, que la taille de la fenêtre soit $N = 13$ et que la taille du tampon de lecture soit $T = 5$ (voir figure 2.13).

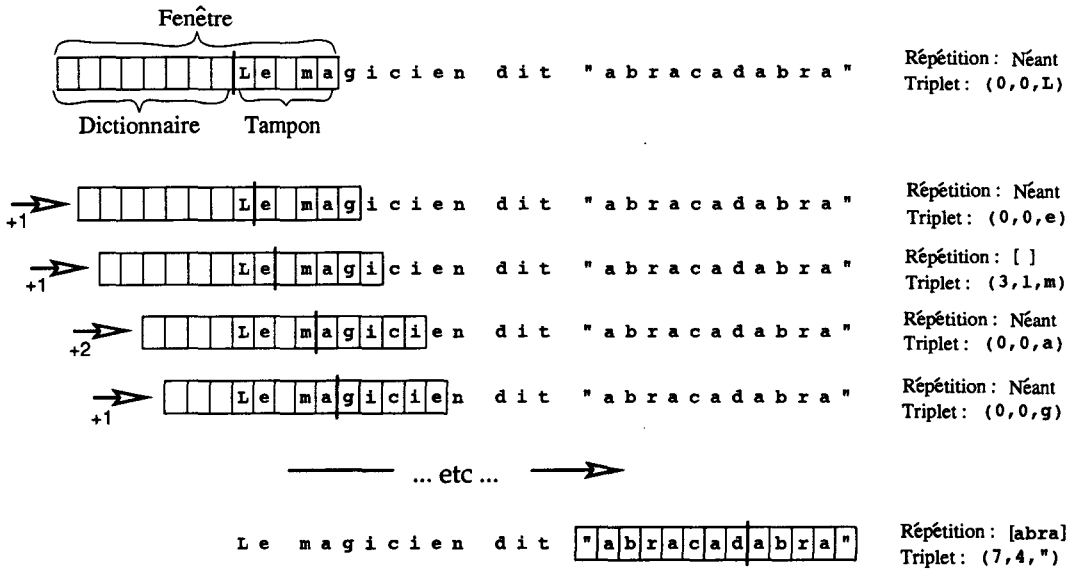


FIG. 2.13 - *Exemple de fenêtre avec $N = 13$ et $T = 5$*

Au point de départ, aucune répétition n'est trouvée puisque le dictionnaire ne contient que des espacements. Le codage des deux premiers caractères du texte est alors (0,0,L) (0,0,e), la fenêtre se déplaçant à chaque fois d'un caractère vers la droite. Ensuite l'espacement est rencontré. Comme il est déjà présent dans le dictionnaire 3 caractères avant le début du tampon et puisque le caractère `m` termine la répétition, le triplet est (3,1,m). On continue alors : (0,0,a)(0,0,g)...

Supposons qu'une partie du texte ait déjà été parcourue et que la fenêtre se situe en position 17, sur le premier caractère ". Le tampon de lecture est alors `abra"`. Le plus long mot présent dans le dictionnaire et commençant le tampon contient 4 caractères, c'est `abra`. Dans le dictionnaire, il se situe 7 caractères avant le début du tampon. Le caractère du tampon marquant la différence avec le facteur correspondant du dictionnaire est ". Le triplet est donc (7,4,"). La suite de tous les triplets à coder est donc (0,0,L) (0,0,e) (3,1,m) (0,0,a) (0,0,g) (0,0,i) (0,0,c) (2,1,e) (0,0,n) (0,0,) (0,0,d) (5,1,t) (4,1,") (0,0,a) (0,0,b) (0,0,r) (3,1,c) (2,1,d) (7,4,").

On associe en général un nombre de bits fixe pour chacun des triplets. En pratique, un codage efficace est obtenu si N et $N - T$ sont des puissances de 2 : $N = 2^{e_1}$ et $N - T = 2^{e_2}$. Le nombre de bits nécessaires au codage de la position dans le dictionnaire est donc e_2. Le nombre de bits nécessaires au codage de la longueur de la répétition est e_1. Le nombre de bits nécessaires au codage du caractère marquant la différence entre le tampon et le dictionnaire est $\lceil \log \#\mathcal{A} \rceil$ où $\mathcal{A}$ est l'alphabet d'entrée. Le codage est d'autant plus intéressant que le mot répété est long. Pour avoir des chances d'obtenir de longues répétitions, le dictionnaire doit être de taille suffisante (de l'ordre de plusieurs milliers de caractères). Malgré tout, si la fenêtre est trop grande, il faut trop de bits pour coder chaque triplet et l'algorithme ne permet pas d'obtenir une bonne compression. La recherche d'une répétition dans le dictionnaire est très coûteuse en temps (de l'ordre de $(N - T) \times T$ opérations par une méthode en force brute), une trop grande fenêtre rend l'algorithme inutilisable. Larsson a proposé une méthode efficace

pour rechercher la plus longue répétition dans le dictionnaire [Lar96]. Elle maintient à jour l'arbre des suffixes du contenu du dictionnaire. La localisation du plus long facteur, débutant le tampon et répété dans le dictionnaire se fait alors en temps proportionnel à la longueur de ce facteur. L'arbre des suffixes est construit et maintenu à jour grâce à une adaptation de l'algorithme de Ukkonen [Ukk92, Ukk95]. La construction et la mise à jour de l'arbre des suffixes se font en un temps proportionnel à la longueur de la séquence à comprimer.

L'algorithme comprime les informations au fur et à mesure du déplacement de la fenêtre. Il oublie les parties de texte déjà comprimées à l'exception des caractères qui constituent la fenêtre. L'algorithme est donc "en ligne", les mots code sont écrits au fur et à mesure sur le périphérique de sortie avant la fin de la lecture du texte.

La méthode fonctionne bien sur la plupart des textes lorsque la fenêtre est de taille modérée ($N \leq 8192$), ceci pour les raisons suivantes :

- Beaucoup de mots et de fragments de mots sont suffisamment courants pour apparaître plusieurs fois dans une fenêtre. C'est le cas, par exemple, dans les textes en français, des mots et fragments "...sion", "...que", "de", "le", "la", "...ment", "ce",...
- Les mots rares ont tendance à être répétés à des positions très proches. C'est par exemple le cas du mot "fenêtre" dans cette section.
- Si une suite de caractères est répétée un certain nombre de fois consécutivement, le codage peut être très économe parce qu'il autorise la répétition à chevaucher le dictionnaire et le tampon. Supposons que l'on doive comprimer la suite de 16 répétitions consécutives de TC (voir figure 2.14). Les deux premiers triplets sont (0,0,T) et (0,0,C) car cette suite n'est pas encore connue. Ensuite, le début du tampon est positionné sur la répétition suivante. La plus longue répétition est TCTCTCTCTCTCTCTCTCTCTCTC (14 fois TC) 2 caractères avant. La compression se terminera donc avec le triplet (2,28,).

FIG. 2.14 - *16 Répétitions consécutives de* TC

La décompression est simple et rapide. A partir de la suite de triplets, le décodage s'effectue en faisant coulisser la fenêtre comme pour le codage. Le dictionnaire est reconstruit de gauche à droite ce qui permet l'interprétation correcte des triplets.

2.3.2.2 Compression LZ78

Dans l'algorithme LZ78, le dictionnaire n'est plus constitué d'une fenêtre coulissante mais il est construit à partir de tous les facteurs déjà rencontrés.

Tout comme dans LZ77, le texte est comprimé de gauche à droite. Au point de départ, le dictionnaire ne contient aucun facteur. Les facteurs sont numérotés à partir de 1. A tout moment, l'algorithme recherche le plus long facteur du dictionnaire qui concorde avec la suite des caractères du texte. Il suffit alors d'encoder le numéro de ce facteur et le caractère suivant du texte : (num, c). Ce couple détermine un nouveau facteur qui est ajouté au dictionnaire pour être utilisé comme référence dans la suite du texte. L'entier *num* est codé en format fixe sur $\lceil \log \#D \rceil$ bits où $\#D$ est la taille du dictionnaire au moment courant.

Prenons la compression du texte **aaabbabaabaaabab** (voir tableau 2.4). Au point de

Parcours du texte :	a	aa	b	ba	baa	baaa	bab
Numéro de facteur :	1	2	3	4	5	6	7
Codage :	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

TAB. 2.4 - *Codage de* **aaabbabaabaaabab** *par LZ78*

départ, le dictionnaire ne contenant aucun facteur, aucune concordance ne peut être trouvée entre les caractères du début du texte et un facteur. On code donc (0,a) ce qui signifie le facteur vide suivi du caractère a. Ce nouveau facteur portera le numéro 1 dans le dictionnaire. Ensuite, aa correspond au premir facteur du dictionnaire suivi de a. On code (1,a). Ce nouveau facteur est numéroté 2. Les caractères suivants (bba...) sont inconnus dans le dictionnaire. On code le facteur vide suivi du caractère b : (0,b), et ainsi de suite...

Comme pour LZ77, LZ78 comprime les informations au fur et à mesure de leur réception. Il peut donc être implémenté "en ligne". Il faut toutefois remarquer que tout le dictionnaire est conservé en mémoire. La taille mémoire disponible doit donc être suffisante. Une alternative consiste, lorsque la mémoire est entièrement remplie, à effacer complètement le dictionnaire pour continuer la compression en construisant un nouveau dictionnaire. La compression sera donc moins bonne mais la méthode pourra être utilisée même si la mémoire ne peut pas contenir l'ensemble de tous les facteurs du texte.

Le décodage est très simple également parce qu'il suffit au décodeur de reconstruire le dictionnaire D au fur et à mesure du décodage. Dès lors, à tout moment il connaît $\#D$, il est donc capable de connaître le nombre de bits utilisés par le codage du numéro de facteur : $\lceil \log \#D \rceil$. Les numéros des facteurs sont les mêmes que pour le codage et les facteurs sont reconstruits sans problème.

2.4 Complexité de Kolmogorov

Le concept de *complexité de Kolmogorov* est la base de la *théorie algorithmique de l'information* [LV93, Del94, RDDD96].

Dans la théorie algorithmique de l'information, une séquence est considérée comme une *description* d'un objet.

Exemple 2.18

- *Une séquence nucléique est la description d'un morceau d'une molécule d'ADN.*
- *Un fichier informatique, créé à l'aide d'un programme de traitement de texte, est la description d'un texte et de sa présentation.*

Étant donné la description S d'un objet, un compresseur C tente de trouver une description S' du même objet qui soit plus courte : $|S'| < |S|$. En écrivant que S' est une description du même objet, nous exprimons le fait que la description initiale S peut être complètement reconstruite à partir de la description S' : il s'agit de compression conservative. Pour obtenir une réduction de taille, le compresseur supprime des régularités de la description initiale et les remplace par des codes. En fait, le code est une vue synthétique de la régularité : une sorte d'*explication* de la régularité.

Exemple 2.19 *La répétition d'un même symbole consécutivement dans une séquence est*

un type de régularité. *Le segment d'une séquence, constitué d'une telle répétition, est la* régularité. *On la remplace par un code constitué du symbole répété et du nombre de répétitions.*

Connaissant le type de régularité qu'un compresseur utilise, on peut étudier la présence de ces régularités dans une séquence en lui appliquant le compresseur. S'il parvient à réduire la taille, on peut conclure la présence de ce type de régularité dans la séquence. De plus, le gain de compression obtenu mesure la quantité de régularités détectées.

Dans la théorie algorithmique de l'information, tout programme qui produit une séquence est vu comme une description de la séquence.

Soit $S = TT$ une séquence composée de deux segments identiques. Considérons le programme :

print S

C'est une description de S dont la taille est légèrement plus grande que la description initiale. Considérons maintenant le programme :

répéter 2 fois (print T**)**

C'est une nouvelle description de S dont la taille est plus petite que la description initiale si T est de longueur suffisante (l'instruction de répétition engendre un certain coût).

Définition 2.18 *La* complexité de Kolmogorov *d'une description S, notée $K(S)$ est la longueur du plus court programme capable de produire S.*

Cette définition est robuste, c'est-à-dire que si on change de machine et de langage, la complexité de Kolmogorov est la même à une constante près qui est indépendante de S. Précisément, pour K relatif à une machine M et K' relatif à une machine M', $|K(S) - K'(S)| \leq C$ avec C indépendant de S. La constante C peut être vue comme le coût, évalué en nombre de pas de programme, de simulation de la machine M par la machine M' ou l'inverse.

Malheureusement, $K(S)$ n'est pas calculable : "être incompressible" est indécidable. Il n'est donc pas possible de créer un compresseur capable de rechercher la plus courte description d'un objet et d'affirmer que c'est la plus courte.

En 1966, Martin-Löf a défini la notion de séquence *aléatoire* : il s'agit d'une séquence qui réussit *tous les tests statistiques effectifs d'aléatoirité* [Del94]. L'analyse actuelle de la théorie algorithmique de l'information établit qu'**une séquence aléatoire, au sens de Martin-Löf est une séquence incompressible au sens de la complexité de Kolmogorov** [LV93].

Intuitivement, on conclut qu'une séquence incompressible est une séquence qui ne contient pas de "régularité". Il s'agit là de la définition informelle de la notion de *régularité*.

Il est important de remarquer que la notion de régularité est une question d'échelle. Ainsi, imaginons que l'on désire exploiter le type de régularité suivant : la répétition consécutive (ou en "tandem") d'un même bit, par exemple $\overbrace{00...0}^{k \text{ fois}}$. Une séquence $R = 00...0 = 0^k$ se comprime très bien : il suffit de coder le bit répété (0) et ensuite le nombre k de répétitions de manière auto-délimitée. Le tout comporte de l'ordre de $\log k$ bits. Supposons maintenant que R soit "noyé" dans une séquence S de longueur n : $S = S'RS''$. Pour exploiter la régularité de R il faut, en plus du codage précédent, coder la position de R dans S. Cela demande de l'ordre de $\log n$ bits. Il n'y a donc compression effective que si $\log n + \log k < k$.

Il est connu qu'en moyenne, une séquence aléatoire binaire de longueur n, contient une répétition 0^k de longueur $k = \log n$. D'après le processus de codage ci-dessus, le codage d'une

telle répétition nécessite $\log n + \log k = \log n + \log\log n = k + \log k$ bits. L'exploitation d'une telle régularité dans une séquence aléatoire ne produit donc pas de compression effective. Communément, on dit que la répétition 0^k n'est *pas significative* : le nombre k de répétitions est insignifiant vis-à-vis de la longueur de la séquence.

Cette propriété importante peut être démontrée pour n'importe quel type de régularité. Les séquences compressibles sont donc en quelque sorte, celles qui contiennent des régularités significatives.

Une autre notion très importante, qui lie la complexité de Kolmogorov aux probabilités, est la loi dite de *distribution a priori* des descriptions, de Solomonoff-Levin. Elle considère que les plus courtes descriptions sont les plus "probables". Si S est une description, alors la probabilité a priori que ce soit la meilleure description est $2^{-|S|}$. Cette loi fonde le principe du rasoir d'Occam qui affirme que "S'il existe plusieurs explications pour un phénomène, dès lors, dans les mêmes conditions, on choisira la plus simple" [LV93]. La notion de distribution a priori repose sur des bases théoriques profondes que nous ne développons pas ici [LV93].

En pratique, les compresseurs approchent la complexité de Kolmogorov des séquences : ils tentent de les raccourcir au maximum en exploitant certaines régularités.

Exemple 2.20 *Considérons les deux séquences $S1$ et $S2$:*

- $S1 = 10101001110011001111010101110101000010100111010100101011$
- $S2 = 10111011101110111011101110111011101110111011101110111011$

Si on tente de donner une description exacte de chacune des deux séquences, on proposera certainement :

- $S1$ *est* 10101001110011001111010101110101000010100111010100101011
- $S2$ *est* 14(1011)

La description de la séquence $S1$ semble montrer qu'elle a été engendrée par un processus aléatoire : cette description semble la plus courte. Par contre, la description de $S2$ montre qu'elle a été engendrée par un processus spécifique. La description donnée est plus courte que la description originale, dès lors elle est plus probable.

Deuxième partie

Optimisation de Courbes par Liftings

Chapitre 1

Présentation

Ce chapitre procède à une présentation de la problématique des *liftings*. Cette problématique trouve ses origines dans le contexte de la compression de séquences (section 1.1), où l'optimisation du gain de compression est un souci permanent.

L'idée générale est la suivante, les sections 1.1 et 1.2 en éclairent la lecture.

Les liftings permettent l'optimisation d'une classe précise de solutions de problèmes portant sur des données en nombres entiers. Supposons un problème décomposable par intervalles de multiples façons en sous-problèmes indépendants. Une solution précise est fournie, elle vérifie les propriétés suivantes :

- Chaque sous-problème possède une solution partielle et quelle que soit la décomposition en intervalles du problème, la solution obtenue par regroupement des solutions partielles des sous-problèmes est celle qui est fournie.

- On dispose d'un critère permettant de connaître la "qualité" d'une solution globale ou partielle.

Imaginons maintenant pour chaque sous-problème une solution alternative simple : laisser le sous-problème irrésolu ! Nous l'appelons la solution de *rupture*. A priori, cette solution est de mauvaise qualité. Il se peut malgré tout, pour certains sous-problèmes, que la solution partielle initiale soit de moins bonne qualité que cette solution de rupture (par exemple, lorsque la solution partielle n'est pas adaptée à ce sous-problème particulier). Dès lors le remplacement, pour ce sous-problème, de la solution partielle par la solution de rupture augmente la qualité de la solution globale. Ce remplacement est possible puisqu'on suppose que la solution de chaque sous-problème est indépendante des autres.

L'optimisation par liftings permet de trouver la meilleure façon de décomposer le problème en sous-problèmes et de choisir, pour chaque sous-problème, soit de conserver la solution partielle proposée, soit de la remplacer par la solution de rupture, de sorte que la qualité de la solution globale devienne optimale.

L'optimisation que nous proposons est basée sur la connaissance d'une courbe caractérisant la qualité de la solution globale fournie ainsi que les qualités des solutions intermédiaires qui la composent. Cette courbe est définie pour certaines abscisses de l'intervalle $[0, n] \subset \mathbb{N}$ et son ordonnée pour l'abscisse n évalue la qualité de la solution globale (n est la "taille" du problème). Plus grande est cette ordonnée, meilleure est la solution. Tout sous-intervalle $[i, j]$ de $[0, n]$ tel que la courbe est définie pour l'abscisse i et l'abscisse j, représente un sous-problème que l'on peut isoler des autres sous-problèmes. La qualité de la solution partielle

de ce sous-problème est évaluée par la différence d'ordonnée entre l'abscisse j et l'abscisse i. L'application de la solution de rupture pour ce sous-problème correspond au remplacement, sur $[i, j]$, de la portion de courbe par une courbe de forme déterminée. La partie de courbe située à droite de l'intervalle est alors soulevée (d'où le terme *lifting*) d'une hauteur égale au gain de qualité obtenu par le remplacement.

Grâce à des exemples fondamentaux, les sections de ce chapitre illustrent l'utilité de l'optimisation par liftings. Dans la première section, il s'agit d'applications dans le domaine de la compression de séquences. Le chapitre se termine par une brève description d'un problème analogue dans le domaine boursier, dont la seule prétention est pédagogique.

1.1 Compression

Le premier exemple, présenté dans cette section, concerne la problématique générale de compression de séquences. Le second s'intéresse plus particulièrement à la compression par codage de Huffman d'ordre 2.

1.1.1 Considérations générales

Pour réduire la taille d'une séquence, une méthode de compression procède par codage des facteurs qui la composent. Elle applique un schéma de codage déterminé sur toute la longueur de la séquence. Ce schéma n'est pas forcément adapté à tous les facteurs. Son application en raccourcit certains et en rallonge d'autres. Soit s la séquence. Notons $code(s)$ le codage de s.

On cherche à comprimer s en utilisant le codage quand il est favorable et en recopiant les facteurs tels quels quand le codage les rallonge. Prenons en abscisse les positions de la séquence où ce choix est possible (il peut y avoir des morceaux de codage qui ne peuvent être interrompus). Plaçons en ordonnée le gain en compression produit par le codage appliqué au préfixe se terminant à cette abscisse. Ce gain peut être négatif. La courbe obtenue est appelée la *courbe de compression*. Par convention, l'ordonnée dont l'abscisse est la plus grande existe. C'est le gain en compression global sur la séquence.

Un facteur est matérialisé sur la courbe de compression par une suite d'abscisses contiguës. Soit x_{min} la plus petite de ses abscisses et x_{max} la plus grande. Le facteur a été rallongé par le codage si l'ordonnée de x_{max} est inférieure à celle de x_{min}. Si les deux ordonnées sont égales, la longueur du facteur a été conservée; si celle de x_{max} est supérieure à celle de x_{min}, alors le facteur a été raccourci. La figure 1.1 montre un exemple de courbe de compression.

Décomposons la séquence en trois facteurs : $s = uvw$ tels que le codage d'un facteur soit totalement indépendant des autres : $code(s) = code(u)code(v)code(w)$. Cela signifie que les positions de début et de fin des facteurs doivent exister dans la courbe (voir figure 1.1). On appelle cette propriété la *modularité du codage*. Elle devra toujours être vérifiée pour notre algorithme d'optimisation.

Supposons que v soit rallongé par le codage : $|code(v)| > |v|$. Envisageons de ne plus coder le facteur v mais de le recopier tel quel dans la séquence comprimée. Cette action est appelée une *rupture* du codage. Le gain de compression global de la séquence s'en trouvera augmenté.

Idéalement, cette opération remplace le morceau de courbe correspondant à v par une série de points situés sur une ligne horizontale (voir figure 1.2). Toute la partie de courbe située à droite du codage du facteur v est alors soulevée d'une hauteur égale au gain engendré par la rupture. Ce soulèvement est appelé un *lifting* de la courbe.

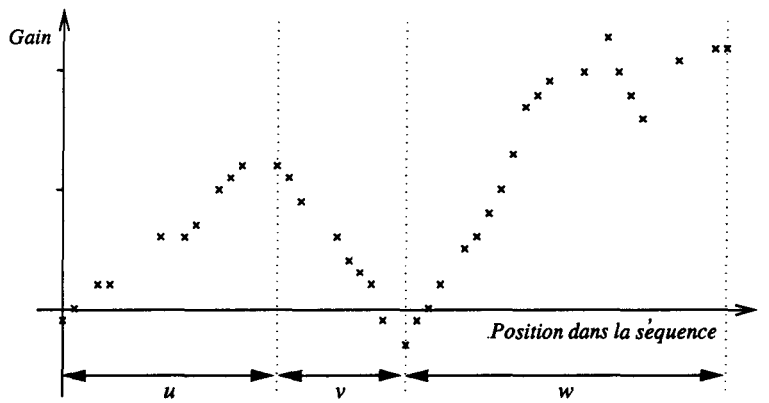


FIG. 1.1 - *Exemple de courbe de compression*

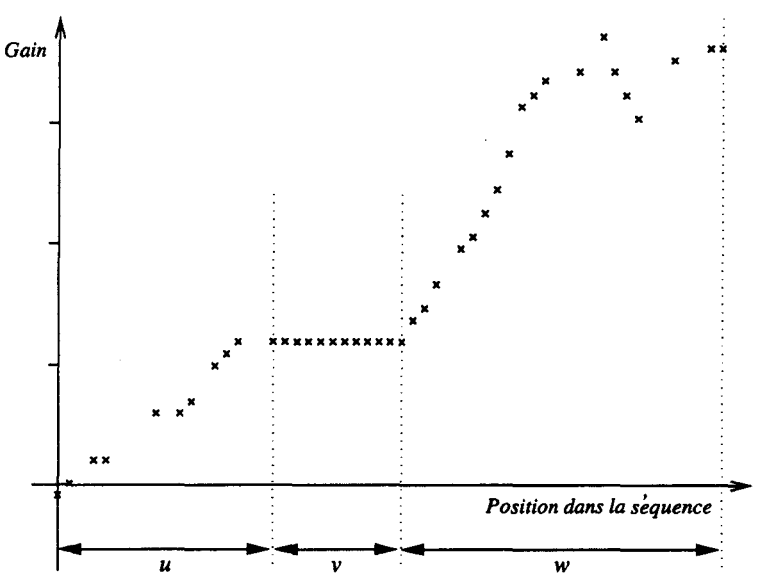


FIG. 1.2 - *Lifting idéal pour la courbe de la figure 1.1*

En pratique, on ne peut pas se contenter de recopier le facteur v tel quel, il faut le délimiter. Il y a deux méthodes classiques de codage pour délimiter un facteur : la *ponctuation* et l'auto-délimitation de la longueur (voir partie I, chapitre 2). La ponctuation est la plus courante (il s'agit de faire suivre un facteur par un mot réservé, que l'on symbolise souvent par "." ou ";" et qui n'apparaît jamais comme préfixe d'un mot code $code(m)$), mais il est facile de montrer qu'elle est moins économe que l'auto-délimitation, qui consiste à déclarer de façon préfixe, la longueur de la séquence. Nous notons $AD(|x|)$ le codage préfixe de la longueur de x.

Exemple 1.1 *Soit x, x', x'' les trois facteurs à coder l'un après l'autre. Les deux méthodes de délimitation sont :*

Ponctuation :	$x; x'; x''$
Auto-délimitation de la longueur :	$AD(\|x\|)xAD(\|x'\|)x'AD(\|x''\|)x''$

Nous allons dans la suite choisir l'auto-délimitation pour déterminer la longueur des facteurs non codés (c'est-à-dire pour déterminer la longueur des ruptures). Pour délimiter les fins de facteurs codés (c'est-à-dire les débuts de ruptures) nous allons choisir la ponctuation, que nous appelerons *annonce de rupture* et noterons $a_{\mathcal{R}}$.

Il est important de comprendre la motivation de ces choix pour se convaincre de la pertinence du modèle : l'auto-délimitation donne un codage plus compact, mais il ne peut être utilisé pour les parties codées, car il ferait perdre la modularité du codage (dans la pratique, nos exemples montreront que la modularité implique des contraintes au code qui font que la ponctuation n'est pas plus coûteuse).

On remplace donc $code(u)code(v)code(w)$ par $code(u)a_{\mathcal{R}}AD(|v|)v\,code(w)$. Le *gain du lifting* est $|code(v)| - (|a_{\mathcal{R}}| + |AD(|v|)| + |v|)$. Toute la partie de courbe située à droite du facteur v est soulevée de cette hauteur (voir figure 1.3). Convenons, pour chaque abscisse le long de la rupture, de placer en ordonnée le gain obtenu si c'était la position de fin de rupture. Puisque le coût de l'auto-délimitation $AD(l)$ d'une longueur l est en $\theta(\log l)$ (voir partie I, chapitre 2), cette *courbe de rupture* a la forme d'un "logarithme retourné" (voir figure 1.3).

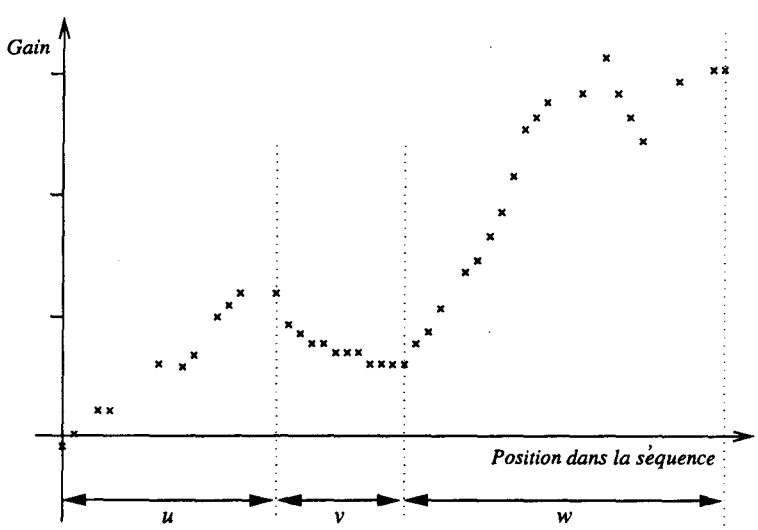


FIG. 1.3 - *Lifting pour la courbe de la figure 1.1*

L'optimisation du gain de compression global d'une séquence ne peut pas se faire par applications aveugles de ruptures sur chaque portion décroissante de la courbe. En effet, la

courbe de rupture est elle même décroissante. Elle peut apporter une perte supplémentaire si elle est appliquée à certaines portions faiblement décroissantes. De plus, il faut parfois choisir entre plusieurs ruptures possibles et appliquer la meilleure.

L'algorithme d'optimisation que nous proposons impose une forme particulière à la courbe de rupture : elles doit être *DCL* (*Décroissante, Convexe et de "décroissance Limitée"*, voir chapitre 2, section 2.1). Cela signifie que la courbe doit être décroissante, que les marches doivent être de plus en plus longues et que la hauteur de toutes les marches (sauf éventuellement la première) doit être égale à 1. C'est une propriété naturelle dans la mesure où cela correspond à la discrétisation de l'opposé d'une courbe logarithmique. Si le code *AD* utilisé pour coder les longueurs est ICL (voir partie I, chapitre 2), alors la courbe de rupture possède la propriété DCL.

1.1.2 Codage de Huffman

Nous détaillons ici l'exemple de la section 1.1.1 appliqué à une compression par codage de Huffman d'ordre 2. Cet exemple sera développé plus formellement dans le chapitre 1 de la partie III.

Soit $\mathcal{B}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ l'ensemble des mots de longueur 3 sur l'alphabet binaire $\mathcal{B}$. Pour plus de facilités, notons :

$$\begin{array}{llll} a = 000 & c = 010 & e = 100 & g = 110 \\ b = 001 & d = 011 & f = 101 & h = 111 \end{array}$$

Considérons le codage de Huffman sur $\mathcal{B}^3$, décrit par l'arbre de la figure 1.4. Notons

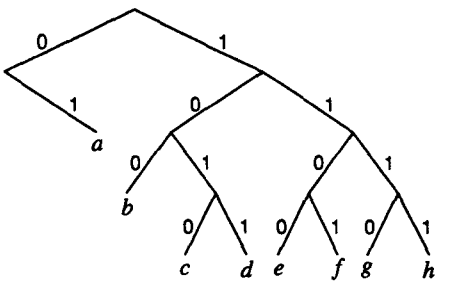


FIG. 1.4 - *Arbre de Huffman sur $\mathcal{B}^3$ qui exploite la richesse en a*

$Huff(s)$ le résultat du codage de la séquence binaire s. C'est un codage qui exploite la richesse en a.

Remarque 1.1 *L'arbre n'est pas complet, il lui manque la branche correspondant au mot code* 00. *Il s'agit du mot code $a_{\mathcal{R}}$ dont on va se servir pour annoncer une rupture. Il n'est préfixe d'aucun autre mot code. Cette nécessité sera discutée plus en profondeur à la section 1.3 du chapitre 1 de la partie III.*

Soit $s = aaaabaaabbaaabaaaabbaahgfecdbhgecfdfghcaabacaabaaabaaaba$ une séquence de 168 bits. Le codage $Huff(s)$ contient 157 bits, sa courbe de compression est représentée à la figure 1.5. La courbe est définie une abscisse sur trois car il est possible d'interrompre le codage entre deux mots code uniquement. Décomposons s en trois facteurs : $s = uvw$ avec $u = s_{..66}$, $v = s_{67..117}$ et $w = s_{118..}$. Le facteur v est la partie soulignée de s. On a $Huff(s) =$

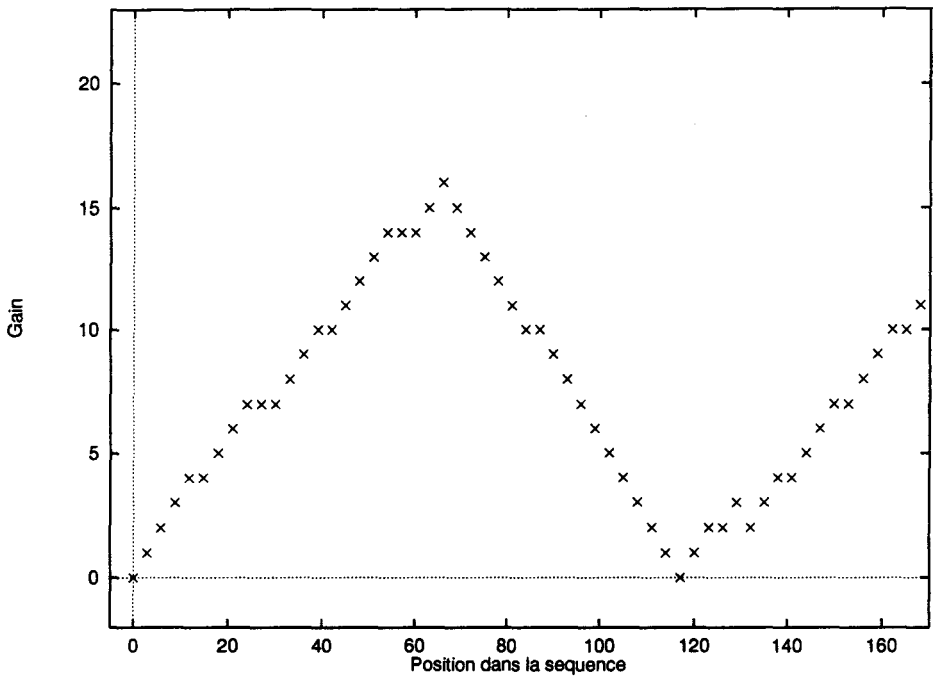


FIG. 1.5 - *Courbe de compression de Huff(s)*

$Huff(u)Huff(v)Huff(w)$. La partie $Huff(v)$ engendre une grande perte dans la compression. Il s'agit de la longue portion décroissante sur la courbe de la figure 1.5. Remplaçons la par une rupture en utilisant le code *Fibo* pour auto-délimiter la longueur (voir sa définition, partie I, chapitre 2). On obtient la séquence comprimée $Huff(u)00Fibo(51)v\ Huff(w)$ qui contient 152 bits, soit 5 bits de moins que la version $Huff(s)$ (voir figure 1.6).

1.2 Stratégie optimale dans le milieu boursier

Un cas particulier de courbe de rupture DCL est une courbe en "L", où le coût est constant, quelle que soit la longueur de la rupture. Notre algorithme prend en compte ce cas mais il est très simple et peut être traité efficacement par une méthode directe.

Considérons le milieu boursier et intéressons-nous particulièrement à une action que nous supposons suivre l'indice Dow Jones, et que nous possédons. Plaçons en abscisse les jours et en ordonnée la valeur de l'indice Dow Jones du jour. Cette ordonnée n'est définie que les jours ouvrables.

La signification d'une rupture est la revente de l'action: lorsque sa valeur descend, il faut vendre et lorsqu'elle remonte, il faut acheter. En réalité, le choix est plus subtil car la revente et le rachat d'une action entraînent des frais supplémentaires. La forme de la courbe de rupture est en L car le coût est indépendant du temps qui sépare la revente du rachat. La hauteur du L est égale au montant des frais imputés à la revente et à l'achat.

La figure 1.7 présente l'effet d'un lifting sur la courbe de l'indice Dow Jones.

L'optimisation par liftings correspond à une stratégie optimale de vente et d'achat. Chaque lifting augmente le bénéfice de gestion de l'action.

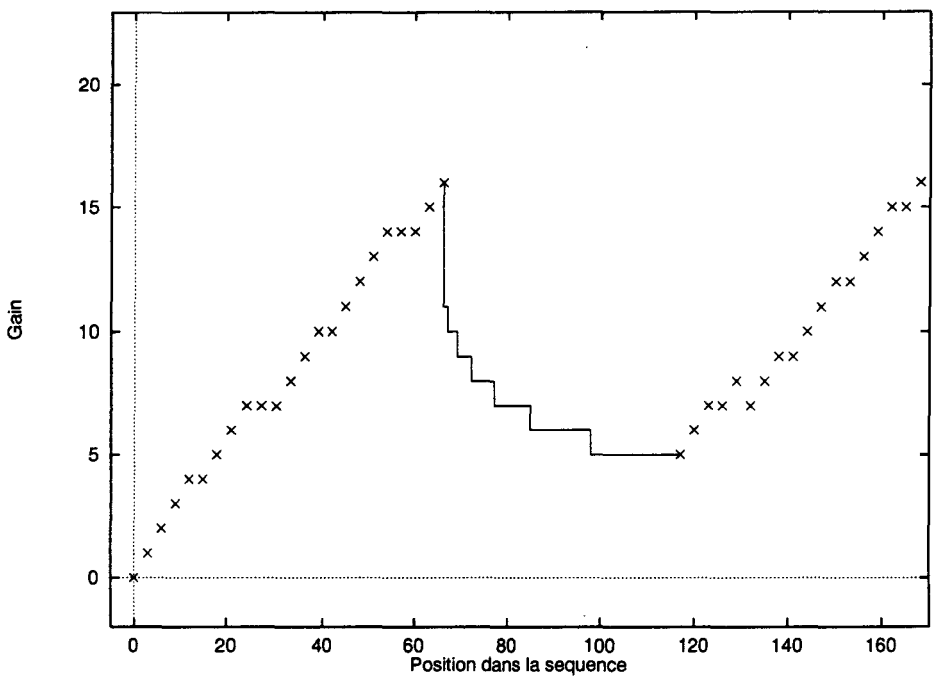


FIG. 1.6 - *Lifting sur la courbe de la figure 1.5*

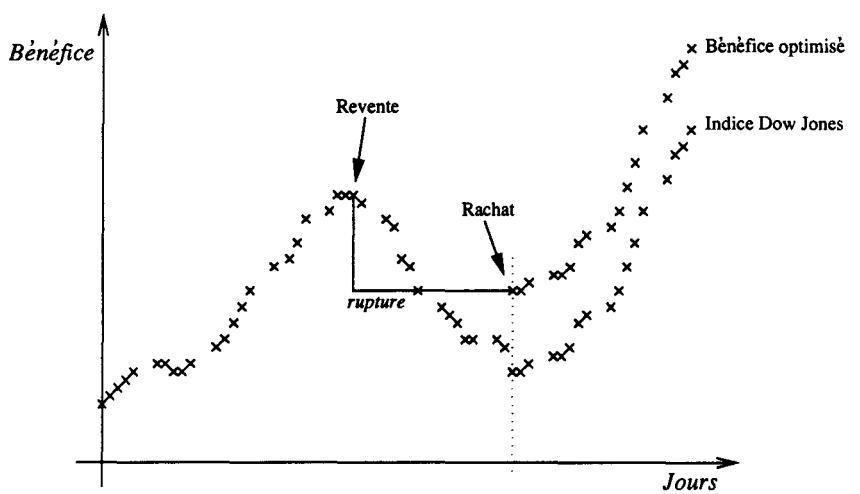


FIG. 1.7 - *Lifting par courbe de rupture en* L

Chapitre 2

Formalisation du problème

Le problème d'*optimisation d'une courbe par liftings* est présenté de manière formelle dans ce chapitre. Il est exposé de façon générale en ne faisant référence à aucune de ses applications potentielles ni à la méthode de résolution développée au chapitre suivant.

Il s'agit d'un problème mathématique basé sur la manipulation de courbes à l'aide d'outils appelés *ruptures*. Le but est de maximiser la valeur de la courbe pour son abscisse maximale. L'application d'une rupture remplace une portion de la courbe par un morceau de courbe de forme déterminée, appelée *courbe de rupture*. Toute la portion de courbe située à droite du remplacement est translatée verticalement, ce qui provoque une hausse ou une baisse de la valeur de la courbe à son abscisse maximale.

Grâce au concept de fonction DCL (Décroissante, Convexe et de "décroissance Limitée"), nous imposons une forme particulière à la courbe de rupture. Les fonctions DCL sont intimement liées aux codes ICL définis dans la première partie de ce travail. Nous expliquons le lien qui les unit.

La dernière section de ce chapitre illustre la difficulté de résolution du problème. Des approches de type "force brute" ou heuristiques sont envisagées et leur inadéquation pour la résolution du problème d'optimisation est mise en évidence.

2.1 Fonctions DCL

Cette section constitue un préambule à l'énoncé formel du problème d'optimisation que l'on se pose. Elle définit le concept de fonction DCL. C'est un concept très important, l'efficacité de notre algorithme d'optimisation repose entièrement sur l'utilisation d'une telle fonction comme fonction de rupture.

Définition 2.1 *Une fonction* $f : \mathbb{N} \to \mathbb{Z}$ *est* DCL *(pour Décroissante, Convexe et de "décroissance Limitée") si les conditions suivantes sont vérifiées.*

1. f *est partout définie.*

2. f *est décroissante :* $\forall i \in \mathbb{N}, f(i) \geq f(i+1)$.

3. $f(0) = 0$.

4. *Il existe* $r > 0$ *tel que la fonction est constante et inférieure à* -1 *sur* $[1, r[$*, est de "décroissance limitée" à 1 à partir de* r *et est convexe également à partir de* r*, c'est-à-dire :*

   - $\forall i : 0 < i < r : f(i) = f(1) < -1$ *: f est constante et inférieure à* -1 *sur* $[1, r[$.
   - $\forall i \geq r : f(i) \leq f(i+1) + 1$ *: f est de "décroissance limitée" à 1 à partir de* r.
   - *Soient* $j, j', k, k' : r \leq j < j' \leq k < k'$ *tels que (voir figure 2.1) :*

$$\begin{cases} & f(j-1) & = & f(j)+1 \\ f(j) & = f(j'-1) & = & f(j')+1 \\ & f(k-1) & = & f(k)+1 \\ f(k) & = f(k'-1) & = & f(k')+1 \end{cases}$$

   Alors $(k' - k) \geq (j' - j)$.
   La fonction f est en "escalier", l'abscisse j est le début d'une marche et j' est le début de la marche suivante. De la même manière, k et k' sont les débuts de deux marches successives situées à droite de la marche $[j, j'[$ *(voir figure 2.1).*

Cette dernière condition exprime que la fonction est convexe à partir de r : les marches de la représentation de f sont de plus en plus longues au sens large, lorsque l'on s'éloigne de l'origine.

La constante r est appelée le retard *de la fonction f.*

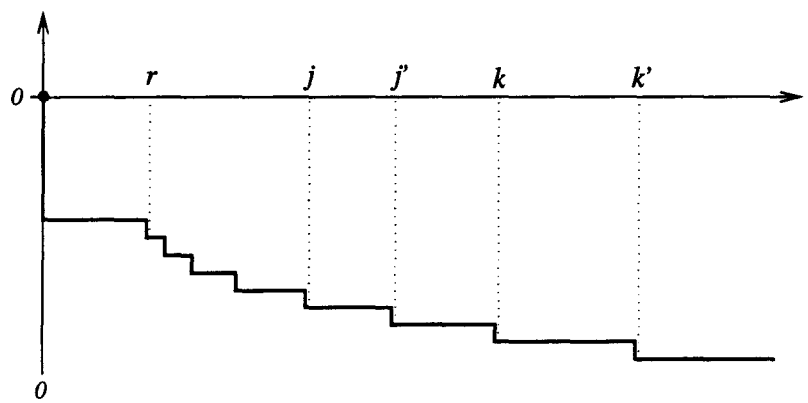


FIG. 2.1 - *Représentation d'une fonction DCL*

À l'exception de la première marche, cette définition correspond à la translation à l'origine de la discrétisation d'une courbe convexe décroissante. Cette première marche est une sorte d'initialisation d'un coût. L'utilité de lui imposer une hauteur supérieure à 1 apparaîtra dans le chapitre suivant lors de la définition 3.1 de l'ordre $<_i$ entre deux courbes de rupture. Elle permet d'assurer que $<_i$ est un ordre total (proposition 3.1). La convexité de la fonction traduit que l'augmentation du coût s'amenuise lorsque l'on s'éloigne de l'origine.

Présentons quelques exemples de fonctions DCL.

Exemple 2.1 *La fonction f_1 définie par :*

$$\begin{cases} f_1(0) &= 0 \\ f_1(i) &= -C & \forall i : 0 < i < r_1 \quad \textit{avec } C > 1 \textit{ une constante} \\ f_1(i) &= -C - (i - r_1 + 1) & \forall i \geq r_1 \end{cases}$$

une fonction DCL dont toutes les marches ont la même longueur à partir du retard r_1.

Exemple 2.2 *La fonction f_2 définie par :*

$$\begin{cases} f_2(0) &= 0 \\ f_2(i) &= -C \quad \forall i > 0 \quad \textit{avec } C > 1 \textit{ une constante} \end{cases}$$

est une fonction DCL en "L" : elle ne contient qu'une seule marche.

Exemple 2.3 *La fonction f_3 définie par :*

$$\begin{cases} f_3(0) &= 0 \\ f_3(i) &= -C & \forall i : 0 < i < r_3 \quad \textit{avec } C > 1 \textit{ une constante} \\ f_3(i) &= -C - \lfloor \sqrt{i - r_3 + 1} \rfloor & \forall i \geq r_3 \end{cases}$$

est une fonction DCL de retard r_3, ses marches sont de plus en plus longues au sens strict.

Toute fonction logarithmique peut être utilisée à la place de la racine carrée de l'exemple 2.3.

Exemple 2.4 *La fonction f_4 définie par :*

$$\begin{cases} f_4(0) &= 0 \\ f_4(i) &= -C & \forall i : 0 < i < r_4 \quad \textit{avec } C > 1 \textit{ une constante} \\ f_4(i) &= -C - \lfloor \log(i - r_4 + 2) \rfloor & \forall i \geq r_4 \end{cases}$$

est une fonction DCL de retard r_4.

Les fonctions DCL sont intimement liées aux longueurs des mots code des codes ICL (voir partie I, chapitre 2). La proposition 2.1 construit une fonction DCL à partir des longueurs des mots code d'un code ICL.

Proposition 2.1 *Soit $I : \mathbb{N} \to \mathcal{B}^*$ un code ICL. Alors la fonction f_I définie par :*

$$\begin{cases} f_I(0) &= 0 \\ f_I(i) &= -C - |f_I(0)| & \forall i : 0 < i < r_I \quad \textit{avec } C > 1 \textit{ une constante} \\ f_I(i) &= -C - |f_I(i - r_I)| & \forall i \geq r_I \end{cases}$$

est une fonction DCL de retard r_I.

Preuve. Puisque I est ICL, $|I(i)|$ est croissante, concave et son accroissement est limité à 1 (définition d'un code ICL). Dès lors, $f_I(i) = -C - |f_I(i - r_I)|$ est décroissante, convexe et de décroissance limitée à 1 à partir du retard r_I. La soustraction de $|f_I(0)|$ lorsque $i < r_I$ évite d'avoir un saut de plus de 1 bit entre $r_I - 1$ et r_I lorsque $f_I(0)$ comporte plus de 1 bit. □

Le codage d'entiers à l'aide d'un code tel que BIN, $Fibo$ ou $PrefFibo$ engendre donc une fonction de coût de codage qui est DCL.

Exemple 2.5 *La fonction f_{Fibo} définie par:*

$$\begin{cases} f_{Fibo}(0) &= 0 \\ f_{Fibo}(i) &= -C - |Fibo(0)| \quad \forall i : 0 < i < r \quad \textit{avec } C > 1 \textit{ une constante} \\ f_{Fibo}(i) &= -C - |Fibo(i-r)| \quad \forall i \geq r \end{cases}$$

est une fonction DCL de retard r.

2.2 Énoncé formel

Nous définissons le problème mathématique d'optimisation par liftings dans un contexte tout à fait général. Commençons par les définitions des objets qui entrent en jeu.

L'objet à optimiser est une *application partielle* (ou *courbe partielle*) $f : [0,n] \subset \mathbb{N} \to \mathbb{Z}$.

Définition 2.2 *Soit $\mathcal{F}^n$, la* famille d'applications partielles *de $[0,n] \subset \mathbb{N}$ à valeurs dans $\mathbb{Z}$: $f \in \mathcal{F}^n \iff f : [0,n] \subset \mathbb{N} \to \mathbb{Z}$. Pour certaines abscisses $i \in [0,n]$, $f(i)$ peut ne pas être défini. Par convention, $f(0)$ et $f(n)$ sont définis.*

L'outil d'optimisation est la *rupture*.

Définition 2.3 *Soit $\mathcal{R}$ la* fonction de rupture *. C'est une fonction DCL, sa représentation est appelée la* courbe de rupture *ou plus simplement la* rupture. *Le* retard *de $\mathcal{R}$ est noté $r_{\mathcal{R}}$.*

Définition 2.4 *Soit $\mathcal{F}^n_{\mathcal{R}}$ la* famille d'applications partielles avec ruptures *de $[0,n] \subset \mathbb{N}$ à valeurs dans $\mathbb{Z}$. C'est-à-dire que $f \in \mathcal{F}^n_{\mathcal{R}} \iff [0,n]$ peut être décomposé en intervalles disjoints $[x,y]$ tels que la portion de courbe de f sur $[x,y]$ est de l'un des deux types suivants:*

- *Une* portion d'application*: c'est la courbe d'une application partielle de $[x,y]$ dans $\mathbb{Z}$. Aucun point de la portion n'est sur une courbe de rupture.*
- *Une* portion de rupture*: $\forall i \in [x,y] : f(i) = k + \mathcal{R}(i-x)$, avec $k \in \mathbb{Z}$.*

Par définition, $\mathcal{F}^n \subset \mathcal{F}^n_{\mathcal{R}}$.

La figure 2.2 représente une application partielle avec rupture $f \in \mathcal{F}^n_{\mathcal{R}}$. Les portions de la

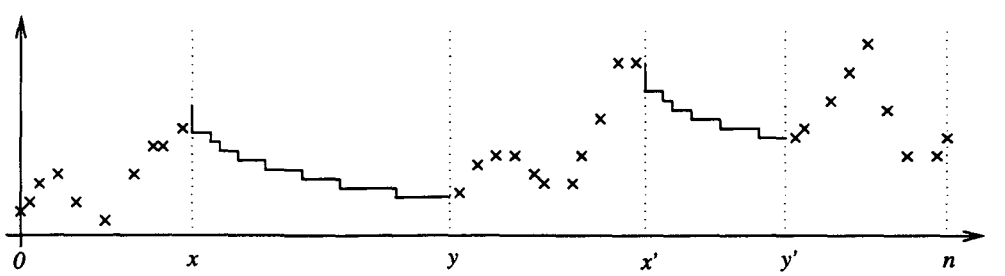


FIG. 2.2 - *Courbe d'une fonction partielle avec rupture $f \in \mathcal{F}^n_{\mathcal{R}}$*

courbe sur les intervalles $[0, x-1], [y+1, x'-1]$ et $[y'+1, n]$ sont des portions d'application. Les portions de la courbe sur les intervalles $[x, y]$ et $[x', y']$ sont des portions de rupture.

Remarque 2.1 *Soit $f \in \mathcal{F}^n_{\mathcal{R}}$. Par convention, la notation $f(i)$ sous entend que f est définie pour l'abscisse i. Nous n'accédons de cette façon qu'aux abscisses i pour lesquelles $f(i)$ existe. En pratique, il est nécessaire avant chaque accès à $f(i)$ de tester que l'application f est définie en i. Si ce n'est pas le cas, une autre abscisse doit être considérée.*

Définissons l'opération de *lifting* qui permet d'introduire ou de modifier une portion de rupture dans la courbe d'une application $f \in \mathcal{F}^n_{\mathcal{R}}$.

Définition 2.5 *Soit $f \in \mathcal{F}^n_{\mathcal{R}}$ une application partielle avec rupture. Soit $[x, y] \subseteq [0, n]$ tel que $f(x)$ et $f(y)$ soient définis.*

Notons $f \xrightarrow{\mathcal{R}_{[x,y]}} \overline{f}$ le lifting *qui transforme f en $\overline{f} \in \mathcal{F}^n_{\mathcal{R}}$, définie par:*

- $\forall i \in [0, x] : \overline{f}(i) = f(i)$.

Pour les abscisses $i \geq x$, deux cas sont à distinguer:

1er cas: *y est sur une portion d'application de f. On doit avoir $y - x \geq r_{\mathcal{R}}$. On applique alors la nouvelle rupture sur $[x, y]$:*

- $\forall i \in [x, y] : \overline{f}(i) = f(x) + \mathcal{R}(i - x)$
- $\forall i \in [y, n] : \overline{f}(i) = f(i) + f(x) - f(y) - \mathcal{R}(y - x)$

L'intervalle $[x, y]$ est appelé le domaine *du lifting.*

2ème cas: *y est sur une portion de rupture $[x', y']$ de f. On applique la nouvelle rupture sur $[x, y']$ plutôt que sur $[x, y]$:*

- $\forall i \in [x, y'] : \overline{f}(i) = f(x) + \mathcal{R}(i - x)$
- $\forall i \in [y', n] : \overline{f}(i) = f(i) + f(x) - f(y') - \mathcal{R}(y' - x)$

L'intervalle $[x, y']$ est appelé le domaine (étendu) *du lifting.*

La figure 2.3 présente le lifting $f \xrightarrow{\mathcal{R}_{[x,y]}} \overline{f}$ dans le cas où y appartient à une portion d'application de f. Sur $[0, x]$ la courbe est inchangée. Sur $[x, y]$, on place la courbe de rupture. Au-delà de y, toute la partie de la courbe est soulevée d'une hauteur $f(x) - f(y) - \mathcal{R}(y - x)$. Cette quantité est appelée le *gain du lifting*.

La raison de l'extension du domaine $[x, y]$ à $[x, y']$ lorsque y appartient à la portion de rupture $[x', y']$ de f provient de la définition d'une application partielle avec rupture: on ne peut pas couper une portion de rupture en deux et ne conserver que sa partie droite. La première marche d'une rupture doit toujours figurer dans la portion de rupture. On ne peut donc pas couper la portion $[x', y']$ de rupture en deux et ne conserver que le morceau $[y, y']$. Dès lors, on fait disparaître la portion de rupture $[x', y']$ en la "recouvrant" par $[x, y']$ (voir figure 2.4).

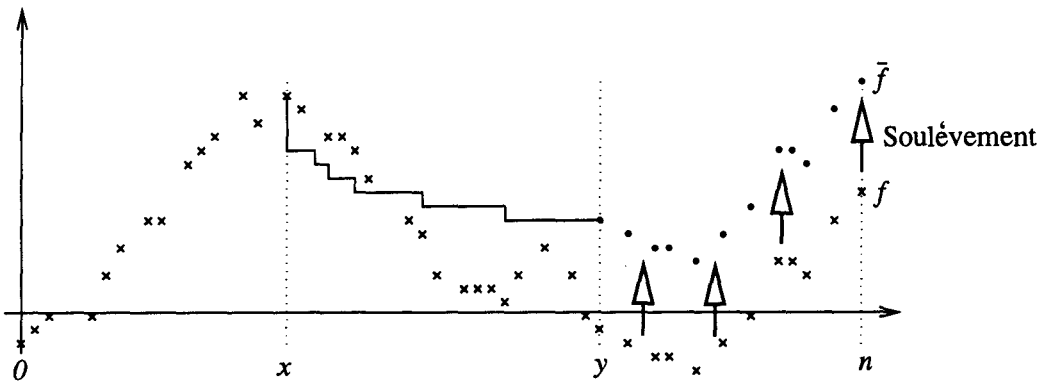


FIG. 2.3 - *Exemple de lifting* $f \xrightarrow{\mathcal{R}_{[x,y]}} \overline{f}$

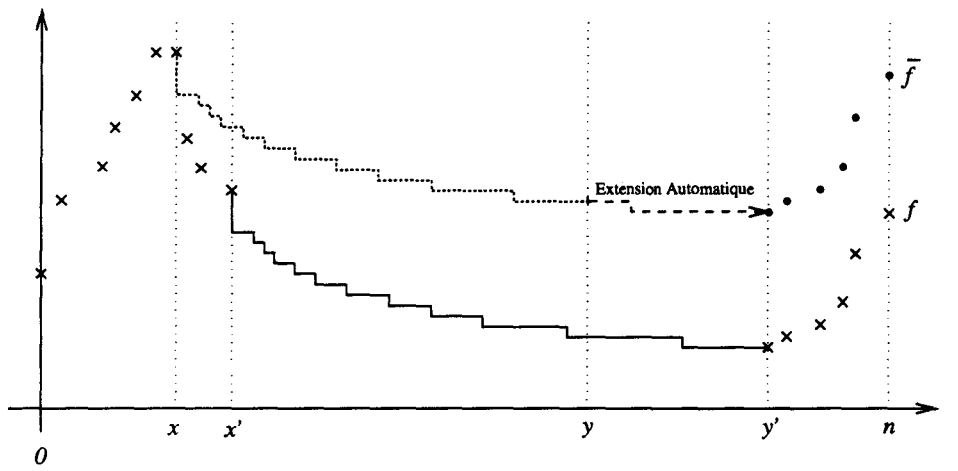


FIG. 2.4 - *Extension automatique d'une rupture*

Étant donné une application partielle avec rupture f, on définit l'ensemble des applications partielles avec ruptures qui sont obtenues par 0 ou 1 lifting de f.

Définition 2.6 *Soit* $f, \overline{f} \in \mathcal{F}^n_{\mathcal{R}}$.

$$\overline{f} \in Lift_{\mathcal{R}}(f) \iff (\overline{f} = f) \text{ ou } (\exists [x,y] \subseteq [0,n] : f \overset{\mathcal{R}_{[x,y]}}{\longrightarrow} \overline{f})$$

Définissons $Lift^*_{\mathcal{R}}(f)$ la clôture transitive de $Lift_{\mathcal{R}}(f)$. C'est l'ensemble des applications partielles avec ruptures qui sont obtenues par itération de liftings à partir de la courbe f.

Définition 2.7

$$\overline{f} \in Lift^*_{\mathcal{R}}(f) \iff (\overline{f} \in Lift_{\mathcal{R}}(f)) \text{ ou } (\overline{f} \in Lift_{\mathcal{R}}(f') \text{ avec } f' \in Lift^*_{\mathcal{R}}(f))$$

Nous pouvons maintenant définir précisément le problème d'optimisation.

Problème 2.1 *Étant donnés* $f \in \mathcal{F}^n$ *et* $\mathcal{R}$ *une fonction DCL de retard* $r_{\mathcal{R}}$, *trouver* $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ *tel que* $\overline{f}(n)$ *soit maximal.*

2.3 Difficulté du problème d'optimisation

Considérons le problème 2.1 présenté à la section précédente : étant donnés $f \in \mathcal{F}^n$ et $\mathcal{R}$ une fonction DCL de retard $r_{\mathcal{R}}$, trouver $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ tel que $\overline{f}(n)$ soit maximal. Dans cette section, nous mettons en évidence la difficulté du problème d'une part en montrant que la cardinalité de $\#Lift^*_{\mathcal{R}}(f)$ est exponentielle par rapport à n et, d'autre part, que les heuristiques simples peuvent manquer la solution optimale.

2.3.1 Cardinalité de $Lift^*_{\mathcal{R}}(f)$

Imaginons un algorithme en force brute qui engendre toutes les applications $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ et qui choisit celle qui maximise $\overline{f}(n)$. Montrons qu'un tel algorithme est inutilisable parce que $\#Lift^*_{\mathcal{R}}(f) = O(C^n)$, avec C une constante.

Étant donnée l'application partielle $f \in \mathcal{F}^n$ et la fonction $\mathcal{R}$ de rupture, une application $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ est entièrement déterminée par la connaissance de la décomposition de $[0,n]$ en sous-intervalles disjoints tels que chaque sous-intervalle soit de type portion d'application ou portion de rupture. Remarquons que plusieurs décompositions donnent la même application $\overline{f}$: si une décomposition contient deux sous-intervalles $[x,y]$ et $[y+1,z]$ consécutifs de portions d'applications, alors la réunion de $[x,y]$ et $[y+1,z]$, en un seul sous-intervalle $[x,z]$ de portion d'application, caractérise la même application $\overline{f}$. Par contre, deux intervalles consécutifs de portion de rupture ne peuvent pas être réunis en une seule portion de rupture car cela désigne une application différente.

Dès lors, $\#Lift^*_{\mathcal{R}}(f)$ est majoré par le nombre de décompositions possibles de $[0,n]$ en sous-intervalles consécutifs de type application ou de type rupture de telle sorte que deux sous-intervalles de type application soient toujours séparés par au moins un sous-intervalle de type rupture. La majoration provient d'une part du fait que f n'est pas définie pour tout

$i \in [0, n]$ et donc qu'une rupture ne peut pas forcément débuter ou se terminer à n'importe quelle abscisse. D'autre part, une portion de rupture doit être de largeur supérieure ou égale au retard $r_{\mathcal{R}}$.

Le théorème 2.2 calcule la cardinalité de $Lift^*_{\mathcal{R}}(f)$ dans le pire des cas. Elle est exponentielle.

Théorème 2.2 $\#Lift^*_{\mathcal{R}}(f) = O(\phi^{2n})$ *avec* $\phi = \frac{1+\sqrt{5}}{2}$.

Preuve. Pour faciliter les notations, chaque sous-intervalle peut être de deux types: A (Application) ou R (Rupture).

Soit $NA(i)$ le nombre de décompositions possibles d'un sous-intervalle de longueur i, en sous-intervalles consécutifs tels que:

- Le premier sous-intervalle soit de type A.
- Deux sous-intervalles de type A ne se suivent jamais.

Soit $NR(i)$ le nombre de décompositions possibles d'un sous-intervalle de longueur i, en sous-intervalles consécutifs tels que:

- Le premier sous-intervalle soit de type R.
- Deux sous-intervalles de type A ne se suivent jamais.

Dès lors, $N(i) = NA(i) + NR(i)$ est le nombre de décompositions possibles d'un sous-intervalle de longueur i, en sous-intervalles consécutifs tels que deux sous-intervalles de type A ne se suivent jamais.

Calculons $NA(i)$: la décomposition commence par un sous-intervalle de type A. Le second sous-intervalle doit donc être de type R. Pour chaque longueur possible du premier sous-intervalle (de 1 à i), le nombre de décompositions est donné par le nombre de décompositions commençant par un sous-intervalle de type R sur la longueur i diminuée de la longueur du premier sous-intervalle.

Longueur du premier sous-intervalle	Nombre de décompositions commençant par A
1	$NR(i-1)$
2	$NR(i-2)$
3	$NR(i-3)$
...	...
$i-1$	$NR(1)$
i	1

$$NA(i) = NR(i-1) + NR(i-2) + \ldots NR(1) + 1 \tag{2.1}$$

Suivons le même raisonnement pour $NR(i)$: la décomposition commence par un sous-intervalle de type R, le second peut donc être de n'importe quel type.

Longueur du premier sous-intervalle	Nombre de décompositions commençant par R
1	$NA(i-1)+NR(i-1)$
2	$NA(i-2)+NR(i-2)$
3	$NA(i-3)+NR(i-3)$
...	...
$i-1$	$NA(1)+NR(1)$
i	1

$$NR(i) = NR(i-1)+NR(i-2)+\ldots NR(1)+1 \\ +NA(i-1)+NA(i-2)+\ldots NA(1) \quad (2.2)$$

Les équations 2.1 et 2.2 entraînent

$$NR(i) = NA(i)+NA(i-1)+\ldots+NA(1) \quad (2.3)$$

Dès lors:

$$\text{Équation 2.1} \Rightarrow NA(i) = NA(i-1)+NR(i-1) \quad (2.4)$$
$$\text{Équation 2.3} \Rightarrow NR(i) = NA(i)+NR(i-1) \quad (2.5)$$

Donc $N(i-1) = NA(i)$.

Par définition, $NA(1) = NR(1) = 1$. Dès lors grâce aux équations 2.4 et 2.5, la suite:

$$NA(1), NR(1), N(1), NR(2), N(2), NR(3), N(4), NR(4), \ldots N(i-1), NR(i)\ldots$$

décrit la suite des nombres de Fibonacci.

La quantité $N(i)$ est donc égale au $2i+1^{\text{ème}}$ nombre de Fibonacci: F_{2i+1} (la définition des nombres de Fibonacci est donnée à la section 2.2.4.2.2 de la partie I). Pour rappel, F_j est égal à $\frac{\phi^j}{\sqrt{5}}$ arrondi à l'entier le plus proche, avec $\phi = \frac{1+\sqrt{5}}{2}$ le nombre d'or [Knu73a].

L'intervalle $[0, n]$ étant de longueur $n+1$ et puisqu'une rupture ne peut pas forcément commencer ou se terminer à n'importe quelle abscisse, $N(n+1)$ est une majoration de $\#Lift^*_{\mathcal{R}}(f)$. Il s'ensuit que $\#Lift^*_{\mathcal{R}}(f) = O\left(\frac{\phi^{2n+3}}{\sqrt{5}}\right) = O(\phi^{2n})$. □

Il semblait évident que la cardinalité de $Lift^*_{\mathcal{R}}(f)$ était exponentielle mais le théorème 2.2 permet de préciser l'ordre de grandeur de la borne.

La génération exhaustive de toutes les applications partielles de $Lift^*_{\mathcal{R}}(f)$ devient très vite infaisable. Prenons par exemple $n = 1000$. La quantité $\frac{\phi^{2n+3}}{\sqrt{5}}$ vaut de l'ordre de 10^{418}.

2.3.2 Heuristique d'application de ruptures

Considérons l'heuristique consistant à appliquer systématiquement des ruptures sur les portions décroissantes de la courbe. En raison de la forme particulière des courbes de rupture, elle ne trouvera pas toujours la courbe optimale.

En effet, la courbe de rupture est elle même décroissante. Son application aveugle sur une portion de courbe faiblement décroissante ou trop courte peut engendrer une diminution de la valeur de l'application pour l'ordonnée n plutôt qu'une augmentation.

D'autre part, la convexité des fonctions DCL peut faire préférer l'application d'une longue rupture, sur une portion de courbe contenant des portions croissantes, à plusieurs courtes ruptures sur les parties décroissantes uniquement. Ainsi, si la courbe optimale de la figure 2.5 est obtenue par application aveugle de ruptures sur les parties décroissantes de la courbe, ce n'est pas le cas de celle de la figure 2.6 obtenue par application d'une seule rupture.

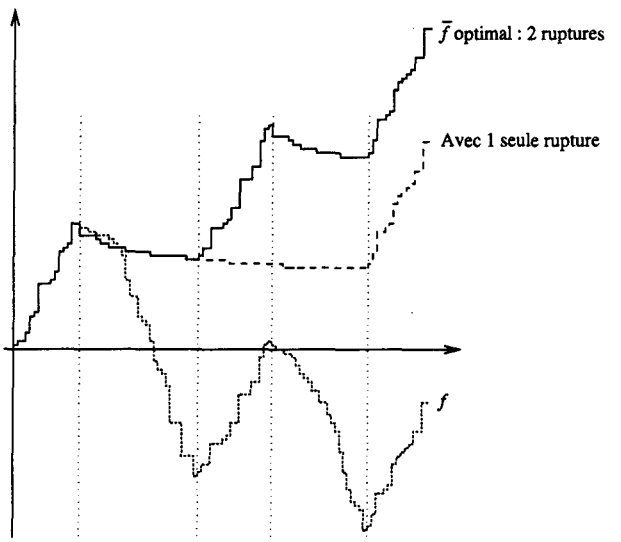


FIG. 2.5 - *Exemple de courbe optimale contenant deux ruptures*

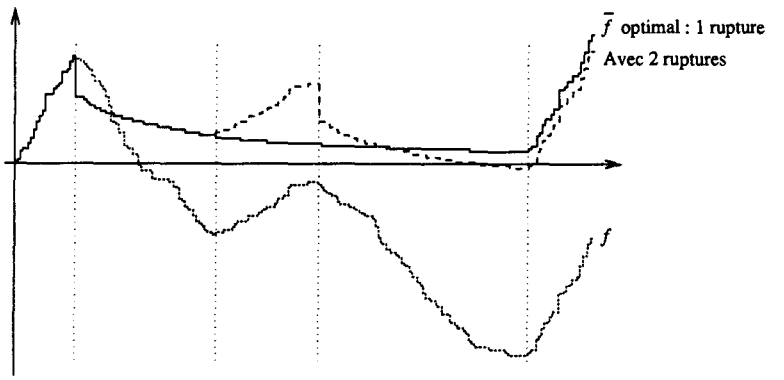


FIG. 2.6 - *Exemple de courbe optimale contenant une seule rupture*

Chapitre 3

Algorithme d'optimisation

Nous abordons la résolution du problème d'optimisation posé au chapitre précédent. Il s'agit, pour une courbe $f \in \mathcal{F}^n$ et une fonction de rupture $\mathcal{R}$, de construire une courbe $f^* \in Lift^*_{\mathcal{R}}(f)$ de valeur maximale pour l'abscisse n. On dit que f^* est une *courbe optimale*. Nous développons l'algorithme TURBOOPTLIFT qui construit une solution en temps $O(n\mathcal{R}(n))$. C'est un algorithme glouton : il procède par itération de liftings sur la courbe initiale f et ne remet jamais en cause les liftings effectués.

Les concepts de portions réductibles et irréductibles, courbes réductibles et irréductibles sont tout d'abord définis pour manipuler les sous-intervalles de $[0, n]$ sur lesquels la courbe courante peut être améliorée par l'application d'une rupture. Nous allons traduire le fait que, dans notre algorithme, toutes les ruptures appliquées sont de domaines suffisamment petits pour ne pas contrecarrer le choix d'un futur lifting qui pourrait produire un gain élevé. Pour cela, un ordre strict total $\prec$ est défini entre les domaines de ruptures potentielles (les ruptures qui peuvent être appliquées pour améliorer la courbe courante). Le domaine minimal pour $\prec$ détermine l'unique rupture applicable : celle dont l'application améliore la courbe courante mais n'handicape aucunement le choix d'une futur lifting potentiellement intéressant.

La première version de l'algorithme, que nous appelons OPTLIFT, se contente d'appliquer l'unique rupture applicable tant que la courbe courante peut être améliorée, c'est-à-dire tant qu'elle est réductible. Si on impose la condition de minimalité en rupture parmi toutes les courbes optimales irréductibles, on montre qu'il n'existe qu'une seule courbe $f^* \in Lift^*_{\mathcal{R}}(f)$ optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles. Nous montrons que l'algorithme OPTLIFT construit cette unique courbe f^*.

L'algorithme OPTLIFT calcule cette courbe optimale en temps $O(n^3)$. Nous l'améliorons en évitant d'appliquer un certain nombre de ruptures applicables qui sont moins intéressantes que d'autres ruptures considérées plus tard. Pour une abscisse i déterminée, les ruptures potentielles sont classées en utilisant l'ordre strict total $<_i$ qui permet de choisir immédiatement la plus intéressante. Toutes les ruptures applicables ne sont plus considérées, la plus intéressante est immédiatement appliquée.

L'algorithme TURBOOPTLIFT parcourt toutes les abscisses i de 0 à n et, à chaque étape, choisit la plus grande rupture potentielle pour l'ordre $<_i$. Si elle améliore la courbe courante sur $[0, i]$, elle est appliquée et la portion $[0, i]$ ne peut alors plus être améliorée par aucune rupture potentielle. L'algorithme est donc en ligne ("on-line" en anglais) puisqu'après avoir considéré la position i, la portion de courbe $[0, i]$ est optimisée. La courbe complète est optimisée après $n + 1$ étapes.

Étant donné la forme DCL des courbes de ruptures, la sélection de la rupture la plus grande pour $<_i$ se fait très efficacement. Nous montrons que l'algorithme TURBOOPTLIFT construit la même courbe que OPTLIFT mais en temps $O(n\mathcal{R}(n))$. Puisque les fonctions DCL sont toutes en $O(n)$, dans le pire des cas TURBOOPTLIFT fonctionne en temps $O(n^2)$. Dans le cas courant, la fonction DCL est logarithmique et TURBOOPTLIFT est en $O(n \log n)$. Nous montrons également que TURBOOPTLIFT est linéaire en espace.

3.1 Principe

Étant donné la courbe $f \in \mathcal{F}^n$ et la fonction de rupture $\mathcal{R}$ de retard $r_{\mathcal{R}}$, nous proposons un algorithme glouton qui procède par itérations successives de liftings sur la courbe. Notons $f^{(0)}$ la courbe initiale et $f^{(i)} \in \mathcal{F}^n_{\mathcal{R}}$ la courbe intermédiaire obtenue après i liftings :

$$f = f^{(0)} \xrightarrow{\mathcal{R}_{[x_1,y_1]}} f^{(1)} \xrightarrow{\mathcal{R}_{[x_2,y_2]}} f^{(2)} \dots \xrightarrow{\mathcal{R}_{[x_k,y_k]}} f^{(k)}$$

Notre solution optimale f^* est obtenue après k étapes : $f^* = f^{(k)}$.

Chaque étape de l'algorithme consiste à choisir un lifting et à l'appliquer. Le choix est guidé par deux contraintes :

1. Le gain du lifting doit être le plus élevé possible. Il est en effet facile de remarquer que $f^*(n) - f(n)$, est égal à la somme des gains de tous les liftings appliqués et donc que le gain du lifting choisi influence directement le gain total.

   Pour comparer des courbes de ruptures potentielles pour une abscisse i déterminée, la définition de l'ordre strict $<_i$ permet de choisir rapidement la plus intéressante. Cet ordre exploite la forme DCL des courbes de rupture.

2. Le choix du lifting à une étape ne doit pas désavantager le choix d'un bon lifting à une étape ultérieure. Un algorithme glouton ne revient jamais sur les décisions qu'il prend. Chaque lifting remplace une portion de la courbe par une portion de rupture, faisant ainsi disparaître des points de la courbe qui auraient pu être l'origine d'autres courbes de ruptures.

   Une rupture ne sera appliquée que si son domaine ne contient aucune portion pouvant elle-même être améliorée par un lifting. Cela définit le concept d'*unique rupture applicable*.

Tout point $(x, f^{(i)}(x))$ de la courbe courante $f^{(i)}$ est l'origine d'une *rupture potentielle* (ou *courbe de rupture potentielle*) qui peut être appliquée sur un intervalle $[x, y]$ pour engendrer le lifting $f^{(i)} \xrightarrow{\mathcal{R}_{[x,y]}} f^{(i+1)}$. Toutes les courbes de ruptures potentielles sont des translations de la courbe de rupture générique $\mathcal{R}$. Pour faciliter la manipulation de ces courbes de ruptures, désignons par $\mathcal{R}_{gx}$ la courbe de rupture potentielle, débutant à l'abscisse x pour la courbe partielle $g \in \mathcal{F}^n_{\mathcal{R}}$. Elle est définie par : $\mathcal{R}_{gx} : [x, n] \longrightarrow \mathbb{Z} : i \longmapsto \mathcal{R}_{gx}(i) = g(x) + \mathcal{R}(i - x)$. La figure 3.1 représente la courbe de rupture potentielle $\mathcal{R}_{f^{(i)}x}$ pour l'abscisse x et la courbe courante $f^{(i)}$.

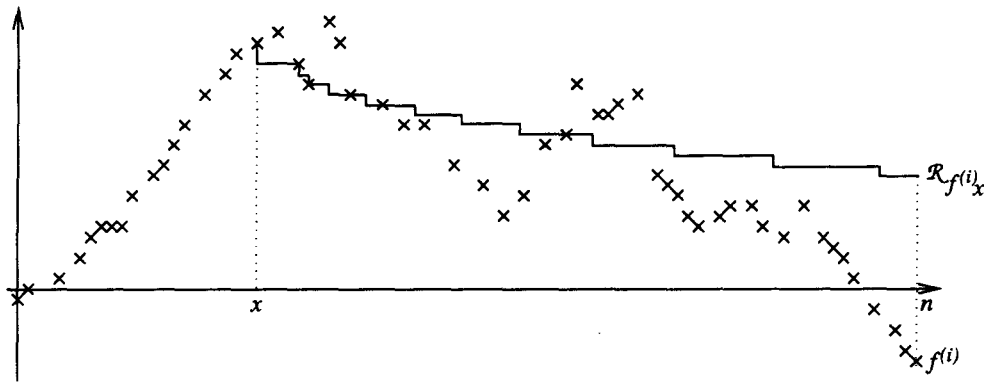


FIG. 3.1 - *Exemple de courbe de rupture* $\mathcal{R}_{f^{(i)}x}$

3.2 Comparaisons de courbes de ruptures

Pour une position i déterminée, nous définissons l'ordre strict total $<_i$ entre toutes les courbes de ruptures potentielles. Il permet de classer toutes les ruptures potentielles susceptibles d'améliorer la courbe courante sur $[0, i]$ et de choisir immédiatement la plus intéressante.

Grâce à cet ordre et à la forme DCL des courbes de ruptures potentielles, nous donnons deux propriétés qui permettent de comparer efficacement deux courbes de ruptures potentielles grâce à leurs positions d'origine.

Définition 3.1 *Soient* $i \in [1, n]$ *une position et* $g \in \mathcal{F}_{\mathcal{R}}^n$ *une courbe partielle avec rupture. Soient* $x \neq y$ *deux débuts de ruptures potentielles, tels que* $x, y \leq i - r_{\mathcal{R}}$. *La relation* $<_i$ *permet de comparer deux ruptures de la façon suivante :* $\mathcal{R}_{gx} <_i \mathcal{R}_{gy} \iff$ *une des deux conditions suivantes est satisfaite :*

1. $\mathcal{R}_{gx}(i) < \mathcal{R}_{gy}(i)$
2. $\mathcal{R}_{gx}(i) = \mathcal{R}_{gy}(i)$ *et* $\mathcal{R}_{gx}(i') < \mathcal{R}_{gy}(i')$ *avec* $i' < i$ *la plus grande abscisse telle que* $\mathcal{R}_{gx}(i') \neq \mathcal{R}_{gy}(i')$.

À la position i, si les deux ruptures ont une valeur différente, elles sont ordonnées selon l'ordre classique des entiers. Si les deux valeurs sont égales, alors l'ordre est effectué en utilisant les valeurs des ruptures aux positions $i-1, i-2, \ldots$ jusqu'à ce qu'une différence soit trouvée.

Montrons que la relation $<_i$ est un ordre strict total.

Proposition 3.1 *La relation* $<_i$ *est un ordre strict total.*

Preuve. Prouvons que $<_i$ est une relation transitive, irréflexive et totale.

1. Transitive : $\mathcal{R}_{gx} <_i \mathcal{R}_{gy}$ et $\mathcal{R}_{gy} <_i \mathcal{R}_{gz}$. Prouvons alors que $\mathcal{R}_{gx} <_i \mathcal{R}_{gz}$.

   1$^{\text{er}}$ cas : $\mathcal{R}_{gx}(i) < \mathcal{R}_{gy}(i)$.
   Dès lors $\mathcal{R}_{gx}(i) < \mathcal{R}_{gz}(i)$ et donc $\mathcal{R}_{gx} <_i \mathcal{R}_{gz}$.

   2$^{\text{ème}}$ cas : $\mathcal{R}_{gx}(i) = \mathcal{R}_{gy}(i)$.

   a. Si $\mathcal{R}_{gy}(i) < \mathcal{R}_{gz}(i)$.
   Alors $\mathcal{R}_{gx}(i) < \mathcal{R}_{gz}(i)$, donc $\mathcal{R}_{gx} <_i \mathcal{R}_{gz}$.

b. Si $\mathcal{R}_{gy}(i) = \mathcal{R}_{gz}(i)$.
Soient $i', i'' < i$ les plus grandes abscisses telles que $\mathcal{R}_{gx}(i') < \mathcal{R}_{gy}(i')$ et $\mathcal{R}_{gy}(i'') < \mathcal{R}_{gz}(i'')$. Elles existent puisque $\mathcal{R}_{gx} <_i \mathcal{R}_{gy}$ et $\mathcal{R}_{gy} <_i \mathcal{R}_{gz}$.
- Si $i' \leq i'' < i$ alors $\mathcal{R}_{gx}(i'') \leq \mathcal{R}_{gy}(i'')$ et $\mathcal{R}_{gy}(i'') < \mathcal{R}_{gz}(i'')$ d'où $\mathcal{R}_{gx} <_i \mathcal{R}_{gz}$.
- Si $i'' \leq i' < i$ alors $\mathcal{R}_{gx}(i') < \mathcal{R}_{gy}(i')$ et $\mathcal{R}_{gy}(i') \leq \mathcal{R}_{gz}(i')$ d'où $\mathcal{R}_{gx} <_i \mathcal{R}_{gz}$.

2. Irréflexive : Évident puisque $\mathcal{R}_{gx}(i) = \mathcal{R}_{gx}(i), \forall i \in [x, n]$.

3. Totale : Puisque la première marche d'une fonction DCL est plus haute que toutes les autres, que $x, y \leq i$ et que $x \neq y$, il existe toujours un indice $i' \leq i$ tel que $\mathcal{R}_{gx}(i') \neq \mathcal{R}_{gy}(i')$. □

La proposition 3.1 est très importante parce qu'elle établit, pour une abscisse i donnée et un ensemble de ruptures potentielles définies en i, qu'il existe une rupture maximale selon l'ordre $<_i$.

Les deux propositions qui suivent permettent de comparer instantanément deux ruptures potentielles lorsque leurs positions de début vérifient des conditions précises. La première considère le cas de deux courbes de ruptures qui ne se croisent jamais. La deuxième considère le cas de deux courbes qui se croisent.

Proposition 3.2 *Soient $g \in \mathcal{F}^n_{\mathcal{R}}$ et $x, x' \in [1, n]$ avec $x < x'$ et $g(x) \leq g(x')$ (voir figure 3.2). Alors, $\forall i \geq x' : \mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$.*

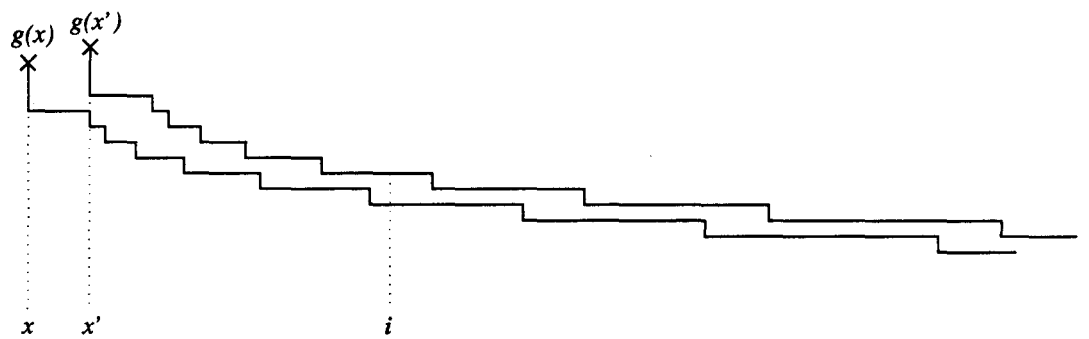


FIG. 3.2 - *Exemple de deux courbes de ruptures qui ne se croisent pas*

Preuve. Pour commencer, montrons que $\mathcal{R}_{gx}(i) \leq \mathcal{R}_{gx'}(i), \forall i \geq x'$. Par définition, $\mathcal{R}_{gx}(i) = g(x) + \mathcal{R}(i - x)$ et $\mathcal{R}_{gx'}(i) = g(x') + \mathcal{R}(i - x')$. On a $i - x > i - x'$ car $x < x'$. Puisque $\mathcal{R}$ est décroissante, $\mathcal{R}(i - x) \leq \mathcal{R}(i - x')$. Comme $g(x) \leq g(x')$ on a bien $\mathcal{R}_{gx}(i) \leq \mathcal{R}_{gx'}(i)$.

1er cas : $\mathcal{R}_{gx}(i) < \mathcal{R}_{gx'}(i)$. Par définition, $\mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$.

2ème cas : $\mathcal{R}_{gx}(i) = \mathcal{R}_{gx'}(i)$. Ce cas est uniquement possible lorsque $g(x) = g(x')$. Il existe une abscisse i' telle que $x' \leq i' < i$ et $\mathcal{R}_{gx}(i') \neq \mathcal{R}_{gx'}(i')$. Dans le pire des cas, $i' = x'$. Comme $\mathcal{R}_{gx}(i') < \mathcal{R}_{gx'}(i')$, on a $\mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$. □

Dans le cas d'une courbe intermédiaire $f^{(i)}$, la proposition 3.2 montre que si $x < x'$ et $f^{(i)}(x) \leq f^{(i)}(x')$ alors la rupture potentielle $\mathcal{R}_{f^{(i)}x}$ ne doit pas être prise en considération car elle n'apportera jamais un gain en lifting meilleur que la rupture potentielle $\mathcal{R}_{f^{(i)}x'}$.

La proposition 3.3 montre que lorsque deux ruptures potentielles $\mathcal{R}_{gx}$ et $\mathcal{R}_{gx'}$ se croisent pour l'ordre $<_i$, elles ne se croiseront plus jamais pour des abscisses supérieures. Elles sont confondues sur quelques positions après leur intersection.

Proposition 3.3 *Soient $g \in \mathcal{F}^n_{\mathcal{R}}$ et $x', x \in [1, n]$ tels que $x' < x$. Si $\mathcal{R}_{gx}(l) < \mathcal{R}_{gx'}(l)$ avec $l \geq x + r_{\mathcal{R}}$ (voir figure 3.3) alors, $\forall i \geq l : \mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$.*

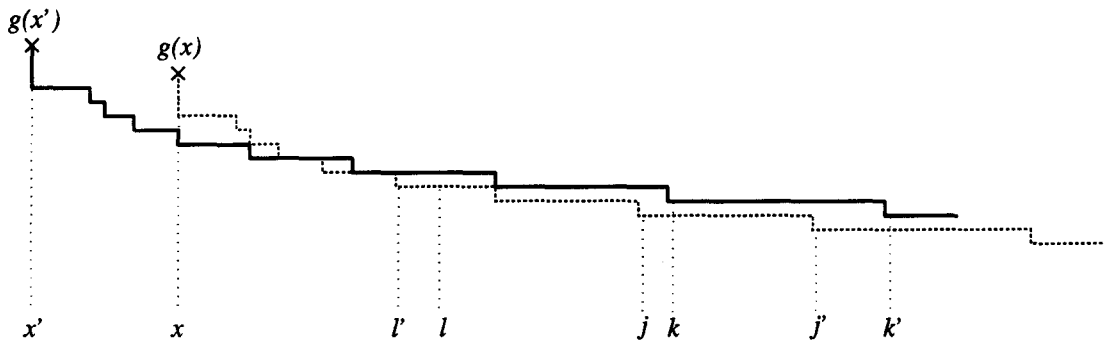


FIG. 3.3 - *Exemple de deux courbes de ruptures qui se croisent*

Preuve. Au premier point de la preuve, nous montrons qu'il existe une abscisse de début de marche $l' \leq l$ de $\mathcal{R}_{gx}$ telle que $\mathcal{R}_{gx}(l') < \mathcal{R}_{gx'}(l')$ (voir figure 3.3).

Au deuxième point, nous montrons que si, pour une abscisse de début de marche j de $\mathcal{R}_{gx}$, on a $\mathcal{R}_{gx}(j) < \mathcal{R}_{gx'}(j)$, alors pour toutes les positions i de cette marche : $\mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$ et pour l'abscisse j' de début de la marche suivante de $\mathcal{R}_{gx}$, $\mathcal{R}_{gx}(j') < \mathcal{R}_{gx'}(j')$.

Dès lors, par induction sur les positions de début des marches de $\mathcal{R}_{gx}$, on conclut $\mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$ pour tout $i \geq l'$.

1. Soit l' l'abscisse de début de la marche de $\mathcal{R}_{gx}$ sur laquelle se trouve l : $\mathcal{R}_{gx}(l'-1)-1 = \mathcal{R}_{gx}(l') = \mathcal{R}_{gx}(l)$. Puisque $\mathcal{R}_{gx'}(l') \geq \mathcal{R}_{gx'}(l)$, nous avons $\mathcal{R}_{gx}(l') < \mathcal{R}_{gx'}(l')$.

2. Soient $j, j' \geq x + r_{\mathcal{R}}$ les deux abscisses de début de deux marches successives de $\mathcal{R}_{gx}$: $\mathcal{R}_{gx}(j-1)-1 = \mathcal{R}_{gx}(j) = \mathcal{R}_{gx}(j'-1) = \mathcal{R}_{gx}(j')+1$. Montrons que si $\mathcal{R}_{gx}(j) < \mathcal{R}_{gx'}(j)$ alors $\forall i : j \leq i \leq j' : \mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$ et de plus $\mathcal{R}_{gx}(j') < \mathcal{R}_{gx'}(j')$.

   Soit $k > j$ le début de la marche de $\mathcal{R}_{gx'}$ avec $k > j$: $\mathcal{R}_{gx'}(j) = \mathcal{R}_{gx'}(k-1) = \mathcal{R}_{gx'}(k)+1$.

   On a forcément $k \geq x' + r_{\mathcal{R}}$ car $x' < x$.

   Il faut distinguer le cas où k n'appartient pas à la marche $[j, j'[$ et le cas où il y appartient.

   1$^{\text{er}}$ cas : $k \geq j'$.

   Dès lors, $\forall i : j \leq i \leq j'$, on a $\mathcal{R}_{gx}(i) < \mathcal{R}_{gx'}(i)$ et donc $\mathcal{R}_{gx} <_i \mathcal{R}_{gx'}$. De plus, $\mathcal{R}_{gx}(j') < \mathcal{R}_{gx'}(j')$.

   2$^{\text{ème}}$ cas : $k < j'$.

   Il est alors possible pour certains $i : j \leq i < j'$ d'avoir $\mathcal{R}_{gx}(i) = \mathcal{R}_{gx'}(i)$.

   Soit k' le début de la marche suivante de $\mathcal{R}_{gx'}$: $\mathcal{R}_{gx'}(k) = \mathcal{R}_{gx'}(k'-1) = \mathcal{R}_{gx'}(k')+1$. Comme $x' < x$, on a $(k-x') > (j-x)$. Puisque $\mathcal{R}_{gx}$ et $\mathcal{R}_{gx'}$ sont deux translations de la fonction DCL $\mathcal{R}$, que $j - x \geq r_{\mathcal{R}}$, que $k - x' \geq r_{\mathcal{R}}$ et que les marches d'une fonction DCL sont de plus en plus longues au-delà du retard $r_{\mathcal{R}}$, lorsque les

abscisses deviennent grandes, on a $((k' - x') - (k - x')) \geq ((j' - x) - (j - x))$. D'où $k' - k \geq j' - j \Rightarrow k' - j' \geq k - j$. Comme $k > j$, on a $k' > j'$. Dès lors, $\forall i : j \leq i < k$: $\mathcal{R}_{gx}(i) < \mathcal{R}_{gx'}(i)$. De plus, $\forall i : k \leq i < j'$: $\mathcal{R}_{gx}(i) \leq \mathcal{R}_{gx'}(i)$ et $\mathcal{R}_{gx}(j') < \mathcal{R}_{gx'}(j)$. □

Dès lors, si pour une courbe intermédiaire $f^{(i)}$, deux ruptures potentielles $\mathcal{R}_{f^{(i)}x}$ et $\mathcal{R}_{f^{(i)}x'}$, avec $x' < x$, sont telles que $\mathcal{R}_{f^{(i)}x}(l) < \mathcal{R}_{f^{(i)}x'}(l)$ pour $l \geq x + r_{\mathcal{R}}$, alors au-delà de l, la rupture potentielle $\mathcal{R}_{f^{(i)}x}$ ne doit pas être prise en considération car elle apporte un gain en lifting moins bon, au sens large, que la rupture potentielle $\mathcal{R}_{f^{(i)}x'}$.

3.3 Portions réductibles et rupture applicable

Pour pouvoir choisir la rupture à appliquer à chaque étape, nous définissons, pour une courbe, les concepts de portions réductibles, totalement réductibles, irréductibles ainsi que les notions de courbes réductibles et courbes irréductibles. Ces notions conduisent à la définition de l'unique rupture applicable d'une courbe réductible.

Une portion $[x, y]$ d'une courbe partielle avec rupture est *totalement réductible* si l'ordonnée de la courbe en y peut être augmentée par l'application d'une rupture de x à y ou par le prolongement d'une rupture jusqu'à l'abscisse y si x était déjà sur une portion de rupture.

Définition 3.2 *La portion $[x, y]$ de la courbe $g \in \mathcal{F}_{\mathcal{R}}^n$ est* totalement réductible *si $g(x)$, $g(y)$ sont définis et si :*

- *$g(x)$ est sur une portion d'application, $y \geq x + r_{\mathcal{R}}$ et $g(y) < \mathcal{R}_{gx}(y)$. On dit également que la rupture potentielle $\mathcal{R}_{gx}$* réduit *la portion $[x, y]$ de g.*
- *$g(x)$ est sur une portion de rupture $\mathcal{R}_{gt}$ avec $t < x$ et $g(y) < \mathcal{R}_{gt}(y)$. L'extension de $\mathcal{R}_{gt}$ jusqu'à l'abscisse y* réduit *la portion $[x, y]$ de g (voir figure 3.4). On préfère étendre la rupture $\mathcal{R}_{gt}$ jusqu'à l'abscisse y car, étant donné la forme des courbes de rupture, on a $\mathcal{R}_{gx} <_i \mathcal{R}_{gt}$ pour toute position $i \geq x + r_{\mathcal{R}}$ et donc $\mathcal{R}_{gx}$ n'est pas intéressante.*

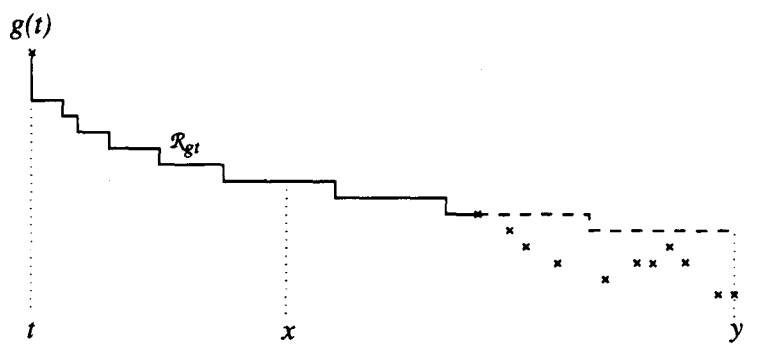


FIG. 3.4 - *Extension d'une rupture existante*

À chaque portion réductible $[x, y]$ de g est donc associé un lifting qui améliore la valeur de la courbe en y.

Si $g(y)$ est déjà sur une portion de rupture $[x', y']$ alors le domaine $[x, y]$ doit être étendu à $[x, y']$ comme l'indique la définition 2.5 d'un lifting.

Par extension, une portion *réductible* d'une courbe est une portion contenant au moins une portion totalement réductible.

Définition 3.3 *La portion* $[x, y]$ *de* $g \in \mathcal{F}_{\mathcal{R}}^n$ *est* réductible *s'il existe au moins une portion* $[x', y'] \subseteq [x, y]$ *telle que* $[x', y']$ *soit totalement réductible pour* g.

Par opposition, une portion *irréductible* ne contient aucune portion réductible.

Définition 3.4 *La portion* $[x, y]$ *de* $g \in \mathcal{F}_{\mathcal{R}}^n$ *est* irréductible *si elle ne contient aucune portion réductible de* g.

Les notions de réductibilité et irréductibilité peuvent être étendues à une courbe complète.

Définition 3.5 *La courbe* $g \in \mathcal{F}_{\mathcal{R}}^n$ *est* réductible *si elle contient au moins une portion réductible.*

Définition 3.6 *La courbe* $g \in \mathcal{F}_{\mathcal{R}}^n$ *est* irréductible *si elle ne contient aucune portion réductible.*

Définissons maintenant la relation $\prec$ qui permet d'ordonner des intervalles. Elle servira à choisir, parmi un ensemble de domaines de ruptures potentielles, celui qui réduit la courbe sans handicaper le choix d'un futur lifting.

Définition 3.7 *Soit* $[x, y]$ *et* $[x', y']$ *deux sous-intervalles de* $[0, n]$. *On dira que* $[x, y] \prec [x', y']$ *si et seulement si* $[x, y] \subsetneq [x', y']$ *ou* $y < y'$.

La figure 3.5 représente les quatre configurations possibles lorsque $[x, y] \prec [x', y']$.

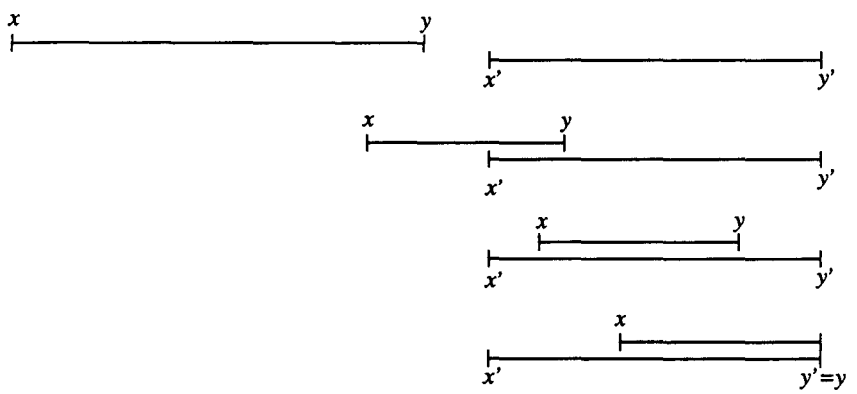


FIG. 3.5 - *Configurations possibles lorsque* $[x, y] \prec [x', y']$

Il est très intéressant de remarquer que $\prec$ définit un ordre strict total sur l'ensemble de tous les sous-intervalles.

Proposition 3.4 *La relation* $\prec$ *définit un ordre strict total sur l'ensemble de tous les sous-intervalles.*

Preuve. Prouvons que la relation $\prec$ est transitive, irréflexive et totale.

1. Transitive : $[x, y] \prec [x', y']$ et $[x', y'] \prec [x'', y'']$. Prouvons alors que $[x, y] \prec [x'', y'']$.

   1$^{\text{er}}$ cas : $[x, y] \subsetneq [x', y']$.

   a. Si $[x', y'] \subsetneq [x'', y'']$, alors $[x, y] \subsetneq [x'', y'']$ d'où $[x, y] \prec [x'', y'']$.

b. Si $[x',y'] \not\subset [x'',y'']$ et $y' < y''$, alors $y < y''$, donc $[x,y] \prec [x'',y'']$.

$2^{\text{ème}}$ cas : $[x,y] \not\subset [x',y']$ et $y < y'$. Dès lors, $y < y''$ et donc $[x,y] \prec [x'',y'']$.

2. Irréflexive : On n'a ni $[x,y] \subsetneq [x,y]$, ni $y < y$, donc $[x,y] \not\prec [x,y]$.

3. Totale : Pour $[x,y] \neq [x',y']$, on a soit $[x,y] \prec [x',y']$, soit $[x',y'] \prec [x,y]$. En effet, si $[x,y] \not\prec [x',y']$ cela signifie que $[x,y] \not\subset [x',y']$ et $y' \leq y$.

1^{er} cas : $y' < y$ dès lors $[x',y'] \prec [x,y]$.

$2^{\text{ème}}$ cas : $y' = y$ dès lors $[x',y'] \subsetneq [x,y]$ et donc $[x',y'] \prec [x,y]$. □

Lorsque $[x,y] \prec [x',y']$, on dit également que $[x,y]$ est *plus petite* que $[x',y']$ pour l'ordre $\prec$.

Puisque $\prec$ est un ordre strict total et puisqu'une courbe réductible contient toujours au moins une portion totalement réductible, une des portions totalement réductible est plus petite que toutes les autres portions totalement réductibles pour l'ordre $\prec$. C'est le domaine de l'*unique rupture applicable.*

Définition 3.8 *Soit $g \in \mathcal{F}_{\mathcal{R}}^n$ une courbe réductible. Une rupture est* applicable *sur la portion $[x,y]$ de g si :*

- *La portion $[x,y]$ de g est totalement réductible.*
- *Il n'existe aucune autre portion réductible $[x',y']$ de g telle que $[x',y'] \prec [x,y]$.*

3.4 Algorithme d'optimisation : OPTLIFT

Étant donné la forme des ruptures $\mathcal{R}$, nous présentons l'algorithme OPTLIFT qui optimise la courbe $f \in \mathcal{F}^n$ en temps $O(n^3)$. Cet algorithme procède par application, à chaque étape, de l'unique rupture applicable jusqu'à ce que la courbe devienne irréductible. La définition de minimalité en rupture parmi les courbes optimales irréductibles permet de prouver que OPTLIFT calcule l'unique courbe optimale qui soit de plus irréductible et minimale en rupture parmi les courbes optimales irréductibles.

3.4.1 L'algorithme

Étant donné $f^{(0)} \in \mathcal{F}^n$ la courbe à optimiser et $\mathcal{R}$ la courbe de rupture, la fonction OPTLIFT construit les courbes intermédiaires par application, à chaque étape, de l'unique rupture applicable. Son algorithme est donné à la figure 3.6.

La fonction DOMAINERUPTUREAPPLICABLE cherche le domaine $[x,y]$ de l'unique rupture applicable pour la courbe courante $f^{(i)}$ (le plus petit pour l'ordre $\prec$). La fonction LIFTING effectue le lifting $f^{(i)} \xrightarrow{\mathcal{R}_{[x,y]}} f^{(i+1)}$, elle retourne la nouvelle courbe $f^{(i+1)}$.

La proposition 3.5 montre que la fonction OPTLIFT réalise au plus n liftings.

Proposition 3.5 *Si $f^{(0)} \in \mathcal{F}^n$, alors* OPTLIFT *retourne la courbe irréductible $f^{(k)}$ avec $k \leq n$.*

OPTLIFT($f^{(0)}, \mathcal{R}$)
1 $i \leftarrow 0$
2 **TantQue** $f^{(i)}$ est réductible
3 **Faire** $[x, y] \leftarrow$ DOMAINERUPTUREAPPLICABLE($f^{(i)}, \mathcal{R}$)
4 $f^{(i+1)} \leftarrow$ LIFTING($f^{(i)}, \mathcal{R}, x, y$)
5 $i \leftarrow i + 1$
6 **Retourner** $f^{(i)}$

FIG. 3.6 - *Algorithme d'itération de la rupture applicable*

Preuve. Soit $f^{\mathcal{R}} \in \mathcal{F}^n_{\mathcal{R}}$ la courbe, qui ne contient qu'une seule rupture, définie par le lifting: $f^{(0)} \xrightarrow{\mathcal{R}_{[0,n]}} f^{\mathcal{R}}$. Il est facile de voir que chaque lifting de l'algorithme OPTLIFT augmente strictement le nombre de points qui sont sur des portions de rupture. Le premier lifting place au moins 2 points sur la première rupture (le début et la fin). De plus, d'après les définitions de lifting (définition 2.5) et de portion totalement réductible (définition 3.2), deux portions de ruptures ne sont jamais "collées" l'une à l'autre. Dès lors, dans le pire des cas, la courbe $f^{\mathcal{R}}$ est obtenue après n liftings. Par construction de $f^{\mathcal{R}}$, elle est irréductible. □

3.4.2 Minimalité en rupture et irréductibilité

Étant donné $f \in \mathcal{F}^n$ et $\mathcal{R}$ la courbe de rupture, il peut exister plusieurs courbes optimales.

Parmi celles-ci, nous définissons le concept de courbe optimale minimale en rupture. Suite à la forme discrète des courbes de ruptures, la solution finale offerte par OPTLIFT n'est pas forcément minimale en rupture mais elle est irréductible. Dès lors, on définit le concept de courbe optimale irréductible minimale en rupture parmi les courbes optimales irréductibles.

Définition 3.9 *Une courbe optimale est* minimale en rupture *parmi toutes les courbes optimales si le nombre de ses points qui appartiennent à des portions de rupture est minimal.*

C'est une propriété naturelle dans la mesure où pour un même gain en lifting, on choisit la courbe qui conserve le plus de points de la courbe initiale.

La courbe fournie par l'algorithme OPTLIFT n'est pas minimale en rupture parmi celles qui possèdent le même gain en lifting. En effet, prenons la courbe f de l'exemple de la figure 3.7.

D'après la proposition 3.3, ce cas n'est possible que lorsque $x' < x$ et $f(x') > f(x)$.

Étant donné l'ordre dans lequel les ruptures sont appliquées par OPTLIFT, la rupture de domaine $[x, y]$ est d'abord appliquée. Appelons $f^{(xy)}$ la courbe résultante: $f \xrightarrow{\mathcal{R}_{[x,y]}} f^{(xy)}$. Elle n'est pas irréductible, elle peut être réduite sur $[x', y'] \prec [x, y]$. Soit $f^{(x'y')}$ le résultat de l'application de la rupture sur $[x', y']$: $f^{(xy)} \xrightarrow{\mathcal{R}_{[x'y']}} f^{(x'y')}$. Remarquons que son domaine est étendu à $[x', y]$ comme l'impose la définition du lifting.

La courbe $f^{(x'y')}$ possède le même gain en lifting que $f^{(xy)}$ mais en plus elle est irréductible. Elle n'est pas minimale en rupture alors que $f^{(xy)}$ l'est, mais on montrera que $f^{(x'y')}$ est *optimale, irréductible et minimale en rupture parmi les courbes optimales irréductibles.*

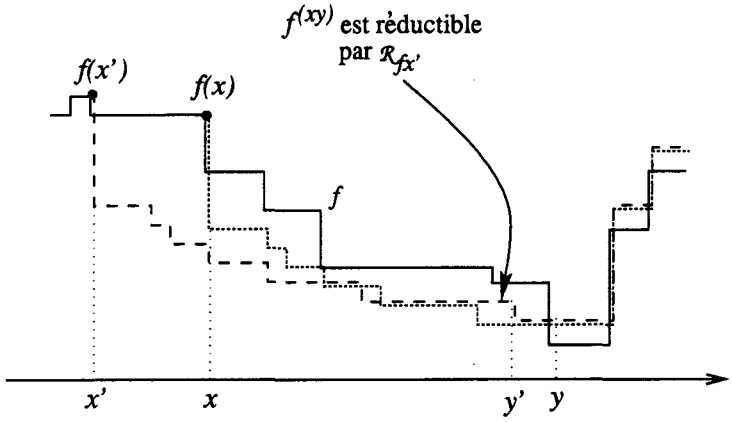


FIG. 3.7 - *Courbe optimale réductible et courbe optimale irréductible*

3.4.3 Optimalité de la solution

Nous montrons que pour toute courbe initiale $f \in \mathcal{F}^n$, il existe une et une seule courbe $f^{\mathcal{O}} \in Lift^*_{\mathcal{R}}(f)$ qui soit optimale, irréductible et minimale en rupture parmi les courbes optimales irréductibles et que cette courbe est calculée par notre algorithme OPTLIFT. C'est le rôle du théorème 3.9.

Avant cela, nous démontrons l'unicité de la courbe optimale, irréductible et minimale en rupture parmi les courbes optimales irréductibles grâce aux lemmes 3.6, 3.7 et au corollaire 3.8.

Le premier lemme montre que deux portions de ruptures de deux courbes optimales irréductibles peuvent uniquement se chevaucher dans un cas bien précis. Ce cas particulier découle de la forme discrète des fonctions DCL.

Lemme 3.6 *Soient $g, g' \in Lift^*_{\mathcal{R}}(f)$ deux courbes optimales et irréductibles. Soit $[x,y]$ une portion de rupture de g et $[x',y']$ une portion de rupture de g'. Si $x' < x < y' < y$ (c'est-à-dire que l'intervalle $[x,y]$ chevauche strictement l'intervalle $[x',y']$) alors (voir figure 3.8) :*

$$g(y') = g'(y') \text{ et } \mathcal{R}_{g'x'}(y) \leq g(y)$$

Preuve. Considérons le prolongement de $\mathcal{R}_{g'x'}$ jusqu'à l'abscisse y. Nous montrons que puisque g et g' sont optimales et irréductibles, il n'est pas possible d'avoir $\mathcal{R}_{g'x'}(y) > g(y)$ ou $g(y') \neq g'(y')$ ($g(y')$ est la valeur de la rupture $\mathcal{R}_{gx}$ pour l'abscisse y').

Il est évident que $g'(x') = g(x')$ et $g'(y) = g(y)$ autrement une des deux courbes n'est pas optimale.

Supposons $\mathcal{R}_{g'x'}(y) > g(y)$. Dans ce cas, g' n'est ni optimale ni irréductible comme nous l'avions supposé car la prolongation de $\mathcal{R}_{g'x'}(y)$ jusqu'à y réduit g' et produit un meilleur gain en lifting que g'.

Supposons $g(y') > g'(y')$. Alors g et g' ne sont pas optimales. En effet, soit $g'' \in Lift^*_{\mathcal{R}}(f)$ la courbe identique à g sur $[0,y']$, et dont la portion $[y',n]$ est celle de g' "soulevée" d'une hauteur de $g(y') - g'(y')$. C'est-à-dire que g'' est la courbe g dans laquelle la rupture $\mathcal{R}_{gx}$ est limitée à la portion $[x,y']$. Il est facile de voir que $g''(y) > g(y) = g'(y)$ et donc que g et g' n'étaient pas optimales comme nous l'avions supposé.

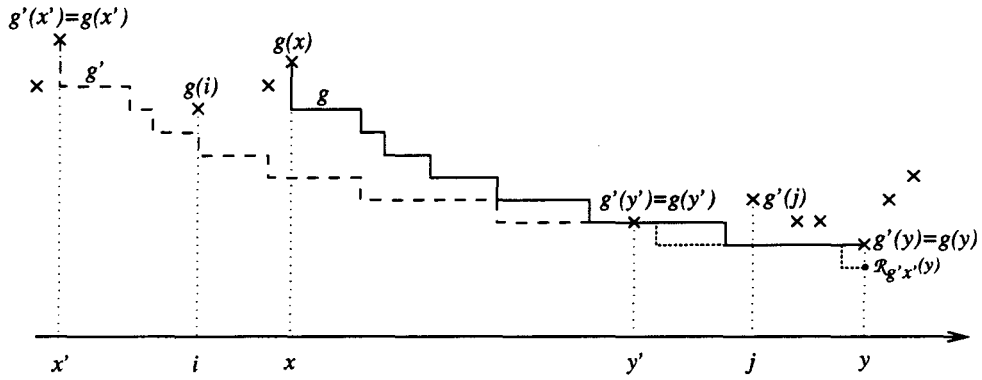


FIG. 3.8 - *Cas de $g, g' \in Lift^*_{\mathcal{R}}(f)$ optimales irréductibles avec $[x', y']$ portion de rupture de g', $[x, y]$ portion de rupture de g et $x' < x < y' < y$*

Supposons enfin que $g(y') < g'(y')$. Puisque $g'(y') = \mathcal{R}_{g'x'}(y') = \mathcal{R}_{gx'}(y')$, g n'est pas irréductible comme nous l'avions supposé: $\mathcal{R}_{gx'}$ réduit g sur $[x', y']$. □

Si l'une des deux courbes optimales irréductibles est minimale en rupture parmi les courbes optimales irréductibles, alors le cas de chevauchement autorisé par le lemme 3.6 n'est plus possible. De plus, toute portion de rupture de la courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles est incluse à une portion de rupture de la courbe optimale irréductible. C'est ce que montre le lemme 3.7.

Lemme 3.7 *Soient $f^O, g' \in Lift^*_{\mathcal{R}}(f)$ deux courbes optimales irréductibles. Supposons que f^O soit minimale en rupture parmi les courbes optimales irréductibles. Soit $[x, y]$ une portion de rupture de f^O et $[x', y']$ une portion de rupture de g'. Alors:*

1. *Il n'est pas possible que $x' < x < y' < y$ ou $x < x' < y < y'$. En d'autres mots, $[x, y]$ et $[x', y']$ ne se chevauchent pas.*

2. *Il existe une portion de rupture $[x'', y'']$ de g' telle que $[x, y] \subseteq [x'', y'']$. C'est-à-dire que la portion $[x, y]$ de g' doit être complètement incluse à une portion de rupture.*

Preuve.

1. Supposons $x' < x < y' < y$. Grâce au lemme 3.6, on sait que $f^O(y') = g'(y')$. Dès lors f^O n'est pas minimale en rupture car la courbe $g \in Lift^*_{\mathcal{R}}(f)$, construite à partir de f^O en limitant la rupture $\mathcal{R}_{fo_x}$ à $[x, y']$, offre le même gain en lifting que f^O mais possède moins de points sur des ruptures (voir figure 3.8, avec $g = f^O$).

   Supposons $x < x' < y < y'$. On se trouve de nouveau dans le cas du lemme 3.6 mais les rôles de f^O et g' ont été permutés tout comme les rôles de $[x, y]$ et $[x', y']$. Alors $f^O(y) = g'(y)$. Dès lors f^O n'est pas minimale en rupture car la courbe $g'' \in Lift^*_{\mathcal{R}}(f)$, construite à partir de g' en limitant la rupture $\mathcal{R}_{g'x'}$ à $[x', y]$, offre le même gain en lifting que f^O mais possède moins de points sur des ruptures.

   Les intervalles $[x, y]$ et $[x', y']$ ne se chevauchent donc pas.

2. On a $f^{\mathcal{O}}(x) = g'(x)$ et $f^{\mathcal{O}}(y) = g'(y)$ autrement une des deux courbes n'est pas optimale. Supposons qu'il existe une abscisse $i \in [x, y]$: $g'(i)$ est sur une portion d'application de g'. La figure 3.9 illustre ce cas lorsque la portion de rupture $[x', y']$ de g' est telle que $[x', y'] \subsetneq [x, y]$. Alors $f^{\mathcal{O}}$ n'est pas minimale en rupture comme nous l'avions supposé

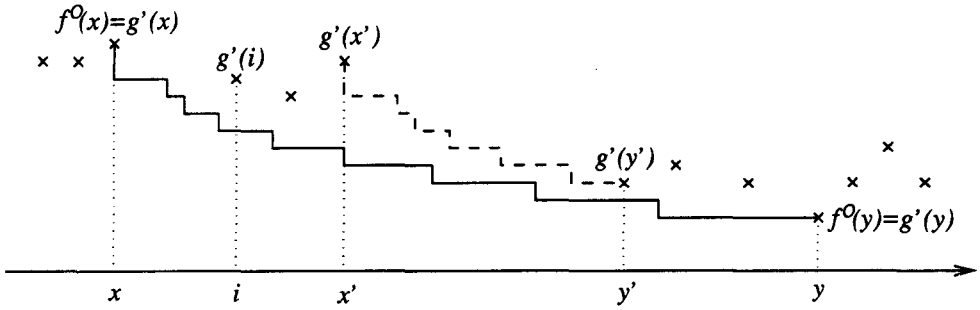


FIG. 3.9 - $[x, y]$ *portion de rupture de* $f^{\mathcal{O}}$ *et* $[x', y']$ *portion de rupture de* g' *avec* $[x', y'] \subsetneq [x, y]$

car le remplacement de sa portion $[x, y]$ par la portion $[x, y]$ de g' donne une courbe possédant le même gain en lifting mais possédant moins de points sur des ruptures.

Donc, il existe une portion de rupture $[x'', y'']$ de g' telle que $[x, y] \subseteq [x'', y'']$. □

Grâce à ce lemme, nous pouvons maintenant démontrer l'unicité de la courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles.

Corollaire 3.8 *Il existe une seule courbe optimale irréductible* $f^{\mathcal{O}} \in Lift^*_{\mathcal{R}}(f)$ *qui soit de plus minimale en rupture parmi les courbes optimales irréductibles.*

Preuve. Supposons que $f^{\mathcal{O}}$ ne soit pas unique: $f^{\mathcal{O}'} \in Lift^*_{\mathcal{R}}(f)$ est une courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles.

Puisque $f^{\mathcal{O}}$ est minimale en rupture, le lemme 3.7 montre que pour toute portion de rupture $[x, y]$ de $f^{\mathcal{O}}$, il existe une portion de rupture $[x', y']$ de $f^{\mathcal{O}'}$ telle que $[x, y] \subseteq [x', y']$. Comme $f^{\mathcal{O}'}$ est également minimale en rupture, en appliquant de nouveau le lemme 3.7, on obtient l'égalité entre les deux portions: $[x, y] = [x', y']$. Dès lors $f^{\mathcal{O}}$ et $f^{\mathcal{O}'}$ ont exactement les mêmes portions de ruptures et portions d'applications. Donc $f^{\mathcal{O}} = f^{\mathcal{O}'}$. □

On peut maintenant prouver que l'algorithme OPTLIFT fournit l'unique courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles.

Théorème 3.9 *Soit* $f \in \mathcal{F}^n$ *et* $\mathcal{R}$ *la courbe de rupture. Soit* $f^* = \text{OPTLIFT}(f, \mathcal{R})$ *alors :*

1. $f^* \in Lift^*_{\mathcal{R}}(f)$.
2. f^* *est irréductible.*
3. *Soit* $f^{\mathcal{O}} \in Lift^*_{\mathcal{R}}(f)$ *l'unique courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles. Alors* $f^* = f^{\mathcal{O}}$.

Preuve.

1. Par définition de $Lift^*_{\mathcal{R}}(f)$ et puisque f^* est obtenue par itération de liftings sur la courbe initiale f, on a $f^* \in Lift^*_{\mathcal{R}}(f)$.

2. f^* est irréductible car c'est la condition d'arrêt de OPTLIFT.

3. Montrons que $f^* = f^{\mathcal{O}}$. Nous commençons par montrer que f^* est optimale.

   Soit $[x, y]$ une portion de rupture de $f^{\mathcal{O}}$. Elle n'a pu être construite que par itération de ruptures applicables, de domaines $[x', y']$ tels que $[x', y'] \cap [x, y] \neq \emptyset$. En effet, considérons le lifting $f^{(i-1)} \xrightarrow{\mathcal{R}_{[x',y']}} f^{(i)}$ effectué à l'étape i de l'algorithme. La portion $[x', y']$ est le domaine de la rupture applicable de $f^{(i-1)}$. Par définition d'un lifting (définition 2.5), si $[x', y'] \cap [x, y] = \emptyset$, alors la portion $[x, y]$ de $f^{(i)}$ est identique à la portion $[x, y]$ de $f^{(i-1)}$ (éventuellement soulevée si $y' < x$).

   Plaçons-nous à l'étape pour laquelle $[0, x-1]$ deviennent irréductible. Cela se produit dans OPTLIFT lorsque toutes les ruptures applicables de domaines $[x', y']$ avec $y' < x$ ont été appliquées. Dès lors, la portion de courbe $[x, y]$ ne sera plus jamais soulevée par l'algorithme en la laissant telle quelle.

   Pour la courbe courante $f^{(i)}$, tant que $f^{(i)}(y) < f^{\mathcal{O}}(y)$, la portion $[x, y]$ de $f^{(i)}$ est totalement réductible par $\mathcal{R}_{f^{(i)}x}$. Soit $[x', y']$ le domaine de la rupture applicable.

   Montrons alors que $[x', y'] \subseteq [x, y]$. On sait que $y' \leq y$ par définition du domaine de la rupture applicable. Supposons $x' < x$. Alors, $\forall t \in [x, y]$, on a $\mathcal{R}_{f^{(i)}x}(t) \geq \mathcal{R}_{f^{(i)}x'}(t)$ autrement $f^{\mathcal{O}}$ n'est pas optimale ou n'est pas irréductible. Dès lors $\mathcal{R}_{f^{(i)}x}(y') > f^{(i)}(y')$ et donc la portion $[x, y']$ de $f^{(i)}$ est totalement réductible. Comme $[x, y'] \prec [x', y']$, la portion $[x', y']$ n'est pas le domaine de la rupture applicable comme nous l'avions supposé (voir figure 3.10)

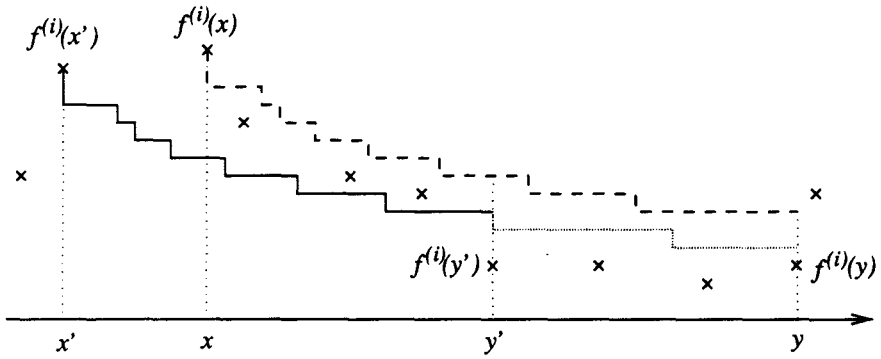


FIG. 3.10 - $[x, y'] \prec [x, y]$ *et* $[x, y]$ *totalement réductible*

Tant que $f^{(i)}(y) < f^{\mathcal{O}}(y)$, $f^{(i)}$ est réductible par l'application d'une rupture sur $[x', y'] \subseteq [x, y]$. Puisque $f^{\mathcal{O}}$ est optimale, l'optimisation se termine avec $f^{(i)}(y) = f^{\mathcal{O}}(y)$.

Nous avons montré que f^* est optimale. Il nous reste à montrer que $f^* = f^{\mathcal{O}}$, c'est-à-dire que f^* a exactement les mêmes portions de rupture que $f^{\mathcal{O}}$.

Montrons d'abord que toute portion de rupture de $f^{\mathcal{O}}$ est également une portion de rupture de f^*.

D'après le lemme 3.7, aucune portion de rupture de $f^{\mathcal{O}}$ ne chevauche une portion de rupture de f^*. De plus, pour toute portion de rupture $[x, y]$ de $f^{\mathcal{O}}$, il existe une portion de rupture $[x', y']$ de f^* telle que $[x, y] \subseteq [x', y']$. Supposons que $[x, y] \subsetneq [x', y']$. Puisque

les deux courbes sont optimales et irréductibles, $f^*(x') = f^{\mathcal{O}}(x')$ et $f^*(y') = f^{\mathcal{O}}(y')$. Comme $[x,y] \subsetneq [x',y']$ on a $[x,y] \prec [x',y']$ et donc la rupture de domaine $[x,y]$ devient applicable avant celle de domaine $[x',y']$. La courbe $f^{\mathcal{O}}$ étant irréductible, l'application de la rupture de domaine $[x,y]$ rend $[x',y']$ irréductible pour la courbe courante. On ne peut donc pas avoir $[x,y] \subsetneq [x',y']$. Donc $[x,y] = [x',y']$.

Il nous reste à montrer que toute portion de rupture de f^* est également une portion de rupture de $f^{\mathcal{O}}$.

Supposons que la portion de rupture $[x',y']$ de f^* soit, pour la courbe $f^{\mathcal{O}}$, une portion d'application. Puisque $f^{\mathcal{O}}$ est irréductible, la portion $[x',y']$ de la courbe initiale ne contient aucune portion susceptible de devenir le domaine d'une rupture applicable. La portion de rupture $[x',y']$ de f^* est donc obligatoirement une portion de rupture dans $f^{\mathcal{O}}$. □

3.4.4 Étude de complexité

Montrons que l'algorithme OPTLIFT fonctionne en temps $O(n^3)$ si la courbe à optimiser est $f \in \mathcal{F}^n$.

Pour cela, nous supposons que l'application d'une rupture se fait en temps constant. Cette condition n'est pas restrictive, elle sera discutée à la section 3.5.5.

Théorème 3.10 *Soit $f \in \mathcal{F}^n$ et $\mathcal{R}$ la courbe de rupture. L'algorithme* OPTLIFT *fournit l'unique solution $f^* \in Lift^*_{\mathcal{R}}(f)$, optimale irréductible et minimale en rupture parmi les solutions optimales irréductibles, en temps $O(n^3)$.*

Preuve. Grâce à la proposition 3.5, nous savons qu'au maximum n liftings sont appliqués. Pour chaque lifting, le domaine de la rupture applicable doit être recherché (fonction DOMAINERUPTUREAPPLICABLE).

Montrons que DOMAINERUPTUREAPPLICABLE recherche le domaine en temps $O(n^2)$. Considérons l'algorithme de la figure 3.11. Il parcourt tous les sous-intervalles de $[0,n]$ en suivant l'ordre $\prec$. En effet, $\forall x,y : x \leq 1 \leq y \leq n$, on a $[x,y] \prec [x-1,y]$ et $[0,y-1] \prec [y,y]$.

```
DOMAINERUPTUREAPPLICABLE(f^(i), R)
1  Pour y ← 0 Jusque n
2  Faire Pour x ← y Jusque 0
3        Faire Si [x,y] est totalement réductible pour f^(i)
4              Alors Retourner [x,y]
```

FIG. 3.11 - *Recherche du domaine de la rupture applicable de $f^{(i)}$*

La première portion totalement réductible rencontrée est celle de domaine minimum, elle est donc applicable. L'algorithme DOMAINERUPTUREAPPLICABLE fonctionne en temps $O(n^2)$ et donc OPTLIFT fonctionne en temps $O(n^3)$. □

Remarque 3.1 *Une question qui vient immédiatement à l'esprit est: ne peut-on pas améliorer l'algorithme de* DOMAINERUPTUREAPPLICABLE *pour ne pas reconsidérer à chaque fois les portions plus petites que le domaine de la dernière rupture appliquée? La réponse est non!*

Prenons en effet l'exemple présenté à la section 3.4.2 (figure 3.7). Après avoir appliqué la rupture applicable sur $[x, y]$, le domaine de la prochaine rupture applicable est $[x', y'] \prec [x, y]$. À cause de la forme discrète des courbes de ruptures, il faut donc aussi considérer à chaque étape les domaines plus petits que celui de la dernière rupture appliquée.

3.5 Algorithme amélioré TURBOOPTLIFT

La complexité de l'algorithme d'optimisation peut-être améliorée lorsque l'on sait que l'application de certaines ruptures applicables réduisent la courbe sur le domaine de ruptures précédemment appliquées. En l'occurrence, si une courbe intermédiaire est irréductible sur l'intervalle $[0, i-1]$, mais réductible sur l'intervalle $[0, i]$; alors la rupture qu'il conviendra d'appliquer est la plus grande pour l'ordre $<_i$. Son application rend automatiquement $[0, i]$ irréductible sans devoir passer par l'intermédiaire de toutes les ruptures applicables.

L'algorithme TURBOOPTLIFT optimise la courbe en ligne: l'unique courbe optimale irréductible définie sur $[0, i]$ et minimale en rupture parmi les courbes optimales irréductibles définies sur $[0, i]$ est calculée à l'étape i. Les étapes sont numérotées de 0 à n. La solution au problème initial est donc complètement calculée à la fin de l'étape n. C'est la même solution que celle proposée par l'algorithme OPTLIFT.

Le choix de la plus grande rupture potentielle pour l'ordre $<_i$ est facilité par la connaissance de la *Liste des Ruptures Potentielles* (LRP), classées selon l'ordre $<_i$. Les propositions 3.2 et 3.3 permettent de limiter le nombre de ruptures présentes dans LRP à $|\mathcal{R}(n)|$. En évitant de construire toutes les courbes intermédiaires $f^{(i)}$, l'algorithme TURBOOPTLIFT construit l'unique courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles en temps $O(n\mathcal{R}(n))$. De plus, TURBOOPTLIFT est linéaire en espace.

3.5.1 Principe

Nous allons voir qu'il n'est pas nécessaire d'appliquer toutes les ruptures applicables. L'ordre $<_t$ nous permet de choisir efficacement la rupture à appliquer.

Lemme 3.11 (d'application directe) *Supposons $f^{(i)}$ irréductible sur $[0, t-1]$ mais réductible par les k ruptures potentielles $\mathcal{R}_{f^{(i)}x_1}, \ldots, \mathcal{R}_{f^{(i)}x_k}$ sur $[x_1, t], \ldots, [x_k, t]$. Par applications des ruptures applicables, c'est finalement la plus grande, pour l'ordre $<_t$, qui rend $[0, t]$ irréductible.*

Preuve. Soit E l'ensemble de toutes les ruptures qui réduisent $f^{(i)}$ sur $[0, t]$:
$E = \{\mathcal{R}_{f^{(i)}x_1}, \ldots, \mathcal{R}_{f^{(i)}x_k}\}$ et $X = \{x_1, \ldots, x_k\}$ l'ensemble des abscisses de début de ces ruptures. Soit $\mathcal{R}_{f^{(i)}x_j}$ l'unique rupture applicable: $f^{(i)} \xrightarrow{\mathcal{R}_{[x_j,t]}} f^{(i+1)}$ pour l'algorithme OPTLIFT. Par définition du domaine de la rupture applicable, x_j est l'abscisse maximale de X.

Si, à l'étape suivante, une rupture $\mathcal{R}_{f^{(i+1)}x}$ réduit la portion $[x, t]$ de $f^{(i+1)}$, alors $\mathcal{R}_{f^{(i)}x}$ aurait réduit $f^{(i)}$ sur $[x, t]$. En effet, cela signifie qu'il existe $u \in [x, t]$: $\mathcal{R}_{f^{(i+1)}x}(u) > f^{(i+1)}(u)$. Cette situation n'est possible que si $u \in [x_j, t]$ (dans le cas contraire, $[x_j, t]$ n'était pas le domaine de la rupture applicable de $f^{(i)}$ comme nous l'avions supposé car $[x, u] \prec [x_j, t]$) et $x < x_j$ (autrement $\mathcal{R}_{f^{(i+1)}x}$ est sous $f^{(i+1)}$ partout) comme le montre la figure 3.12. D'après la proposition 3.2, ce n'est possible que si $f^{(i+1)}(x) > f^{(i)}(x_j)$ et d'après la proposition 3.3,

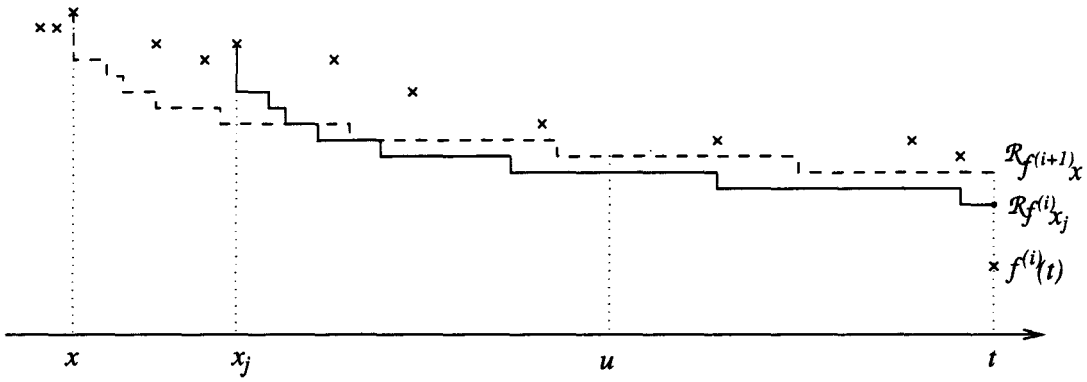


FIG. 3.12 - *La rupture $\mathcal{R}_{f^{(i)}x}$ appartient à E*

$\mathcal{R}_{f^{(i+1)}x}(t) \geq \mathcal{R}_{f^{(i+1)}x_j}(t) = \mathcal{R}_{f^{(i)}x_j}(t) > f^{(i)}(t)$. Donc $\mathcal{R}_{f^{(i)}x}$ réduit $f^{(i)}$ sur $[x,t]$. Donc $\mathcal{R}_{f^{(i)}x} \in E$.

Soit $\mathcal{R}_{f^{(i)}x_h}$ la plus grande rupture pour l'ordre $<_t$. Elle réduit toutes les autres ruptures de E sur $[x_h,t]$. Dès lors, si on applique $\mathcal{R}_{f^{(i)}x_h}$ sur $[x_h,t]$ plutôt que la rupture applicable, alors la portion $[0,t]$ devient irréductible.

Puisque l'ordre $<_t$ est un ordre strict total, parmi toutes les ruptures de E qui ont une ordonnée maximale en t pour l'ordre classique de comparaison des entiers, $\mathcal{R}_{f^{(i)}x_h}$ est la rupture dont l'origine est d'abscisse maximale. Elle devient donc applicable avant les autres. Après application des ruptures applicables de E, c'est finalement celle là qui rend $[0,t]$ irréductible. □

On peut maintenant préciser l'amélioration proposée par l'algorithme TURBOOPTLIFT de la section 3.5.5.

L'idée est de parcourir toutes les abscisses i de 0 à n et pour chaque i, de regarder si la plus grande rupture potentielle $\mathcal{R}_{f^{(i)}x_i}$, pour l'ordre $<_i$, réduit la courbe courante $f^{(i)}$ sur $[0,i]$. Si oui, le lifting $f^{(i)} \xrightarrow{\mathcal{R}_{[x_i,i]}} f^{(i+1)}$ est effectué. Dans le cas contraire, la courbe courante est laissée telle quelle: $f^{(i+1)} \longleftarrow f^{(i)}$. Après chaque étape i, la courbe courante devient irréductible sur $[0,i]$.

À la fin de l'étape n, l'unique courbe optimale, irréductible et minimale en rupture parmi les courbes optimales irréductibles aura été calculée.

3.5.2 La Liste des Ruptures Potentielles: *LRP*

Nous définissons le concept de *Liste des Ruptures Potentielles (LRP)* qui limite le nombre de ruptures à considérer. Nous montrons qu'à chaque étape de l'algorithme, cette liste contient au maximum $|\mathcal{R}(n)|$ ruptures. Cette borne est très importante car l'efficacité de l'algorithme TURBOOPTLIFT en découle directement.

À chaque étape i, il faut être capable de déterminer en temps raisonnable quelle est la plus grande rupture potentielle $\mathcal{R}_{f^{(i)}x_i}$ pour l'ordre $<_i$. Pour cela, à chaque étape i l'algorithme TURBOOPTLIFT gère la *Liste des Ruptures Potentielles* $LRP^{(i)}$ qui contient toutes les positions $x_k \leq i - r_{\mathcal{R}}$ qui sont des abscisses de début de ruptures potentielles $\mathcal{R}_{f^{(i)}x_k}$ qui pourraient devenir les plus grandes pour des ordres $<_h$ avec $i \leq h$. Nous avons vu (proposi-

tions 3.2 et 3.3) que certaines ruptures potentielles ne doivent pas être considérées car elles apporteront toujours un gain moins bon que d'autres.

Pour être dans $LRP^{(i)}$, une position x doit satisfaire les quatre conditions suivantes.

1. La courbe doit être définie en x : $f^{(i)}(x)$ existe.
2. L'origine x de la rupture potentielle doit au moins se trouver à une distance $r_{\mathcal{R}}$ de la position courante : $i - x \geq r_{\mathcal{R}}$. On n'applique en effet jamais une rupture sur un domaine de largeur inférieure au retard $r_{\mathcal{R}}$ (définition d'une portion totalement réductible).
3. Il n'existe pas, dans $LRP^{(i)}$, de position x' de début de rupture potentielle avec $x < x'$ et $f^{(i)}(x) \leq f^{(i)}(x')$. En effet, si une telle position x' est présente dans $LRP^{(i)}$, alors $\mathcal{R}_{f^{(i)}x} <_j \mathcal{R}_{f^{(i)}x'}$ pour toute abscisse $j \geq x'$ (proposition 3.2) et donc la rupture potentielle $\mathcal{R}_{f^{(i)}x}$ ne pourra jamais être la plus grande pour un ordre $<_h$ avec $h \geq i$.
4. Il n'existe pas, dans $LRP^{(i)}$, de position x' de début de rupture potentielle avec $x' < x$ et $\mathcal{R}_{f^{(i)}x}(l) < \mathcal{R}_{f^{(i)}x'}(l)$ pour une abscisse $l \leq i$. Autrement, on aurait $\mathcal{R}_{f^{(i)}x} <_j \mathcal{R}_{f^{(i)}x'}$ pour toute position $j \geq l$ (proposition 3.3) et donc la rupture potentielle $\mathcal{R}_{f^{(i)}x}$ ne pourra jamais être la plus grande pour un ordre $<_h$ avec $h \geq i$.

Ces quatre conditions limitent fortement le nombre de positions de $LRP^{(i)}$. Le lemme 3.12 montre qu'au maximum $|\mathcal{R}(i)|$ positions peuvent figurer dans $LRP^{(i)}$.

Lemme 3.12 (de simplification) $\#LRP^{(i)} \leq |\mathcal{R}(i)|$.

Preuve. Soit $x_m \leq i - r_{\mathcal{R}}$ la plus grande abscisse telle que : $\forall j \leq i : f^{(i)}(j) \leq f^{(i)}(x_m)$. C'est-à-dire que l'abscisse x_m est la plus grande d'ordonnée maximale (voir figure 3.13). L'abscisse

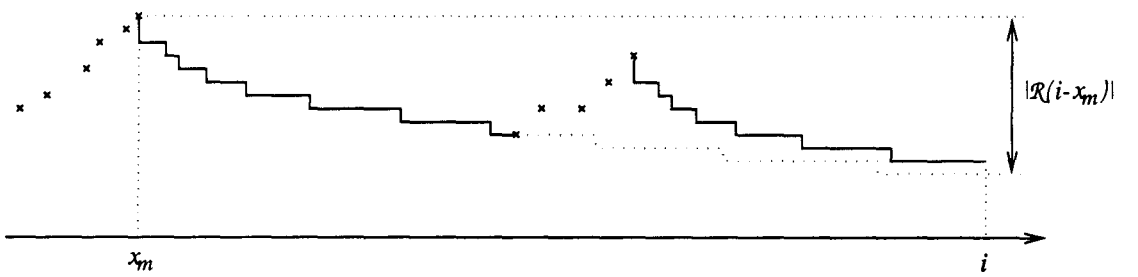


FIG. 3.13 - *Nombre de ruptures potentielles de* $LRP^{(i)}$

x_m est dans $LRP^{(i)}$, en effet :

1. $f^{(i)}(x_m)$ existe.
2. $i - x_m \geq r_{\mathcal{R}}$.
3. Il n'existe pas, dans $LRP^{(i)}$, de position x' de début de rupture potentielle avec $x_m < x'$ et $f^{(i)}(x_m) \leq f^{(i)}(x')$ puisque x_m est le maximum le plus à droite.
4. Il n'existe, dans $LRP^{(i)}$, aucune position x' de début de rupture potentielle avec $x' < x_m$ car dans ce cas, $f^{(i)}(x') \leq f^{(i)}(x_m)$ et la proposition 3.2 montre alors que $\mathcal{R}_{f^{(i)}x'} <_l \mathcal{R}_{f^{(i)}x_m}, \forall l \geq x_m$.

Toutes les abscisses x de $LRP^{(i)}$ sont donc telles que $x_m \leq x \leq i - r_{\mathcal{R}}$.

La proposition 3.2 interdit deux positions x et x' avec $f^{(i)}(x) = f^{(i)}(x')$ de se trouver en même temps dans $LRP^{(i)}$ car si $x < x'$ alors $\mathcal{R}_{f^{(i)}x} <_l \mathcal{R}_{f^{(i)}x'} \forall l \geq x'$. Il y aura donc au plus dans $LRP^{(i)}$ une rupture potentielle pour chaque ordonnée.

Aucune position x avec $x_m \leq x \leq i$ et $f^{(i)}(x) \leq \mathcal{R}_{f^{(i)}x_m}(x)$ ne peut appartenir à $LRP^{(i)}$ puisque $\mathcal{R}_{f^{(i)}x} <_l \mathcal{R}_{f^{(i)}x_m} \forall l \geq x$.

Donc le nombre maximal de positions de $LRP^{(i)}$ est déterminé par la "hauteur" maximale de la rupture potentielle démarrant en x_m, c'est-à-dire $|\mathcal{R}(i - x_m)|$ (voir figure 3.13). Dans le pire des cas, $x_m = 0$ et donc $\#LRP^{(i)} \leq |\mathcal{R}(i)|$. □

Remarque 3.2 *Lorsque la rupture potentielle $\mathcal{R}_{f^{(i)}x}$ est appliquée sur le domaine $[x, i]$ de la courbe courante, $LRP^{(i)}$ ne doit pas être modifiée. En effet, la rupture appliquée ne doit pas être supprimée car elle peut de nouveau être appliquée lors d'une étape $j > i$ de l'algorithme si elle est la plus grande pour l'ordre $<_j$.*

D'autre part, il n'existait dans $LRP^{(i)}$ aucune abscisse x' appartenant au domaine $[x, i]$ de la rupture appliquée. En effet, cela n'aurait pu être le cas que si $\mathcal{R}_{f^{(i)}x} <_i \mathcal{R}_{f^{(i)}x'}$ (proposition 3.3 et définition de $LRP^{(i)}$) et dès lors c'est $\mathcal{R}_{f^{(i)}x'}$ qui aurait dû être appliquée.

Remarque 3.3 *Pour toute étape i, il y a donc au maximum $|\mathcal{R}(n)|$ ruptures potentielles dans $LRP^{(i)}$.*

3.5.3 Classement de LRP

Bien que $LRP^{(i)}$ contienne au maximum $|\mathcal{R}(n)|$ ruptures potentielles, la recherche de la rupture maximale pour l'ordre $<_i$ ne se fait pas en temps $O(\mathcal{R}(n))$. En effet, l'ordre $<_i$ impose la comparaison des courbes de ruptures sur plusieurs positions précédentes lorsqu'il y a égalité à la position i.

Dès lors, nous imposons que $LRP^{(i)}$ soit classée selon l'ordre $<_i$. La recherche de la plus grande rupture potentielle se fait alors en temps $O(1)$. Nous montrons dans cette section que la construction de $LRP^{(i+1)}$, triée selon l'ordre $<_{i+1}$, se fait en temps $O(\mathcal{R}(n))$ à partir de $LRP^{(i)}$ triée selon l'ordre $<_i$.

Lemme 3.13 (de classement) *La construction de $LRP^{(i+1)}$, triée selon l'ordre $<_{i+1}$, se fait en temps $O(\mathcal{R}(n))$ à partir de $LRP^{(i)}$ triée selon l'ordre $<_i$.*

Preuve. Soit $LRP^{(i)} = (x_1, x_2, \ldots, x_k)$. Nous montrons que la construction de $LRP^{(i+1)}$ peut être réalisée par un seul parcours des ruptures de $LRP^{(i)}$ en comparant simplement les ruptures consécutives deux à deux.

Considérons x_j et x_{j+1} deux positions consécutives de $LRP^{(i)}$. On a $\mathcal{R}_{f^{(i)}x_j} <_i \mathcal{R}_{f^{(i)}x_{j+1}}$. Par la définition de l'ordre $<_i$, il faut distinguer deux cas :

1$^{\text{er}}$ cas : $\mathcal{R}_{f^{(i)}x_j}(i) < \mathcal{R}_{f^{(i)}x_{j+1}}(i)$. Puisque la fonction $\mathcal{R}$ est DCL, on a $\mathcal{R}_{f^{(i)}x_j}(i+1) \leq \mathcal{R}_{f^{(i)}x_{j+1}}(i+1)$. Donc $\mathcal{R}_{f^{(i)}x_j} <_{i+1} \mathcal{R}_{f^{(i)}x_{j+1}}$ car si $\mathcal{R}_{f^{(i)}x_j}(i+1) = \mathcal{R}_{f^{(i)}x_{j+1}}(i+1)$, on a de toute façon $\mathcal{R}_{f^{(i)}x_j}(i) < \mathcal{R}_{f^{(i)}x_{j+1}}(i)$. L'ordre relatif entre x_j et x_{j+1} reste donc inchangé dans $LRP^{(i+1)}$.

$2^{\text{ème}}$ cas : $\mathcal{R}_{f^{(i)}x_j}(i) = \mathcal{R}_{f^{(i)}x_{j+1}}(i)$. Si $\mathcal{R}_{f^{(i)}x_j}(i+1) \leq \mathcal{R}_{f^{(i)}x_{j+1}}(i+1)$ alors l'ordre relatif entre x_j et x_{j+1} reste inchangé dans $LRP^{(i+1)}$. Autrement, si $\mathcal{R}_{f^{(i)}x_j}(i+1) > \mathcal{R}_{f^{(i)}x_{j+1}}(i+1)$ alors l'ordre relatif entre x_j et x_{j+1} est modifié : il y a un croisement entre les deux courbes de ruptures potentielles. Grâce à la proposition 3.3, nous savons que la rupture potentielle d'origine x_{j+1} n'aura plus jamais une valeur supérieure à celle d'origine x_j pour un ordre $<_l$ avec $l \geq i$. La liste $LRP^{(i+1)}$ ne doit plus contenir x_{j+1}.

Si l'ordre entre x_j et x_{j+1} est conservé, on continue les comparaisons avec x_{j+1} et x_{j+2}, autrement on continue avec x_j et x_{j+2}. Lorsque toute la liste $LRP^{(i)}$ a été parcourue, on a construit $LRP^{(i+1)}$ triée selon l'ordre $<_{i+1}$. Puisque $LRP^{(i)}$ contient au maximum $|\mathcal{R}(n)|$ ruptures, le tri est réalisé en temps $O(\mathcal{R}(n))$. □

Remarque 3.4 *Le lemme de classement permet simplement de "reclasser" une liste de ruptures potentielles selon l'ordre $<_{i+1}$ lorsque le classement selon l'ordre $<_i$ est connu. Il ne permet pas de savoir si d'autres modifications de la liste ne vont pas se produire lors du passage de i à $i+1$. C'est le but de la section suivante.*

3.5.4 Ajout d'une nouvelle rupture dans *LRP*

Lorsque l'on passe de la position courante à la position suivante, il faut envisager l'ajout d'une nouvelle rupture potentielle dans $LRP^{(i)}$. Puisqu'on exige qu'une rupture appliquée ait un domaine de largeur minimale égale au retard $r_{\mathcal{R}}$ de la fonction de rupture $\mathcal{R}$, on envisage l'ajout de l'origine $i - r_{\mathcal{R}}$ dans LRP à l'étape i. Nous devons assurer que les quatre conditions pour que $i - r_{\mathcal{R}}$ appartienne à $LRP^{(i)}$ soient satisfaites (voir section 3.5.2). L'ajout peut être réalisé en temps $O(\mathcal{R}(n))$ comme le montre le lemme suivant.

Lemme 3.14 (d'ajout) *L'ajout éventuel de $x = i - r_{\mathcal{R}}$ dans $LRP^{(i)}$ est réalisé en temps $O(\mathcal{R}(n))$.*

Preuve. Pour vérifier la première condition d'appartenance à $LRP^{(i)}$, l'ajout n'est envisagé que si $f^{(i)}(x)$ existe. Il faut évidemment que $x \geq 0$.

Soit $LRP^{(i)} = (x_1, x_2, \ldots, x_k)$ avec x_k l'origine de la plus grande rupture pour l'ordre $<_i$. Pour chaque origine $x_j \in LRP^{(i)}$, on a $x_j < x$ puisqu'elle a été introduite à l'étape $x_j + r_{\mathcal{R}} \leq i = x + r_{\mathcal{R}}$. Dès lors, si $\mathcal{R}_{f^{(i)}x} <_i \mathcal{R}_{f^{(i)}x_k}$ alors x ne peut pas être introduit dans $LRP^{(i)}$ car la quatrième condition d'appartenance à $LRP^{(i)}$ n'est pas satisfaite.

Autrement, si $\mathcal{R}_{f^{(i)}x_k} <_i \mathcal{R}_{f^{(i)}x}$ alors x doit être introduit dans $LRP^{(i)}$ à la dernière position : elle devient la plus grande rupture pour l'ordre $<_i$. De plus, d'après la proposition 3.2, toutes les abscisses $x_j \in LRP^{(i)}$ telles que $f^{(i)}(x_j) \leq f^{(i)}(x)$ doivent être supprimées de $LRP^{(i)}$ pour satisfaire la troisième condition d'appartenance.

L'ajout est effectué en une seule comparaison des ruptures de $LRP^{(i)}$ avec la nouvelle introduite. Puisque $\#LRP^{(i)} \leq |\mathcal{R}(n)|$, l'ajout est réalisé en temps $O(\mathcal{R}(n))$. □

3.5.5 Algorithme formel

Cette section présente formellement la version finale de notre algorithme TURBOOPTLIFT. Nous décrivons d'abord la version intermédiaire NEWOPTLIFT qui exploite les lemmes d'application directe, de simplification, de classement et d'ajout. Ce sont les seules modifications apportées par rapport à l'algorithme OPTLIFT. En plus de ces modifications, l'algorithme

TURBOOPTLIFT évite la génération inutile des courbes intermédiaires $f^{(i)}$. Cette amélioration le rend particulièrement efficace (voir section 3.5.6).

Grâce à la liste des ruptures potentielles et à ses propriétés, l'algorithme NEWOPTLIFT présenté à la figure 3.14 optimise plus efficacement la courbe f.

```
NEWOPTLIFT(f, R)
 1  f^(0) ← f
 2  LRP^(0) ← NIL
 3  Pour i ← 0 Jusque n
 4  Faire x ← i − r_R
 5        Si x ≥ 0 et f^(i)(x) défini
 6           Alors LRP^(i) ← AJOUTERUPTURE(LRP^(i), x, f^(i))
 7        Si LRP^(i) ≠ NIL et f^(i)(i) défini
 8           Alors x_i ← PLUSGRANDERUPTURE(LRP^(i), i)
 9                 Si R_{f^(i)_{x_i}}(i) > f^(i)(i)      /* La rupture R_{f^(i)_{x_i}} réduit f^(i) sur [x, i] */
10                    Alors f^(i+1) ← LIFTING(f^(i), R, x_i, i)
11                    Sinon f^(i+1) ← f^(i)
12           Sinon f^(i+1) ← f^(i)
13        LRP^(i+1) ← TRILRP(LRP^(i), i + 1)
14  Retourner f^(n+1)
```

FIG. 3.14 - *Algorithme amélioré d'optimisation*

Il s'agit de l'adaptation de l'algorithme OPTLIFT qui applique, pour chaque position i, la plus grande rupture potentielle pour l'ordre $<_i$ si elle réduit $[0, i]$.

Dans cet algorithme :

- AJOUTERUPTURE$(LRP^{(i)}, x, f^{(i)})$ tente l'ajout de la nouvelle position x d'origine de rupture potentielle dans $LRP^{(i)}$ en suivant le schéma décrit par le lemme d'ajout.
- PLUSGRANDERUPTURE$(LRP^{(i)}, i)$ retourne l'origine de la plus grande rupture de la liste $LRP^{(i)}$ pour l'ordre $<_i$.
- LIFTING$(f^{(i)}, \mathcal{R}, x_i, i)$ est la même fonction de lifting que celle utilisée dans l'algorithme OPTLIFT. Elle effectue le lifting $f^{(i)} \xrightarrow{\mathcal{R}_{[x_i,i]}} f^{(i+1)}$.
- TRILRP$(LRP^{(i)}, i + 1)$ construit la liste $LRP^{(i+1)}$ triée selon l'ordre $<_{i+1}$ à partir de la liste $LRP^{(i)}$ triée selon l'ordre $<_i$. Cette fonction suit le schéma décrit par le lemme de classement.

Grâce au lemme d'application directe (lemme 3.11), nous savons que NEWOPTLIFT produit la même courbe que OPTLIFT, c'est-à-dire l'unique courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles.

Nous allons maintenant améliorer encore cet algorithme en évitant de construire les courbes intermédiaires $f^{(i)}$. Un lifting peut en effet être effectué de manière formelle sans avoir à construire la nouvelle courbe. Décrivons comment.

Remarquons tout d'abord que toutes les listes de ruptures potentielles $LRP^{(i)}$ ne doivent pas être conservées : lorsque $LRP^{(i+1)}$ a été construite, on peut oublier $LRP^{(i)}$. Dès lors, on ne considère qu'une seule liste LRP que l'on actualise à chaque étape.

Soit $TabLift[0 \rightarrow n - r_{\mathcal{R}}]$ le tableau d'entiers destiné à contenir les positions de fin de toutes les ruptures appliquées au long de l'algorithme:

- $TabLift[x] = y$ si la rupture de domaine $[x, y]$ a été appliquée.
- $TabLift[x] = 0$ si aucune rupture commençant à l'abscisse x n'a été appliquée.

Étant donné la courbe initiale f, le tableau *TabLift* après optimisation et la fonction de rupture $\mathcal{R}$, il est facile de construire la courbe optimale. Il suffit de parcourir les abscisses x du tableau *TabLift* de 0 à $n - r_{\mathcal{R}}$, à chaque fois que la valeur $y = TabLift[x]$ est non nulle, on applique la rupture sur $[x, y]$ et on continue à parcourir *TabLift* à partir de l'abscisse y.

Avant de commencer l'optimisation, le tableau *TabLift* doit être initialisé avec des 0 pour chaque abscisse, ce qui indique qu'aucune rupture n'a encore été appliquée. À chaque fois qu'une rupture de domaine $[x, y]$ est appliquée, la mémorisation du lifting consiste à placer y à la position x du tableau *TabLift*.

Bien entendu, chaque lifting translate vers le haut la partie droite de la courbe à partir de l'abscisse courante. Il faut en tenir compte pour continuer l'optimisation. Soit G la variable qui désigne le gain en lifting obtenu sur la partie de courbe déjà optimisée. Sa valeur est égale à la somme des gains en lifting de toutes les ruptures déjà appliquées. Initialement, $G = 0$. Étant donné l'abscisse courante i, $G + f(i)$ représente la valeur actualisée de $f(i)$ après optimisation de la courbe sur $[0, i - 1]$. Pour chaque abscisse i, la valeur de $f(i)$ est actualisée pour tenir compte de ce gain en lifting: $f(i) \leftarrow f(i) + G$. De cette manière, l'introduction d'une nouvelle rupture potentielle d'origine $x = i - r_{\mathcal{R}}$ dans *LRP* se fait en utilisant la valeur actualisée de $f(x)$.

Grâce à ces remarques, nous pouvons maintenant préciser la version finale de notre algorithme: TURBOOPTLIFT. Il est présenté à la figure 3.15. Il retourne le tableau *TabLift* des liftings.

Remarque 3.5 *L'algorithme* TURBOOPTLIFT *modifie la courbe initiale f pour certaines abscisses. La valeur de f après optimisation n'est donc plus cohérente, il faut en conserver une copie effectuée avant l'optimisation.*

3.5.6 Étude de complexité

Nous montrons ici que la complexité en temps de l'algorithme TURBOOPTLIFT est en $O(n\mathcal{R}(n))$ et que, dans le pire des cas du choix de la fonction de rupture $\mathcal{R}$, la complexité est en $O(n^2)$. La complexité en espace de TURBOOPTLIFT est en $O(n + |\mathcal{R}(n)|) = O(n)$

Théorème 3.15 *Soit $f \in \mathcal{F}^n$ et $\mathcal{R}$ la courbe de rupture. L'algorithme* TURBOOPTLIFT *calcule l'unique solution $f^* \in Lift^*_{\mathcal{R}}(f)$, optimale irréductible et minimale en rupture parmi les solutions optimales irréductibles, en temps $O(n\mathcal{R}(n))$ et en espace $O(n + |\mathcal{R}(n)|)$.*

Preuve. L'algorithme comporte exactement $n + 1$ étapes. Majorons le temps de chaque étape.

Puisque *LRP* est toujours classée selon l'ordre $<_i$, le travail de PLUSGRANDERUPTURE est réalisé en temps $O(1)$. À l'exception des appels aux fonctions AJOUTERUPTURE et TRILRP, toutes les instructions réalisées au cours d'une étape sont effectuées en temps constant. Grâce au lemme d'ajout, nous savons que AJOUTERUPTURE fonctionne en temps $O(\mathcal{R}(n))$. De même, le lemme de classement montre que TRILRP fonctionne en temps $O(\mathcal{R}(n))$.

```
TurboOptLift(f, R)
 1  LRP ← Nil
 2  G ← 0                                              /* Gain actuel en lifting */
 3  Pour i ← 0 Jusque n
 4  Faire TabLift[i] ← 0
 5        x ← i − r_R
 6        Si x ≥ 0 et f(x) défini
 7           Alors LRP ← AjouteRupture(LRP, x, f)
 8        Si f(i) défini
 9           Alors f(i) ← f(i) + G                    /* Valeur actualisée de f(i) */
10                 Si LRP ≠ Nil
11                    Alors x_i ← PlusGrandeRupture(LRP, i)
12                          Si R_{f x_i}(i) > f(i)   /* La rupture R_{f x_i} réduit f sur [x, i] */
13                             Alors G ← G + R_{fx}(i) − f(i)      /* Nouveau gain */
14                                   f(i) ← R_{fx}(i)              /* Valeur optimisée de f(i) */
15                                   TabLift[x_i] ← i              /* Mémorisation du lifting */
16        LRP ← TriLRP(LRP, i + 1)
17  Retourner TabLift
```

FIG. 3.15 - *Algorithme final* TURBOOPTLIFT

Chaque étape de l'algorithme est donc réalisée en temps $O(\mathcal{R}(n))$. Le temps total d'exécution de TURBOOPTLIFT est en $O((n+1)\mathcal{R}(n)) = O(n\mathcal{R}(n))$.

La complexité en espace est de $O(n+|\mathcal{R}(n)|)$ car la taille de *TabLift* est en $\theta(n)$ et la taille de *LRP* en $O(\mathcal{R}(n))$. □

La définition des fonctions DCL impose certaines conditions sur leurs décroissances (voir la définition 2.1 dans le chapitre 2). Le corollaire 3.16 montre que, dans le pire des cas pour le choix de la fonction de rupture, TURBOOPTLIFT fonctionne en temps $O(n^2)$.

Corollaire 3.16 *Dans le pire des cas pour le choix de la fonction de rupture $\mathcal{R}$, l'algorithme* TURBOOPTLIFT *a un temps d'exécution en $O(n^2)$.*

Preuve. Puisque les fonctions DCL sont décroissantes, convexes et de décroissance limitée à 1 à partir de leur retard, dans le pire des cas $\mathcal{R} = \theta(n)$. En effet, si $\mathcal{R}$ décroissait plus rapidement, elle serait concave : les marches devraient être de plus en plus courtes.

Lorsque toutes les marches de la fonction $\mathcal{R}$ ont la même longueur (fonction f_1 de l'exemple 2.1 du chapitre 2), ce cas limite est atteint : $\mathcal{R} = \theta(n)$. La complexité en temps de TURBOOPTLIFT est alors en $O(n^2)$. □

Puisque dans tous les cas $\mathcal{R} = O(n)$, la complexité en espace de TURBOOPTLIFT est $O(n)$.

La troisième partie de ce travail illustre l'utilisation de TURBOOPTLIFT dans le domaine de la compression de séquences. La courbe à optimiser est la courbe de compression issue de la phase de codage et la fonction de rupture est le coût de déclaration, de manière auto-délimitée, de la longueur d'un facteur que l'on décide de laisser tel quel, c'est-à-dire de ne pas comprimer (voir l'exemple présenté dans le premier chapitre de cette partie). Si le codage auto-délimité des longueurs de facteurs se fait en utilisant un code ICL, la fonction de rupture $\mathcal{R}$ est DCL.

De plus, si le codage est universel alors $\mathcal{R} = \theta(\log n)$ et l'algorithme TURBOOPTLIFT optimise la courbe en temps $O(n \log n)$.

Chapitre 4

Problèmes annexes

Nous présentons ici trois problèmes non résolus directement inspirés du problème principal d'optimisation. Le premier problème présente la recherche de la courbe optimale maximale en rupture. Ce cas est intéressant dans le domaine de la compression de séquences. Le second problème est la généralisation au cas continu de l'optimisation des courbes par liftings : les courbes de ruptures sont des fonctions CDC (Continûment dérivable, Décroissantes et Convexes). Ce problème présente un joli cadre de travail dans lequel l'unicité des courbes optimales ne pose pas de problème (on pourrait voir le chapitre précédent comme une discrétisation de ce problème). Nous terminons avec la généralisation des liftings pour construire une courbe optimale à partir de portions sélectionnées d'un nombre fini de courbes alternatives données. C'est un problème difficile. Il est potentiellement intéressant lorsque l'on dispose d'un nombre fini de solutions à un problème et que l'on désire les composer pour trouver une solution optimale.

4.1 Solution optimale maximale en rupture

Étant donné $f \in \mathcal{F}^n$ et $\mathcal{R}$ la fonction de rupture, il peut être intéressant de construire une courbe optimale maximale en rupture, c'est-à-dire qui possède un maximum de points dans des portions de ruptures.

Prenons l'exemple de la compression de séquences. La rupture permet de rompre le schéma de codage pour copier un facteur en sortie sans aucune compression (voir section 1.1 du chapitre 1). Appelons un tel facteur un *facteur de rupture*. Considérons une courbe optimale $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ maximale en rupture. Cette courbe est intéressante car la compression qu'elle représente conserve un maximum de symboles de la séquence originelle. On peut alors envisager l'utilisation d'une autre méthode de compression sur la séquence obtenue par concaténation de tous les facteurs de ruptures.

Remarque 4.1 *Cette façon de faire peut conduire à un programme de compression très puissant capable d'exploiter une méthode là où elle est intéressante et d'exploiter une autre là où elle ne l'est pas.*

Malheureusement, la forme discrète des courbes de rupture nous empêche de définir une unique courbe optimale maximale en rupture. En effet, considérons l'exemple de la figure 4.1. Si la courbe $\overline{f}$ optimale maximale en rupture a $[x, y]$ comme portion de rupture, alors il existe une autre courbe $\overline{f}' \in Lift^*_{\mathcal{R}}(f)$ optimale maximale en rupture contenant $[x', y']$ comme

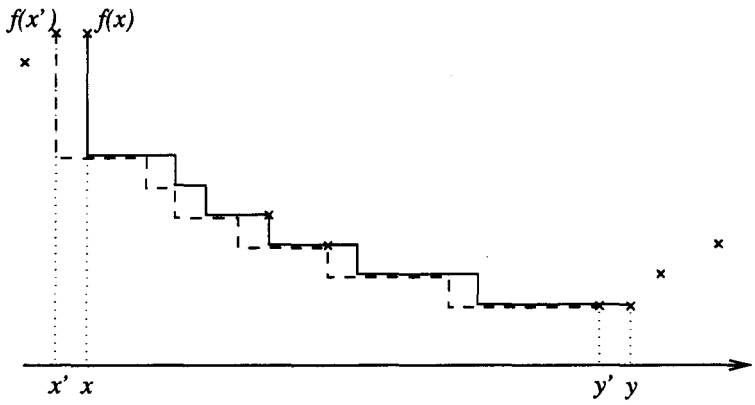


FIG. 4.1 - *Cas maximal en rupture*

portion de rupture (pour cela, il faut que $y' - x' = y - x$, $f(x) = f(x')$, $f(y) = f(y')$ et y (resp. y') est une abscisse de fin de marche de $\mathcal{R}_{fx}$ (resp. $\mathcal{R}_{fx'}$) : $\mathcal{R}_{fx}(y) = \mathcal{R}_{fx}(y+1) + 1$ (resp. $\mathcal{R}_{fx'}(y') = \mathcal{R}_{fx'}(y'+1) + 1)$). La courbe f est "optimisée" (sans gain) de deux façons en courbes maximales en rupture. C'est la forme discrète des courbes de rupture qui en est la cause. Il n'est donc pas possible de définir une unique courbe optimale maximale en rupture.

Une idée simple pour résoudre ce problème serait de partir de l'unique courbe optimale irréductible $f^{\mathcal{O}}$ et minimale en rupture parmi les courbes optimales irréductibles, fournie par TURBOOPTLIFT, et de lui ajouter ou de rallonger des portions de ruptures lorsque c'est possible. Puisque $f^{\mathcal{O}}$ est optimale et irréductible, aucune rupture potentielle ne réduit une portion $[x, y]$ de $f^{\mathcal{O}}$. Toutefois, si la portion $[x, y]$ de $f^{\mathcal{O}}$ contient une portion d'application $[x', y']$ et est telle que $f^{\mathcal{O}}(y) = \mathcal{R}_{f \circ_x}(y)$, alors l'application de la rupture $\mathcal{R}_{f \circ_x}$ le long de $[x, y]$ conserve l'optimalité de la courbe mais augmente le nombre de points situés sur des portions de rupture.

Le problème est de déterminer la suite de ruptures à appliquer à $f^{\mathcal{O}}$ pour rendre la courbe maximale en ruptures.

4.2 Cas continu

Le problème d'optimisation de courbes par liftings peut être étendu au cas continu. La fonction f à optimiser est une fonction partielle définie sur un sous-intervalle $[0, t]$ de $[0, +\infty[\subset \mathbb{R}$. On parle alors de fonction de rupture CDC (Continûment dérivable, Décroissante et Convexe).

Définition 4.1 *Une fonction* $g : [0, +\infty[\longrightarrow \mathbb{R}$ *est* CDC *(Continûment dérivable, Décroissante et Convexe) si les conditions suivantes sont vérifiées.*

1. *g est partout définie.*
2. $g(0) = 0$
3. *Il existe $r > 0$ telle que la fonction est constante et négative sur $]0, r]$ et à partir de r est continûment dérivable, décroissante et convexe, c'est-à-dire :*
   - $\forall x \in]0, r] : g(x) = -C$, *avec* $C > 0$.

- $\forall x > r$:
  - *g est continûment dérivable : la dérivée* $g'(x)$ *existe.*
  - *g est décroissante :* $g'(x) \leq 0$.
  - *g est convexe : la dérivée seconde* g'' *existe et* $g''(x) \geq 0$.

La constante r est appelée le retard de la fonction g. La figure 4.2 représente une fonction CDC.

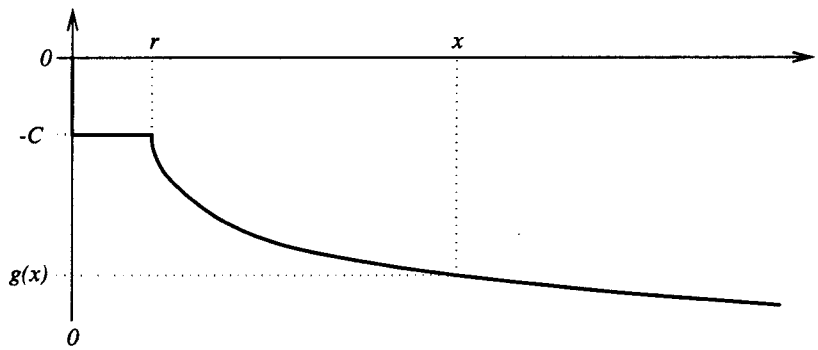


FIG. 4.2 - *Représentation d'une fonction CDC g*

Ce cadre de travail théorique est intéressant car il donne de plus beaux résultats que le cas discret. Le cas continu peut être vu comme le cas discret "vu de loin". On n'est pas obligé de définir l'ordre $<_i$ pour déterminer précisément le point d'intersection entre deux courbes de ruptures potentielles. Après leur intersection, les deux courbes ne se confondent pas.

Il doit être possible de démontrer l'unicité de la courbe optimale minimale en rupture ainsi que l'unicité de la courbe optimale maximale en rupture.

La figure 4.3 représente l'optimisation d'une fonction continue à l'aide d'une fonction CDC. Une rupture a été appliquée sur l'intervalle $[x, y]$. On voit que deux ruptures potentielles ont une intersection mieux définie que dans le cas discret. La rupture potentielle d'origine x' croise la rupture potentielle d'origine x et, de ce fait, devient moins bonne. L'obtention d'algorithmes d'optimisation imposerait des hypothèses restrictives sur les courbes manipulées.

4.3 Choix entre diverses alternatives de courbes

Le problème d'optimisation que nous avons considéré dans le chapitre précédent consistait en la décomposition de l'intervalle $[0, n]$ en sous-intervalles contigus et le choix sur chaque sous-intervalle $[x, y]$ entre deux alternatives : soit conserver la portion $[x, y]$ de la courbe donnée, soit choisir de la remplacer par un morceau de courbe de rupture $\mathcal{R}$.

Imaginons maintenant que l'on dispose de k courbes $f_1, f_2, \ldots f_k \in \mathcal{F}^n$. Le problème que l'on se pose est la décomposition de l'intervalle $[0, n]$ en sous-intervalles contigus et le choix sur chaque sous-intervalle $[x, y]$ de la portion $[x, y]$ d'une des k courbes $f_1, f_2, \ldots f_k$. Les portions de courbes ainsi sélectionnées composent une nouvelle courbe f de telle sorte que le dernier point d'une portion coïncide avec le premier point de la portion suivante. La figure 4.4 représente la composition d'une courbe f à partir des portions des trois courbes alternatives f_1, f_2, f_3. Le but de la composition est la construction d'une courbe $\overline{f}$ dont l'ordonnée en n est maximale. On dit alors que $\overline{f}$ est la *courbe composée optimale.*

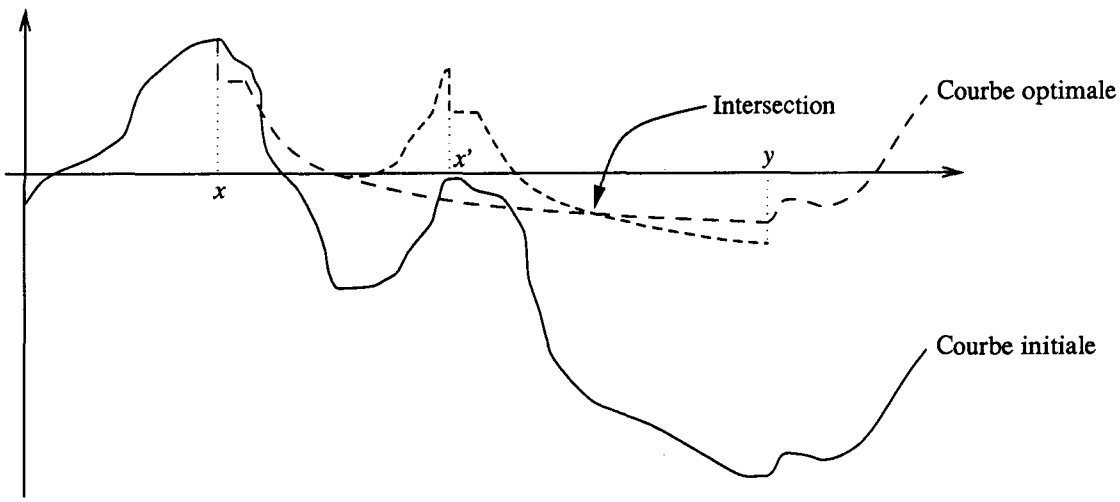


FIG. 4.3 - *Optimisation d'une courbe dans le cas continu*

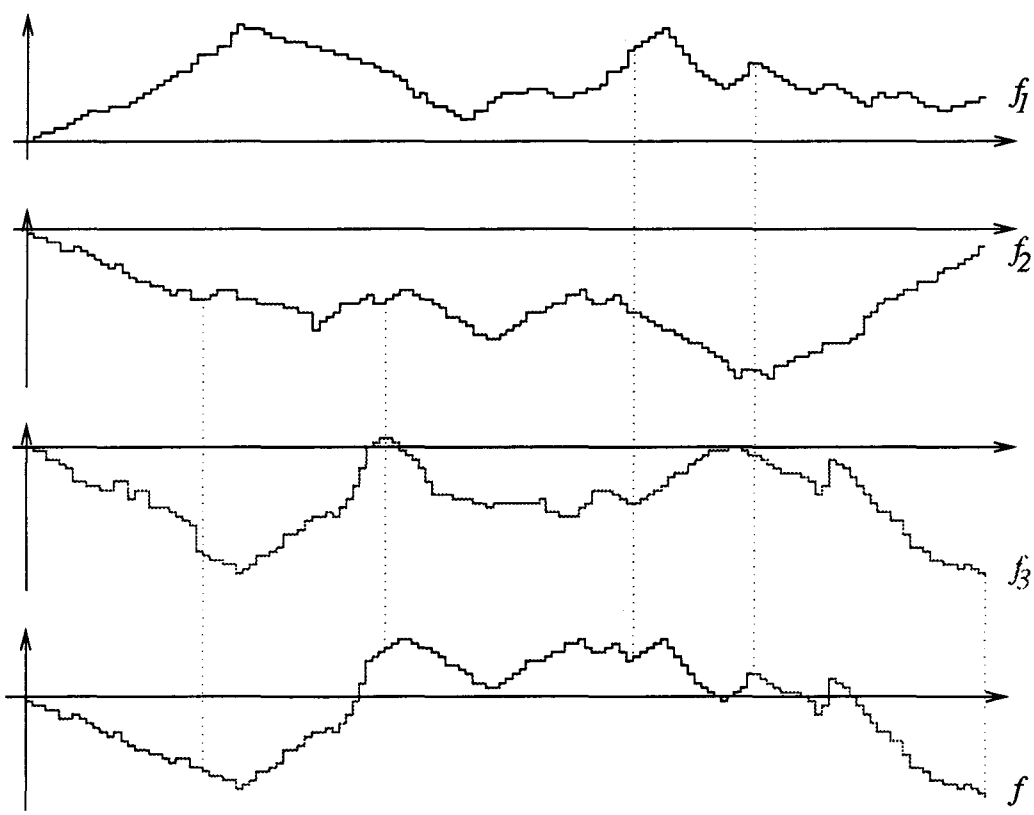


FIG. 4.4 - *Composition des trois courbes alternatives* f_1, f_2, f_3

Les courbes $f_1, f_2, \ldots f_k$ représentent k solutions à un même problème et la courbe composée optimale est la meilleure solution obtenue par composition des k solutions.

En pratique, les k courbes ne sont pas définies partout car une solution ne peut pas être décomposée en sous-intervalles de n'importe quelle manière. Il faut en tenir compte pour l'optimisation: les courbes sont partielles et à chaque fois qu'une portion $[x, y]$ est extraite d'une courbe f_i pour composer la courbe f, $f_i(x)$ et $f_i(y)$ doivent exister.

D'autre part, le passage d'une portion $[x, y]$ de la courbe f_i à la portion $[y + 1, y']$ de la courbe f_j engendre un coût qui abaisse légèrement la partie de la courbe située à droite de l'abscisse y. Appelons ce coût le *coût de changement.*

La figure 4.5 représente la courbe composée optimale $\overline{f}$ construite à partir des quatre

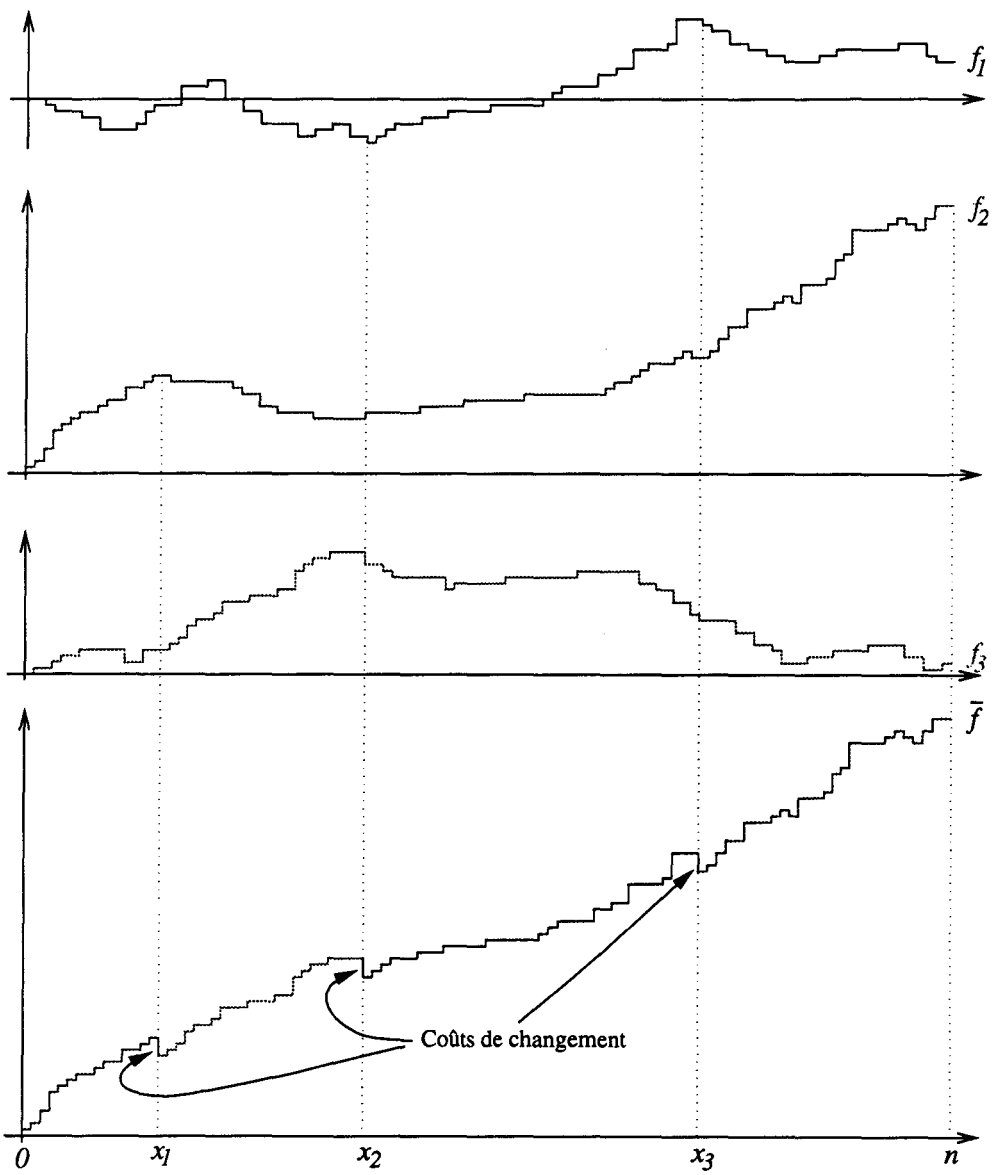


FIG. 4.5 - *Courbe composée optimale* $\overline{f}$

portions $[0, x_1], [x_1 + 1, x_2], [x_2 + 1, x_3]$ et $[x_3 + 1, n]$ extraites respectivement des courbes

f_2, f_3, f_1, f_2.

Dans le domaine de la compression de séquences, ce problème est très intéressant car il permet de composer une courbe de compression optimale étant donné les courbes de compression de k méthodes différentes. Bien entendu, les schémas de codage de ces k méthodes doivent autoriser le passage à une autre méthode pour certaines des positions de la séquence. Chaque passage d'une méthode à une autre impose le codage de deux informations supplémentaires dans la séquence de sortie :

1. L'annonce de changement de méthode.

2. L'identification de la nouvelle méthode.

Dans chaque schéma de codage, l'annonce de changement de méthode doit être un mot réservé à cet usage. C'est le nombre de bits du codage de ces deux informations qui détermine les coûts de changement visualisés à la figure 4.5.

Ce problème n'est pas facile car il s'agit de choisir, par morceaux, parmi k courbes alternatives alors qu'il y en avait seulement deux dans le cas du problème 2.1 d'optimisation. Il faut cependant remarquer que le passage d'une portion à une autre engendre un coût fixe alors que dans le cas du problème 2.1, le coût était intimement lié à la longueur de la rupture.

Troisième partie

Application de l'Optimisation par Liftings à la Compression et à l'Analyse de Séquences Génétiques

Chapitre 1

Optimisation par liftings de méthodes de compression existantes

Souvent, une méthode de compression applique aveuglément le même schéma de codage d'un bout à l'autre de la séquence sans tenir compte des endroits où il est bien adapté et des endroits où il l'est moins. Lorsqu'il est bien adapté, il raccourcit les facteurs codés. Lorsqu'il ne l'est pas il les rallonge. La méthode d'optimisation de courbes par liftings peut être efficacement utilisée pour éviter l'application du schéma de codage sur les facteurs qu'il rallonge. Ces facteurs sont alors recopiés tels quels dans la séquence comprimée. Seule la phase de codage de la méthode de compression est concernée, la phase de détection de régularités n'est pas remise en question.

Dans ce chapitre, nous montrons comment une méthode de compression existante peut être adaptée en vue d'une optimisation par liftings. Nous mettons en évidence les problèmes qui se posent et les compromis qui doivent être trouvés.

Pour que l'adaptation soit possible, le schéma de codage de la méthode de compression doit être *modulaire*, c'est-à-dire que toute séquence doit être décomposable en facteurs de telle sorte que la séquence comprimée soit obtenue par concaténation des codages indépendants de ces facteurs. Cette contrainte est primordiale, elle permet de définir la *courbe de compression* du compresseur vis-à-vis d'une séquence. C'est cette courbe qui peut ensuite être optimisée par l'algorithme TURBOOPTLIFT développé dans la seconde partie de ce travail.

La première section est générale, elle concerne l'adaptation de n'importe quelle méthode de compression possédant un schéma de codage modulaire. Elle présente la façon de construire une courbe de compression. Elle mentionne les modifications éventuelles à apporter au schéma de codage pour disposer d'un mot code qui permet d'annoncer une *rupture de codage*. La construction de la fonction de rupture DCL est également discutée. L'équivalence entre le problème d'optimisation de courbes par liftings et l'optimisation d'une méthode de compression est alors mise en évidence.

La section suivante permet d'interpréter, dans le contexte de l'analyse de séquences, la courbe optimale calculée par l'algorithme TURBOOPTLIFT. Il s'agit de localiser précisément les facteurs réguliers et les facteurs non réguliers d'une séquence.

La section 1.3 présente l'optimisation par liftings du codage de Huffman et la section 1.4 présente l'optimisation de la méthode de compression LZ77 de Ziv et Lempel. Ces deux adaptations de méthodes classiques sont données à titre d'exemple. La majorité des méthodes de compression existantes peuvent facilement être adaptées de la même façon.

1.1 Adaptation d'une méthode existante

Cette section formalise l'illustration de l'utilité des liftings à la compression de séquences (section 1.1.1 de la partie II). Nous montrons comment modifier le schéma de codage d'une méthode existante pour permettre une optimisation par liftings. Notons que tous les schémas de codage ne peuvent pas être modifiés de la sorte. Ainsi, nous ne pouvons dire à l'heure actuelle si le codage arithmétique peut être adapté en vue d'une optimisation par liftings.

Soit $\mathcal{C}$ un compresseur et $S \in \mathcal{B}^n$ une séquence de longueur n à comprimer. La section 1.4 illustre ce qu'il faut faire si la séquence est exprimée sur un autre alphabet. La séquence comprimée est $S' = \mathcal{C}(S)$.

Pour pouvoir être adapté, le schéma de codage doit posséder la propriété de *modularité de codage*. Il doit aussi permettre la définition d'un mot code réservé, appelé *annonce de rupture*, permettant de signaler dans la séquence comprimée lorsque le schéma de codage est rompu.

Définition 1.1 *Nous dirons qu'un schéma de codage possède la propriété de* modularité de codage, *ou que le schéma de codage est* modulaire, *s'il est possible de décomposer la séquence comprimée S' en :*

$$S' = I\, code(S_{1..i_1}) code(S_{i_1+1..i_2}) \dots code(S_{i_k+1..n})$$

pour certaines positions $\{i_1, i_2, \dots i_k\}$.

Où :

- *I est la séquence de bits correspondant aux informations initiales du schéma de codage.*
- *$code(S_{i_j+1..i_{j+1}})$ est le facteur de S' obtenu par application du schéma de codage au facteur $S_{i_j+1..i_{j+1}}$ de S. Chaque mot $code(S_{i_j+1..i_{j+1}})$ est totalement indépendant des autres mots $code(S_{i_h+1..i_{h+1}})$.*

Les positions $i_1, i_2, \dots i_k$ où cette séparation de codage est possible sont appelées des positions de séparation.

Exemple 1.1

- *Le codage de Huffman d'ordre k est modulaire. Pour le codage de Huffman d'ordre 2 par exemple, $S' = I\, code(S_{1..3}) code(S_{4..6}) \dots code(S_{n-2..n})$*

  Où :

  - *Le mot I est le codage initial de l'arbre de Huffman.*
  - *Chaque mot $code(S_{i..i+2})$ est un mot code du codage de Huffman (voir section 2.3.1.4 de la partie I).*

- *Le schéma de codage de la méthode LZ77 est modulaire.*

  En effet : $S' = I\, code(S_{1..i_1}) code(S_{i_1+1..i_2}) \dots code(S_{i_k+1..n})$

  Où :

  - *I est le codage des tailles de la fenêtre et du tampon.*
  - *$code(S_{i_j+1..i_{j+1}})$ est le codage du triplet $(position, longueur, symbole)$ correspondant au facteur $S_{i_j+1..i_{j+1}}$ (voir partie I, page 54).*

L'optimisation de la compression d'une séquence par la méthode des liftings n'est possible que si le schéma de codage du compresseur est modulaire. Définissons dans ce cas la courbe à optimiser : la *courbe de compression.*

Définition 1.2 *La* courbe de compression *de C vis-à-vis de la séquence S est l'application partielle $f \in \mathcal{F}^n : x \mapsto f(x)$ définie par :*

- $f(0) = -|I|$ *: il s'agit du coût du codage des informations initiales.*
- $\forall i_j \in \{i_1, i_2, \ldots i_k\} : f(i_j) = |S_{..i_j}| - |I| - |code(S_{1..i_1})code(S_{i_1+1..i_2}) \ldots code(S_{i_{j-1}+1..i_j})| = i_j - |I| - |code(S_{1..i_1})code(S_{i_1+1..i_2}) \ldots code(S_{i_{j-1}+1..i_j})|$.

  Pour toute position de séparation i_j, $f(i_j)$ évalue le gain de compression partiel de C pour le préfixe $S_{..i_j}$.
- $f(n) = |S| - |S'|$ *: $f(n)$ est le gain de compression de C sur toute la séquence S.*

L'application f est définie uniquement pour ces positions.

Pour chaque intervalle $[i, j] \subseteq [0, n]$ tel que $f(i)$ et $f(j)$ sont tous deux définis, la modularité du schéma de codage nous autorise à "rompre" le codage sur toute la longueur du facteur $S_{i+1..j}$. Cela signifie qu'au lieu de coder le facteur comme l'impose le schéma de codage (une suite de mots $code(S_{....})$), on le recopie tel quel dans la séquence comprimée. Malheureusement, on ne peut pas se contenter de recopier le facteur vraiment tel quel : le programme de décompression s'attend à ce que le schéma de codage ait été appliqué sur toute la séquence. Dès lors, dans la séquence comprimée, il faut signaler que le codage est rompu et fournir le nombre de bits qui ont été recopiés tels quels.

On remplace, dans la séquence comprimée, la suite :

$$code(S_{i+1..i_a})\, code(S_{i_a+1..i_b}) \ldots code(S_{i_g+1..j})$$

où les indices $i_a, i_b, \ldots i_g$ sont toutes les positions de séparation situées entre i et j,
par la séquence :

$$a_{\mathcal{R}} AD(j-i)\, S_{i+1..j}$$

où :

- $a_{\mathcal{R}}$ est l'*annonce de rupture* qui signale, qu'à partir de la position $i+1$, le schéma de codage habituel est rompu. C'est un mot code qui doit être réservé à cet effet dans le schéma de codage. Il faut donc que le schéma de codage autorise son utilisation à la position $i+1$. Dans le cas où $a_{\mathcal{R}}$ ne peut pas être utilisé à cet endroit, on contourne le problème en interdisant à i d'être une position de séparation : nous convenons que $f(i)$ n'est pas défini.
- $AD(j-i)$ est le codage auto-délimité de l'entier $j-i$. Ce mot code signale au décompresseur la longueur de la rupture du codage.
- $S_{i+1..j}$ est le facteur recopié tel quel.

Cette rupture n'est intéressante que si elle augmente le gain en compression de la séquence. Soit G le gain produit par la rupture :

$$G = C - R - L$$

Avec :

- $C = |code(S_{i+1..i_a})\, code(S_{i_a+1..i_b}) \ldots code(S_{i_l+1..j})|$ est le nombre de bits que la rupture évite de coder.
- $R = |a_{\mathcal{R}}| + |AD(j-i)|$ est le *coût de la rupture* : c'est le nombre de bits qui ont dû être codés pour permettre au décompresseur de connaître l'étendue de la rupture du codage.
- $L = |S_{i+1..j}| = j - i$ est le coût de recopie du facteur.

La rupture transforme la courbe f en la courbe f' définie par :

- $\forall h \leq i$ et $f(h)$ défini : $f'(h) \leftarrow f(h)$.
- $\forall h \geq j$ et $f(h)$ défini : $f'(h) \leftarrow f(h) + G$.

Remarque 1.1 *D'après la définition 1.2 de la courbe de compression, on a $f'(j) = f'(i) - R$. La quantité R représente donc la différence de gain de compression partiel entre la position i et la position j dans le cas d'une rupture. S'il n'avait fallu coder ni $a_{\mathcal{R}}$ ni $AD(j-i)$, les valeurs $f'(i)$ et $f'(j)$ auraient été égales, ce qui aurait exprimé que recopier un facteur tel quel ne provoque ni gain, ni perte de compression.*

Le coût R de la rupture est indépendant de la séquence S. Il ne dépend que de la longueur $j - i$ de la rupture.

Imaginons maintenant que l'on ne rompe le schéma de codage que sur des facteurs de longueurs au moins égales à $r_{\mathcal{R}}$. Dans ce cas, il n'est pas nécessaire de coder la longueur $j - i$ de la rupture. On peut se contenter de coder $j - i - r_{\mathcal{R}}$ puisque aucun entier plus petit que $r_{\mathcal{R}}$ ne doit être codé. Le décompresseur est capable de retrouver $j - i$ à partir de $AD(j - i - r_{\mathcal{R}})$.

Le coût de la rupture peut être facilement calculé grâce à la *fonction de rupture.*

Définition 1.3 *Soit $\mathcal{R} : \mathbb{N} \to \mathbb{Z} : i \mapsto \mathcal{R}(i)$ la* fonction de rupture. *Elle est définie par :*

$$\begin{cases} \mathcal{R}(0) &= 0 \\ \mathcal{R}(i) &= -|a_{\mathcal{R}}| - |AD(0)| & \forall i : 0 < i < r_{\mathcal{R}} \\ \mathcal{R}(i) &= -|a_{\mathcal{R}}| - |AD(i - r_{\mathcal{R}})| & \forall i \geq r_{\mathcal{R}} \end{cases}$$

La valeur $-\mathcal{R}(l)$ est le coût de la rupture lorsque le facteur recopié tel quel est de longueur $l \geq r_{\mathcal{R}}$.

D'après la proposition 2.1 de la partie II, nous savons que si le code auto-délimité AD est ICL, alors la fonction $\mathcal{R}$ est DCL. De telles fonctions d'auto-délimitation existent :

- Le code *Fibo* est auto-délimité, ICL et universel (page 39).
- Le code *PrefFibo* est auto-délimité, ICL et asymptotiquement optimal (page 42).

Nous pouvons maintenant établir clairement le lien entre le problème d'optimisation résolu par l'algorithme TURBOOPTLIFT de la partie II et l'optimisation de la compression d'une séquence.

Étant donnés :

- Le compresseur C dont le schéma de compression est modulaire.
- La séquence $S \in \mathcal{B}^n$,
- La courbe de compression $f \in \mathcal{F}^n$ de C vis-à-vis de la séquence S (définition 1.2).
- La fonction de rupture $\mathcal{R}$, de retard $r_{\mathcal{R}}$, construite à partir d'un code auto-délimité ICL (définition 1.3).

Le problème :

"Rechercher l'ensemble des facteurs $\Sigma = \{S_{a..b}, S_{c..d}, \ldots\}$
qui doivent être recopiés tels quels dans la séquence comprimée
dans le but de maximiser le gain de compression global"

revient à :

"Rechercher $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ tel que $\overline{f}(n)$ soit maximal".

Il suffit donc d'exécuter TURBOOPTLIFT$(f, \mathcal{R})$. Chaque rupture appliquée par l'algorithme correspond à un facteur qui doit être recopié tel quel, précédé de l'annonce de rupture $\alpha_{\mathcal{R}}$ et du codage auto-délimité de sa longueur. Si le code auto-délimité utilisé pour coder les longueurs des ruptures est universel, l'algorithme fournit une solution optimale en un temps $O(n \log n)$.

Une fois l'optimisation effectuée, il suffit de parcourir le tableau *TabLift* retourné par la fonction TURBOOPTLIFT, pour les positions i allant de 0 à n, et de coder la séquence comprimée en sortie. Si la valeur $TabLif[i]$ est nulle, le schéma de codage habituel est appliqué. Si la valeur $TabLif[i]$ est non nulle, i est la position de début d'un facteur à recopier tel quel. La valeur $TabLift[i]$ est la position de fin du facteur. L'annonce de rupture $\alpha_{\mathcal{R}}$ est codée, suivie de la longueur de la rupture de manière auto-délimitée : $AD(TabLift[i] - i + 1 - r_{\mathcal{R}})$ suivie du facteur $S_{i..TabLift[i]}$.

Lors du décodage, lorsque l'annonce de rupture $\alpha_{\mathcal{R}}$ est détectée, il suffit de décoder la longueur de la rupture, de lire sans conversion le facteur qui a été codé tel quel et ensuite de reprendre le décodage normal.

Remarque 1.2 *Dans cette section, nous avons supposé que le schéma de codage nous fournissait un mot code $\alpha_{\mathcal{R}}$ pour annoncer les ruptures. Bien entendu, les méthodes de compression conçues sans envisager l'optimisation à l'aide de* TURBOOPTLIFT *ne disposent pas d'un tel mot code. C'est le cas de la méthode de Huffman développée dans la section suivante.*

Dans ce cas, le schéma de codage originel doit être modifié pour disposer d'un tel mot code. Puisque le schéma de codage d'une méthode de compression est étudié pour minimiser autant que possible les redondances dans la version comprimée, il est rare que l'on puisse ajouter un mot code sans provoquer une légère chute de performances du codage. L'optimisation ne peut malheureusement se faire qu'à ce prix. La section suivante montre que l'ajout d'un nouveau mot code dans un arbre de Huffman allonge au moins un des mots code existant.

1.2 Interprétation des résultats de l'optimisation

L'optimisation d'une courbe de compression par liftings permet non seulement d'optimiser le gain en compression global mais elle permet également une analyse fine des régularités.

Prenons un compresseur $\mathcal{C}$ qui exploite le type de régularités R. Soit $S \in \mathcal{B}^n$ une séquence de longueur n et $S' = \mathcal{C}(S) \in \mathcal{B}^+$ sa version comprimée à l'aide du compresseur $\mathcal{C}$. D'une façon globale, nous pouvons déjà conclure et analyser la présence des régularités R dans S. Pour cela, on regarde la valeur du gain en compression : $gain = n - |S'|$. S'il est positif, la séquence S est "globalement régulière" pour R. Le taux de compression est $taux = \frac{gain}{n}$ mesure le taux de régularité de S.

Jusque là, rien n'est nouveau. Il s'agit de l'utilisation de la compression comme outil d'analyse, motivée par la théorie de la complexité de Kolmogorov (voir la section 2.4, partie I ou la thèse de doctorat d'Eric Rivals [Riv96]).

Soit $\overline{f} \in Lift^*_{\mathcal{R}}(f)$, la courbe optimale calculée par l'algorithme TURBOOPTLIFT. Supposons que la portion $[x, y]$ de $\overline{f}$ soit une portion de rupture : $\forall i \in [x, y] : \overline{f}(i) = \overline{f}(x) + \mathcal{R}(i - x)$ et l'abscisse $y + 1$ est sur une portion d'application. Du point de vue de la compression, cela signifie qu'il est préférable de recopier le facteur $S_{x+1..y}$ tel quel, précédé de l'annonce de rupture $a_{\mathcal{R}}$ et du codage auto-délimité de sa longueur $y - x$, plutôt que de lui appliquer le schéma de compression de $\mathcal{C}$. Le facteur $S_{x+1..y}$ ne possède donc pas suffisamment de régularités du type R pour provoquer une hausse du gain de compression.

Puisque $\overline{f} \in Lift^*_{\mathcal{R}}(f)$ est l'unique courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles, les portions de ruptures et les portions d'applications sont choisies de manière optimale. Elles ne peuvent être ni allongées, ni raccourcies tout en conservant la courbe optimale irréductible et minimale en rupture parmi les courbes optimales irréductibles. Cela signifie que la décomposition de la séquence en facteurs "réguliers" et facteurs "non réguliers" est très précise : le moindre changement de longueur d'un facteur régulier modifie la caractéristique de la courbe.

On peut envisager de soumettre les facteurs non réguliers à une autre méthode de compression $\mathcal{C}'$ exploitant un autre type de régularités R'. Les zones de ruptures pour $\mathcal{C}'$ correspondent à des facteurs non réguliers que l'on peut soumettre à un troisième compresseur et ainsi de suite... Cette analyse en série d'une séquence à l'aide de plusieurs compresseurs différents est très intéressante car elle permet de déterminer précisément quels sont les facteurs réguliers d'une séquence et pour chacun d'eux, le type de régularités qui le caractérise. Dans cette optique, la maximisation en rupture de la solution optimale serait intéressante car à égalité de compression, elle considère un maximum de facteurs comme non réguliers (voir le chapitre 4 de la partie II). Ces facteurs non réguliers sont alors soumis au deuxième compresseur qui peut les considérer comme réguliers, et ainsi de suite... Le gain de compression global d'une telle itération de compresseurs est plus intéressant que dans le cas minimal en rupture car il est possible de trouver plus de régularités.

1.3 Codage de Huffman

Nous formalisons et généralisons ce qui a été introduit dans la présentation de la problématique des liftings concernant le codage de Huffman (partie II, section 1.1.2). Le schéma de codage doit être modifié pour permettre l'utilisation d'un nouveau mot code comme annonce de rupture. Nous ne considérons que la version statique du codage de Huffman. La version

dynamique est plus délicate car l'arbre de Huffman utilisé pour le codage n'est pas le même du début à la fin de la compression. Il est donc difficile de décider une fois pour toutes d'un mot code qui doit servir d'annonce de rupture durant toute la compression (la forme de la courbe de rupture est figée d'un bout à l'autre de la phase d'optimisation).

Considérons le codage de Huffman d'ordre $k-1$. L'ensemble des mots source est $E_S = \mathcal{B}^k$.

Soit p la distribution de probabilité des mots de l'ensemble source E_S:

$$\forall w \in E_S : 0 \leq p(w) \leq 1 \text{ et } \sum_{w \in E_S} p(w) = 1$$

L'arbre de Huffman correspondant à la distribution de probabilité p est construit selon le processus décrit à la section 2.3.1.2.2 du chapitre 2 de la première partie. On obtient un code auto-délimité optimal dont le trie est un arbre complet. Pour tout mot $w \in \mathcal{B}^k$, notons $Huff(w)$ le mot code correspondant.

Exemple 1.2 *L'illustration que nous développons au long de tous les exemples de cette section concerne un codage précis de Huffman d'ordre 2 :*

$$E_S = \mathcal{B}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Soit p la distribution de probabilité des mots source définie par :

$$\begin{array}{llll} p(000) = 0.027 & p(001) = 0.257 & p(010) = 0.1 & p(011) = 0.029 \\ p(100) = 0.086 & p(101) = 0.43 & p(110) = 0.043 & p(111) = 0.028 \end{array}$$

L'arbre de Huffman correspondant à cette distribution de probabilité est représenté à la figure 1.1. Par exemple, $Huff(110) = 11101$.

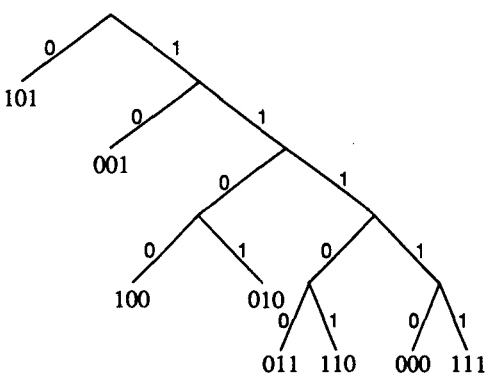


FIG. 1.1 - *Arbre de Huffman de l'ensemble $E_S = \mathcal{B}^3$ pour la distribution de probabilité p*

Le codage de Huffman est modulaire. La compression de la séquence S est la séquence

$$S' = I\, Huff(S_{1..k}) Huff(S_{k+1..2k}) \ldots Huff(S_{n-k+1..n})$$

où I est la séquence de bits permettant de coder l'arbre de Huffman.

Définissons la courbe de compression du codage de Huffman pour la séquence $S \in \mathcal{B}^n$ (n est un multiple de l'ordre k du codage).

La fonction $f \in \mathcal{F}^n$ est la courbe de compression du codage de Huffman vis-à-vis de la séquence S. Elle est définie par :

- $f(0) = -|I|$. On peut montrer que $\#E_S \log \#E_S$ bits suffisent pour coder l'arbre de Huffman [LH87]. Donc $f(0) = -|I| = -2^k k$.
- $f(i) = -2^k k + i - |Huff(S_{1..k})Huff(S_{k+1..2k}) \dots Huff(S_{i-k+1..i})|$, $\forall i : 0 < i \leq n$ et i multiple de k.
- $f(i)$ n'est pas défini si i n'est pas un multiple de k.

La fonction n'est donc définie qu'une seule position sur k car ce sont les seules positions où le codage peut être interrompu.

Exemple 1.3 *Pour notre exemple, considérons la séquence suivante de* 141 *bits :*

$S =$ 111 001 100 101 101 101 101 101 001 011 101 101 101 101 101 101 001 101 001 101 000 011 110 111 010 011 110 000 111 110 011 110 000 111 001 110 111 011 001 101 101 101 101 101 101 101

Nous avons souligné une zone sur laquelle le codage de Huffman n'est pas intéressant (voir plus loin).

Le nombre de bits pour coder l'arbre de Huffman est $24 \Rightarrow f(0) = -24$.

Les valeurs de la courbe sont :

$$f(0) = -24, f(3) = -26, f(6) = -25, f(9) = -26, f(12) = -24, \dots f(141) = -14$$

La courbe $f \in \mathcal{F}^{141}$ *est représentée à la figure 1.2.*

Il reste à choisir le mot a_R qui servira d'annonce de rupture. Puisque l'arbre de Huffman est complet, il n'est pas possible de lui ajouter un nouveau mot code tout en conservant le code auto-délimité (c'est une propriété des codes auto-délimités [BP85]). Le code de Huffman doit être légèrement modifié.

Étant donné la méthode de construction d'un arbre de Huffman, la propriété suivante est toujours vérifiée [Huf52].

Proposition 1.1 $\forall v, w \in E_S$: *si* $p(v) < p(w)$ *alors* $|Huff(v)| \geq |Huff(w)|$.

Sans Preuve.

Nous allons conserver cette propriété en allongeant uniquement un des mots code.

Soit $w_{min} \in E_S$ un mot source tel que :

$$\forall w \in E_S : p(w_{min}) \leq p(w) \text{ et } |Huff(w_{min})| \geq |Huff(w)|$$

C'est un mot source de probabilité minimale qui possède le plus long mot code. Grâce à la proposition 1.1, nous savons qu'un tel mot existe : parmi tous les plus longs mots code, il y en a un de probabilité minimale.

Considérons le nouveau code auto-délimité $h : E_S \to \mathcal{B}^+ : w \mapsto h(w)$ défini par :

$$\begin{aligned} h(w) &= Huff(w) && \text{si } w \neq w_{min} \\ h(w_{min}) &= Huff(w_{min})0 \end{aligned}$$

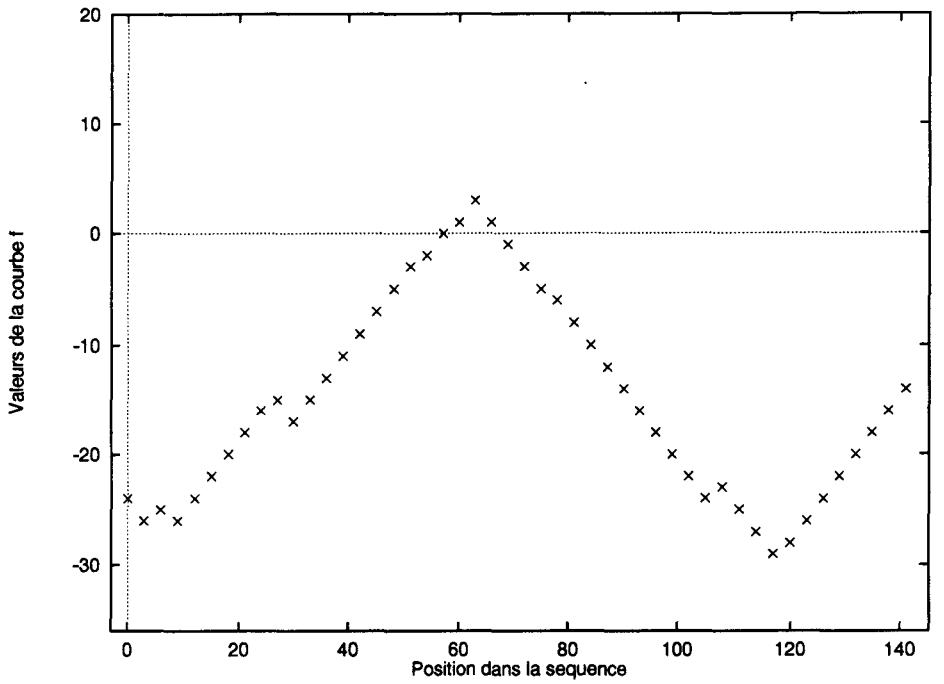


FIG. 1.2 - *Courbe f pour le codage de Huffman sur $E_S = \mathcal{B}^3$ de l'exemple 1.3*

Le code h est identique au code *Huff* si ce n'est que le mot code correspondant à w_{min} a été allongé de 1 bit.

Posons $a_{\mathcal{R}} = Huff(w_{min})1$. Par construction, $a_{\mathcal{R}}$ n'est le préfixe d'aucun mot code de h et aucun mot code de h n'est le préfixe de $a_{\mathcal{R}}$.

Si le nouveau schéma de codage utilise h au lieu d'utiliser le code *Huff*, alors le mot $a_{\mathcal{R}}$ peut être utilisé comme annonce de rupture. Bien entendu, l'utilisation de h au lieu de *Huff* fait chuter légèrement les performances mais il faut se rappeler que le mot code allongé est celui correspondant au mot source le moins probable. La chute de performance est donc négligeable.

Exemple 1.4 *Dans notre exemple, $w_{min} = 000$ et $Huff(w_{min}) = 11110$.*
L'annonce de rupture est donc $a_{\mathcal{R}} = 111101$ et $h(w_{min}) = 111100$.
Tous les autres mots code $h(w)$ sont identiques à $Huff(w)$ (voir figure 1.3).

La courbe de compression à utiliser pour l'optimisation est $f' \in \mathcal{F}^n$ construite de la même manière que la courbe f mais en utilisant le code h plutôt que le code *Huff*.

Exemple 1.5 *Pour notre exemple :*

$$f'(0) = -24, f'(3) = -26, f'(6) = -25, f'(9) = -26, f'(12) = -24, \ldots f'(141) = -17$$

Seulement 3 bits ont été ajoutés à la séquence comprimée car le mot source $w_{min} = 000$ n'apparaît que 3 fois dans S. La figure 1.4 représente la courbe f'.

Il reste maintenant à déterminer complètement la fonction de rupture $\mathcal{R}$ et son retard $r_{\mathcal{R}}$.

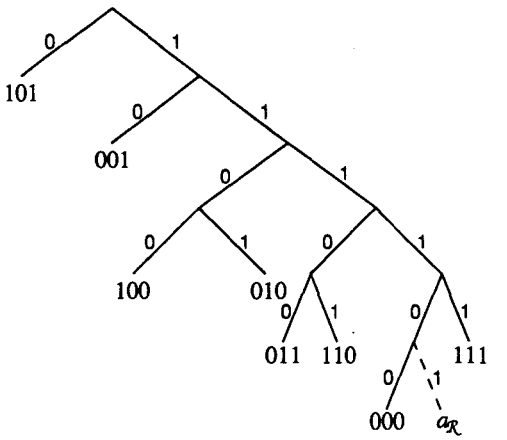


FIG. 1.3 - *Code auto-délimité h*

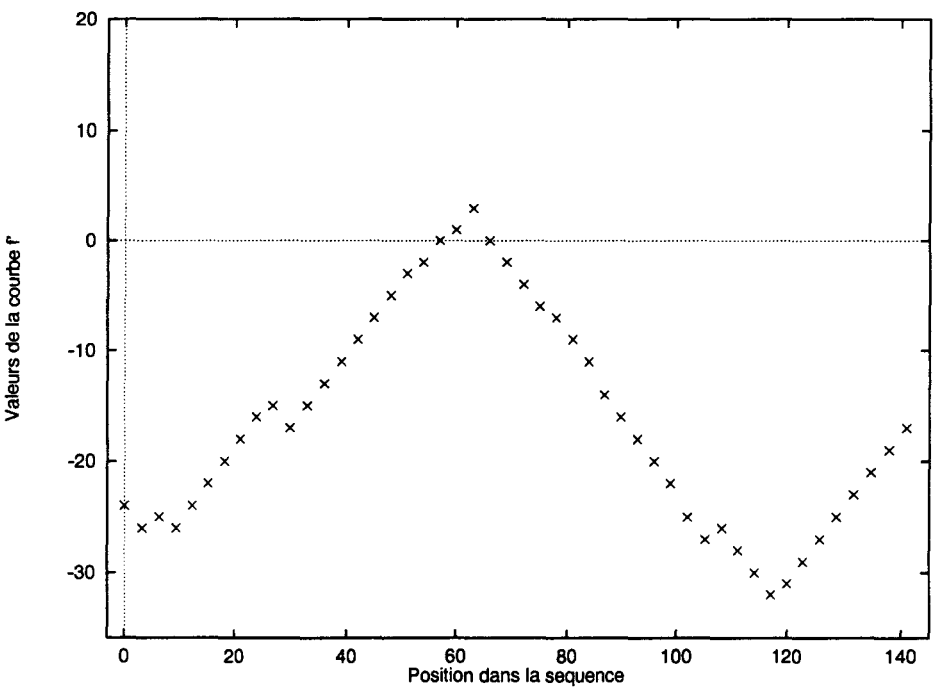


FIG. 1.4 - *Courbe f'*

Choisissons le code *PrefFibo* pour coder la longueur d'une rupture. Il est auto-délimité et ICL. On sait que la fonction f' n'est définie qu'une position toutes les k positions. Il n'est donc pas envisageable qu'une rupture soit d'une longueur inférieure à k bits. On constate également que la rupture ne peut pas porter sur exactement k bits (c'est-à-dire permettre de n'économiser qu'un seul mot code) car l'annonce de rupture est au moins aussi longue que chacun des mots code. Sur k bits, elle ne peut qu'engendrer une perte. Une rupture portera donc toujours sur un minimum de $2k$ bits.

Soit le retard $r_{\mathcal{R}} = 2k$.

Supposons qu'à la position i_j (i_j multiple de k) de la séquence S, on décide de rompre le codage sur l bits, alors le codage de la rupture est $a_{\mathcal{R}}\, PrefFibo(l - r_{\mathcal{R}})S_{i_j..i_j+l-1}$. Le coût de la rupture est $|a_{\mathcal{R}}| + |PrefFibo(l - r_{\mathcal{R}})|$.

La fonction de rupture $\mathcal{R} : \mathbb{N} \to \mathbb{Z} : i \mapsto \mathcal{R}(i)$ est définie par :

$$\begin{cases} \mathcal{R}(0) &= 0 \\ \mathcal{R}(i) &= -|a_{\mathcal{R}}| - |PrefFibo(0)| & \forall i : 0 < i < r_{\mathcal{R}} \\ \mathcal{R}(i) &= -|a_{\mathcal{R}}| - |PrefFibo(i - r_{\mathcal{R}})| & \forall i \geq r_{\mathcal{R}} \end{cases}$$

Exemple 1.6 *Pour notre exemple, le retard est* $r_{\mathcal{R}} = 6$ *et* $|a_{\mathcal{R}}| = 6$.

La fonction de rupture est $\mathcal{R} : \mathbb{N} \to \mathbb{Z} : i \mapsto \mathcal{R}(i)$ *est telle que :*

$$\begin{cases} \mathcal{R}(0) &= 0 \\ \mathcal{R}(i) &= -|a_{\mathcal{R}}| - |PrefFibo(0)| = -10 & \forall i : 0 < i < 6 \\ \mathcal{R}(i) &= -6 - |PrefFibo(i - 6)| & \forall i \geq 6 \end{cases}$$

Son graphique est représenté à la figure 1.5.

FIG. 1.5 - *Fonction de rupture* $\mathcal{R}$ *de retard* 6

L'appel TabLift $\leftarrow$ TURBOOPTLIFT$(f', \mathcal{R})$ *fournit la solution optimale représentée à la figure 1.6. La rupture correspond au facteur initialement souligné dans la séquence S. Elle porte sur* 54 *bits. Malgré la modification du schéma de codage, la courbe optimisée est meilleure que la courbe originelle (figure 1.2).*

Bien que l'optimisation n'ait pas réussi à produire un gain de compression final positif (en l'occurrence il est nul), elle a été très utile : elle a localisé précisément une zone sur laquelle le codage de Huffman n'est absolument pas adapté. Il s'agit de la zone de rupture.

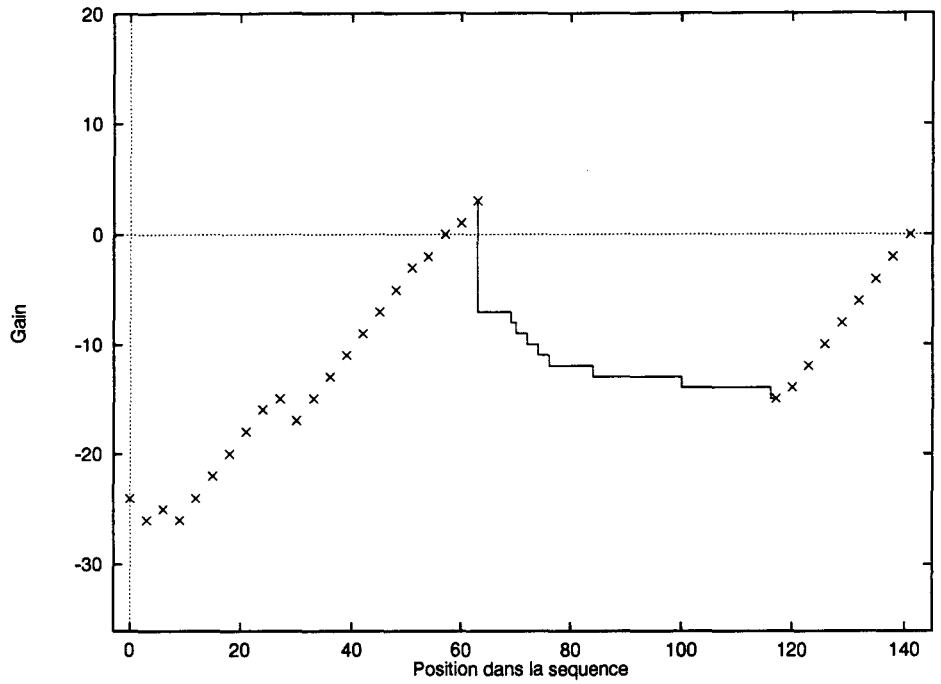


FIG. 1.6 - *Courbe f' après l'optimisation par liftings*

1.4 Méthode LZ77 de Ziv et Lempel

Les méthodes de compression LZ77 et LZ78 de Ziv et Lempel peuvent être optimisées grâce à l'algorithme TURBOOPTLIFT. À titre d'exemple, nous présentons l'adaptation à la méthode LZ77. Cette adaptation est plus aisée que l'adaptation du codage de Huffman car le schéma de codage ne doit pas forcément être modifié: l'annonce de rupture est un mot que le schéma de codage nous autorise à utiliser.

Nous nous en tenons aux méthodes de base exposées dans la première partie de ce travail, nous ne considérons pas leurs extensions et adaptations plus récentes.

Notons que l'optimisation de la méthode LZ78 se fait de manière similaire exception faite que le schéma de codage doit être légèrement modifié pour disposer d'un mot utilisable comme annonce de rupture. On peut par exemple décider de numéroter les facteurs du dictionnaire à partir de 2. L'utilisation du numéro de facteur 1 peut alors servir d'annonce de rupture (la méthode LZ78 est décrite dans le chapitre 2 de la partie I).

Soit $S \in \mathcal{A}^n$ la séquence à comprimer. L'alphabet $\mathcal{A}$ n'est pas l'alphabet binaire. Soit $k = \lceil \log \#\mathcal{A} \rceil$ le nombre de bits nécessaires pour coder chaque symbole de la séquence. Soit $S_{\mathcal{B}} \in \mathcal{B}^{k.n}$ la séquence binaire obtenue en codant chaque symbole de S sur k bits. La méthode LZ77 procède par détection et codage de facteurs répétés de longueurs variables. L'application de LZ77 à la séquence S n'est donc pas équivalente à l'application de LZ77 à la séquence $S_{\mathcal{B}}$ car les symboles de S ne peuvent pas être séparés en bits individuels. En pratique, les symboles sont souvent exprimés sur 8 bits et les ordinateurs permettent de les manipuler et de les comparer efficacement sans descendre au niveau du bit. Pour ces raisons, nous ne convertissons pas la séquence S sur $\mathcal{B}$ avant de lui appliquer la méthode LZ77. Dès lors, comme nous l'avons fait remarquer à la page 29 (remarque 2.1), il faut prendre garde à la manière de calculer le gain de compression, qu'il soit global ou partiel. La séquence comprimée

S' est une séquence binaire : $S' \in \mathcal{B}^+$. Le gain de compression est mesuré en bits. Il est obtenu par la formule : $gain = k.n - |S'|$.

Soit N la longueur de la fenêtre qui coulisse sur le texte et T la longueur du tampon. Soient $e_1 = \lceil \log N \rceil$ et $e_2 = \lceil \log N - T \rceil$ les nombres de bits nécessaires pour coder respectivement une longueur et une position de répétition (la description de la méthode LZ77 se trouve à la page 54).

La compression d'une séquence S est

$$S' = I\, code(S_{1..i_1}) code(S_{i_1+1..i_2}) \dots code(S_{i_k+1..n})$$

Où :

- I est la séquence de bits contenant le codage de la longueur N de la fenêtre et de la longueur T du tampon.
- $code(S_{i_j+1..i_{j+1}})$ est le codage du triplet (p, l, c) qui identifie le facteur $S_{i_j+1..i_{j+1}}$ de la façon suivante :
  - Le caractère c set S_{i_j+1}
  - Le facteur $S_{i_j+1..i_{j+1}-1}$ est le facteur de longueur l qui est apparu p positions avant la position $i_j + 1$:

$$l = i_{j+1} - i_j - 1 \text{ et } S_{i_j+1..i_j+l} = S_{i_j-p+1..i_j-p+l}$$

  Par convention, si $l = 0$, c'est-à-dire si aucun mot débutant en position $i_j + 1$ n'est un facteur déjà apparu dans les $N - T$ positions précédentes, alors le triplet est $(0, 0, c)$ avec $c = S_{i_j+1}$.

  Le triplet (p, l, c) est codé sur $e_1 + e_2 + k$ bits : e_2 bits pour p, e_1 bits pour l et k bits pour c.

Le schéma de codage de LZ77 est donc modulaire.

Les positions de séparations $i_1, i_2, \dots i_k$ sont donc toutes les positions des caractères c des triplets (p, l, c).

Définissons la courbe de compression $f \in \mathcal{F}^n$ de LZ77 pour la séquence $S \in \mathcal{A}^n$. Elle est définie par :

- $f(0) = -|I|$: le coût du codage des tailles du dictionnaire et du tampon.
- $\forall i_j \in \{i_1, i_2, \dots i_k\} : f(i_j) = k|S_{..i_j}| - |I| - |code(S_{1..i_1}) code(S_{i_1+1..i_2}) \dots code(S_{i_{j-1}+1..i_j})|$ $= k\, i_j - |I| - t_{i_j}(e_1 + e_2 + k)$ où t_{i_j} est le nombre de triplets qui constituent la version comprimée du préfixe $S_{..i_j}$.
- $f(n) = k\, n - |I| - t_n(e_1 + e_2 + k)$ où t_n est le nombre de triplets codés dans la version comprimée S'.

La fonction f n'est pas définie pour les autres positions car le codage d'un triplet ne peut pas être interrompu.

Nous allons maintenant choisir l'annonce de rupture $a_{\mathcal{R}}$.

Il est très important de se rappeler que lorsque le triplet (p, l, c) ne code que pour un symbole de la séquence, c'est-à-dire lorsque $l = 0$, alors par convention, $(0, 0, c)$ a été codé.

Un triplet du type $(1,0,c)$ n'est donc jamais codé. Il peut être utilisé pour l'annonce de rupture. Soit $a_{\mathcal{R}}$ le mot de e_1+e_2+k bits codant pour le triplet $(1,0,a)$ avec $a \in \mathcal{A}$ choisi de manière arbitraire.

La courbe de compression $f \in \mathcal{F}^n$ étant définie, l'annonce de rupture $a_{\mathcal{R}}$ également, il nous reste à déterminer la fonction de rupture ainsi que son retard $r_{\mathcal{R}}$. Choisissons le code *PrefFibo* pour coder la longueur d'une rupture. Il est auto-délimité et ICL.

Dans le pire des cas, un triplet ne code que pour un symbole de la séquence : le triplet est $(0,0,c)$, il est codé sur e_1+e_2+k bits. Puisque $|a_{\mathcal{R}}| = e_1+e_2+k$, il n'est pas possible qu'une rupture sur un seul symbole apporte un gain car $|a_{\mathcal{R}}| + |PrefFibo(1-r_{\mathcal{R}})| > |a_{\mathcal{R}}|$ quel que soit le retard $r_{\mathcal{R}}$. Dès lors, convenons que le retard est de 2 : $r_{\mathcal{R}} = 2$.

La fonction de rupture $\mathcal{R} : \mathbb{N} \to \mathbb{Z} : i \mapsto \mathcal{R}(i)$ est définie par :

$$\begin{cases} \mathcal{R}(0) &= 0 \\ \mathcal{R}(1) &= -|a_{\mathcal{R}}| - |PrefFibo(0)| = -(e_1+e_2+k) - 4 \\ \mathcal{R}(i) &= -(e_1+e_2+k) - |PrefFibo(i-2)| \qquad \forall i \geq 2 \end{cases}$$

L'optimisation de la courbe de compression f à l'aide de TURBOOPTLIFT est réalisée par l'appel $TabLift \leftarrow$ TURBOOPTLIFT$(f, \mathcal{R})$.

Exemple 1.7 *Appliquons la méthode LZ77 sur l'alphabet $\mathcal{N} = \{A, C, G, T\}$ des nucléotides avec la taille de la fenêtre $N = 32$ et la taille du tampon $T = 20$.*

Soit $S \in \mathcal{N}^{107}$ la séquence suivante.

$S =$ *GTTGTTGTTGTTGTTGTTGTACAGGGGACTACGACTACTAGCGGATCGATTTCAGTCTATATATATA TATATATAATACGATATATAATACGTATATAATACGC*

Puisque $\#\mathcal{N} = 4$, chaque symbole peut être codé sur 2 bits : $k = 2$. La séquence S est codée sans aucune compression sur 214 bits. Soit $e_1 = \lceil \log N \rceil = 5$ et $e_2 = \lceil \log N - T \rceil = 4$. Chaque triplet est donc codé sur 11 bits.

Décidons de coder la taille $N - T$ du dictionnaire et la taille T du tampon de manière auto-délimitée en utilisant le codage de Fibo : $I = Fibo(12)Fibo(20)$ et donc $|I| = 15$. La séquence S comprimée est (les triplets ont été écrits tels quels pour plus de clarté) :

$S' =$ $Fibo(12)Fibo(20)(0,0,G)(0,0,T)(1,1,G)(3,19,A)(0,0,C)(2,1,G)(1,3,A)(7,1,T)$
$(3,2,G)(6,5,T)(3,1,G)(10,2,G)(5,1,T)(5,2,A)(4,1,T)(6,2,A)(7,1,T)(4,1,T)$
$(10,2,A)(2,14,A)(3,2,C)(0,0,G)(12,12,T)(11,10C)$

qui comporte $15 + 24 \times 11 = 279$ bits. Il y a donc une perte en compression de 65 bits. La courbe de compression f est représentée à la figure 1.7.

Le retard est $r_{\mathcal{R}} = 2$, l'annonce de rupture est le codage binaire du triplet $(1,0,A)$ (par exemple). On a $|a_{\mathcal{R}}| = 11$. La courbe de rupture $\mathcal{R} : \mathbb{N} \to \mathbb{Z} : i \mapsto \mathcal{R}(i)$ est donc définie par :

$$\begin{cases} \mathcal{R}(0) &= 0 \\ \mathcal{R}(1) &= -11 - |PrefFibo(0)| = -15 \\ \mathcal{R}(i) &= -11 - |PrefFibo(i-2)| \qquad \forall i \geq 2 \end{cases}$$

Elle est représentée à la figure 1.8.

L'appel $TabLift \leftarrow$ TURBOOPTLIFT$(f, \mathcal{R})$ fournit la solution optimale représentée à la figure 1.9. Elle comporte deux zones de ruptures qui correspondent aux deux facteurs soulignés de la séquences S. Le gain de compression après optimisation est de 6 bits, soit 71 bits de plus que sans optimisation. De plus, l'optimisation a clairement mis en évidence les facteurs de la séquence sur lesquels la méthode LZ77 ne fonctionne pas très bien.

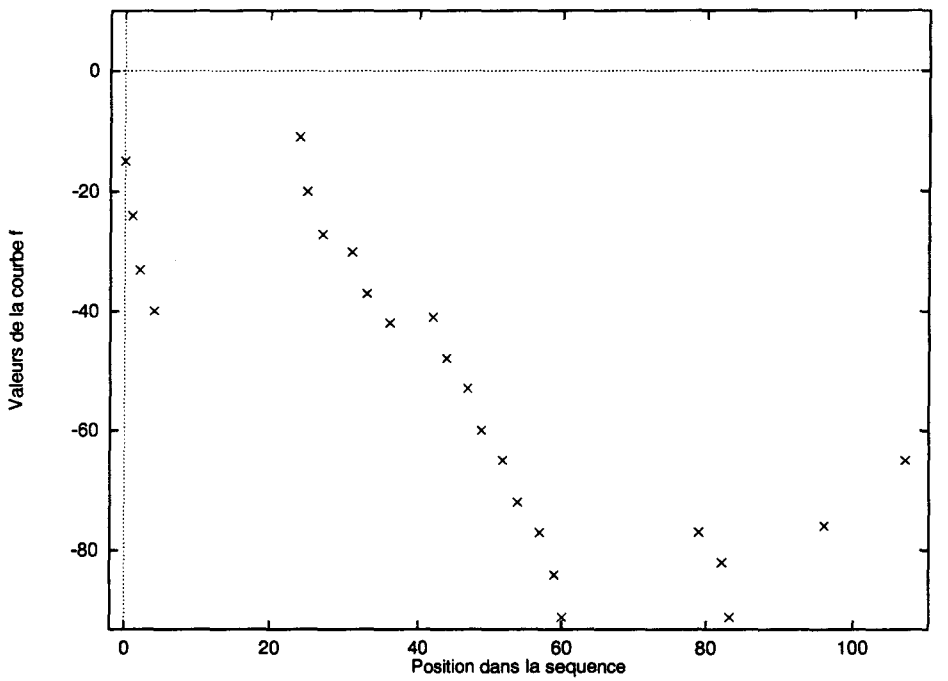


FIG. 1.7 - *Courbe de compression f de LZ77 pour la séquence S*

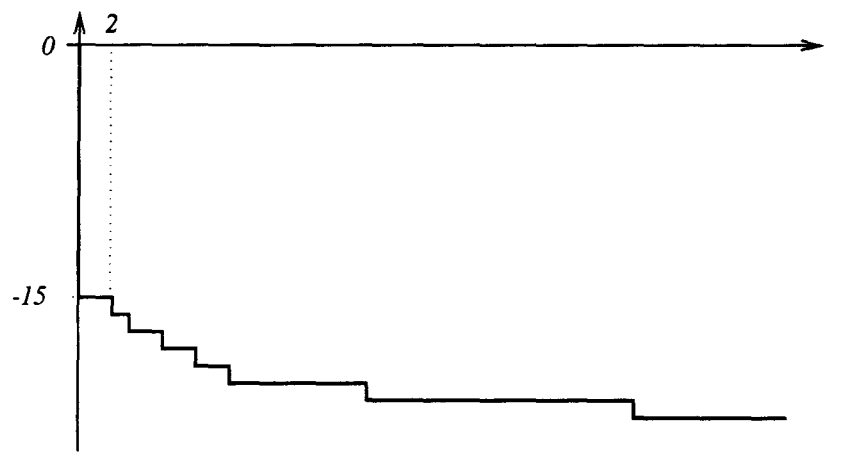


FIG. 1.8 - *Fonction de rupture $\mathcal{R}$ de retard 2*

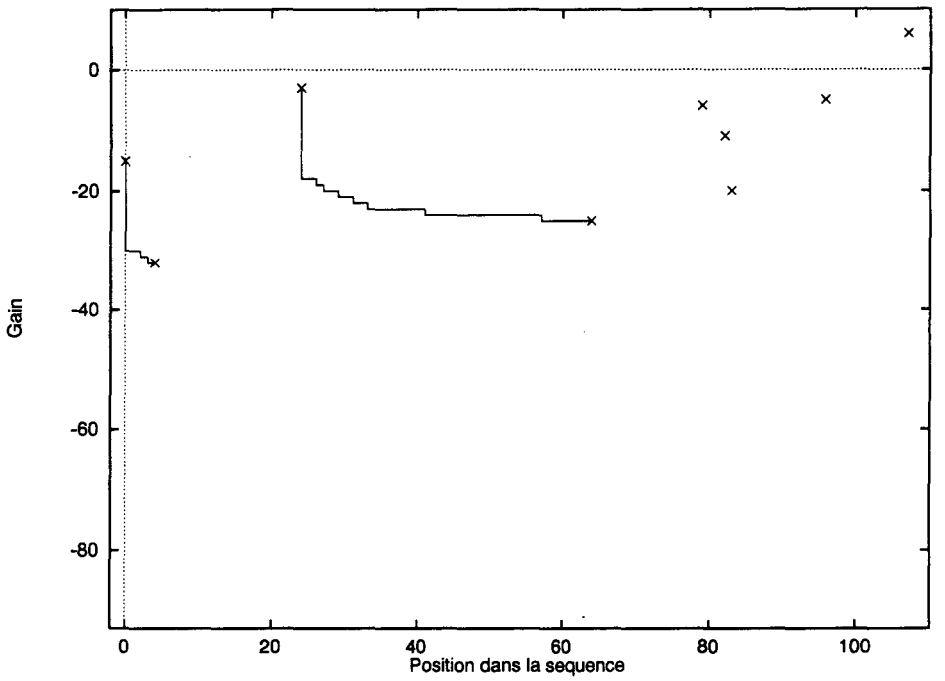


FIG. 1.9 - *Courbe de compression f de LZ77 pour la séquence S, après optimisation*

Lors du décodage, lorsque l'annonce de rupture $a_{\mathcal{R}}$ est détectée, il suffit de décoder la longueur l de la rupture et de lire sans conversion le facteur codé tel quel. Ensuite, avant de reprendre le décodage normal, la fenêtre doit coulisser de l positions vers la droite.

Chapitre 2

Éléments de biologie moléculaire et de bio-informatique

Le but de ce chapitre est d'inculquer quelques notions de biologie moléculaire aux lecteurs non biologistes. Il se contente de faire un vaste survol des quelques concepts et principes de base et n'a certainement pas la prétention d'être un cours de biologie moléculaire. De nombreux ouvrages sont susceptibles d'intéresser le lecteur désireux d'en savoir plus. Parmi ceux-ci, citons arbitrairement [Lew92].

La première section établit les rôles des trois macromolécules chimiques principales des organismes vivants : l'*ADN*, l'*ARN* et les *protéines*. L'ADN est le siège du patrimoine génétique de chaque individu. Certains fragments d'ADN, appelés *gènes* sont utilisés comme matrices pour synthétiser les protéines qui sont les agents actifs de la vie : elles sont les composants de base de toutes les cellules des êtres vivants. L'ARN est une molécule intermédiaire qui joue un rôle fondamental dans l'expression des gènes en protéines. Nous insistons principalement sur les séquences d'ADN qui serviront de matériel expérimental dans la suite du travail.

La seconde section décrit les différents modèles d'évolution de l'ADN au cours du temps. Ces modèles d'évolution constituent la base de beaucoup de recherches en génétique. Ils ont inspiré un certain nombre de modèles et d'algorithmes informatiques d'analyse de séquences d'ADN.

Les deux sections suivantes présentent le contexte du séquençage de fragments d'ADN et l'utilité des méthodes informatiques dans l'analyse de ces fragments.

Un des concepts de base utilisé en analyse et en comparaison de séquences est le concept d'*alignement*. Il fait l'objet de l'avant dernière-section du chapitre. L'alignement permet de comparer plusieurs séquences et d'évaluer leurs ressemblances. C'est un concept central de la bio-informatique, il est utilisé notamment pour conjecturer la fonction d'un gène nouvellement séquencé et pour retracer l'évolution de séquences ou d'espèces vivantes (cette dernière discipline est appelée la *phylogénie*). En ce qui nous concerne, l'alignement sera utilisé dans les chapitres 4 et 5 pour comprimer des séquences génétiques. Nous nous limitons à la méthode classique d'alignement de deux séquences par programmation dynamique.

Le chapitre se termine en établissant quelques notations qui seront utilisées par la suite.

2.1 Les molécules de la vie : ADN, ARN et protéines

De génération en génération, les êtres vivants se transmettent un *patrimoine génétique*. Il est responsable de l'autonomie, la reproduction, le développement et la survie des êtres vivants [Dan93]. Tout ce patrimoine se trouve mémorisé au sein de chaque cellule vivante dans les *molécules d'ADN* (Acide DésoxyriboNucléique). Chez l'homme comme chez la majorité des espèces évoluées, les molécules d'ADN sont confinées à l'intérieur d'un *noyau*. De tels organismes sont appelés des organismes *eucaryotes*. À l'opposé, l'ADN des organismes *procaryotes* (chez qui les cellules ne contiennent pas de noyau) flotte librement dans la cellule. C'est le cas des bactéries telles que *Escherichia coli* ou *Bacillus subtilis* qui sont très étudiées. Les molécules d'ADN d'une cellule constituent ce que l'on appelle les *chromosomes* (ce nom provient de leurs réactions à certains colorants). L'ensemble du patrimoine génétique d'un individu est appelé son *génome*.

L'information véhiculée par l'ADN est très complexe. Ainsi, l'ADN contenu dans une seule cellule est suffisant pour spécifier **totalement** l'individu dont elle est extraite : sa morphologie, ses caractéristiques physiques, les instructions des fonctions vitales telles que la nutrition, la reproduction, la division cellulaire, la régulation de la machinerie génétique,...

Une *chaîne d'ADN* est une longue séquence de *nucléotides* chimiquement liés les uns aux autres. Un nucléotide est l'association d'un sucre (désoxyribose), d'un groupe phosphate et d'une base azotée. La base est le seul composant qui varie d'un nucléotide à l'autre, c'est soit l'*Adénine* (A), la *Cytosine* (C), la *Guanine* (G) ou enfin la *Thymine* (T). Le "squelette" de la chaîne d'ADN est constitué d'une succession de sucre et de phosphate. La base qui complète chaque nucléotide est connectée au sucre. L'extrémité de la molécule qui se termine par le groupe phosphate est appelée *extrémité 5'*, l'autre extrémité est appelée *extrémité 3'* (voir figure 2.1). Une chaîne d'ADN est habituellement orientée de l'extrémité 5' vers l'extrémité 3'. D'un point du contenu en information, une chaîne d'ADN est complètement déterminée par la suite des bases ordonnées de 5' vers 3'.

Définissons dès à présent $\mathcal{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ l'*alphabet des nucléotides*. Une chaîne d'ADN S est un mot sur l'alphabet $\mathcal{N}$: $S \in \mathcal{N}^+$. On parlera désormais de *séquence d'ADN* pour désigner la suite des bases d'un segment orienté d'une chaîne d'ADN. Il s'agit de la suite des bases d'un chromosome entier ou d'un morceau de chromosome. À titre indicatif, le génome humain compte environ 3×10^9 bases réparties sur exactement 46 chromosomes.

En 1953, J.D. Watson et F.H. Crick ont fait la célèbre découverte de la structure de l'ADN [WC53]. Ils ont montré qu'une molécule d'ADN est en fait formée de deux chaînes d'ADN parallèles qui s'enroulent l'une autour de l'autre pour former une double hélice. Chaque base d'une des chaînes d'ADN est *appariée* avec une base de l'autre chaîne par une liaison hydrogène. Pour des raisons stéréochimiques, l'Adénine ne peut s'apparier qu'avec la Thymine et la Cytosine ne peut s'apparier qu'avec la Guanine. Les bases A et T (C et G) sont dites *complémentaires*. La figure 2.2 représente un segment de la structure en double hélice de l'ADN. Les "barreaux horizontaux" de la structure sont les liens "base-base". Les deux brins sont dits complémentaires car la connaissance de l'un est suffisante pour constituer l'autre. La synthèse de nouvelles molécules d'ADN utilise cette propriété. Les deux brins de la molécule sont d'abord séparés, chacun d'eux sert alors de matrice pour reconstruire, nucléotide par nucléotide, un nouveau brin complémentaire (voir figure 2.3). Il y a donc création de deux molécules d'ADN identiques, la molécule initiale n'existe plus. Ce processus est appelé la *réplication* de l'ADN. Il intervient dans la division cellulaire pour dédoubler l'ensemble du matériel génétique de la cellule. Chez les eucaryotes, les deux molécules d'ADN créées sont

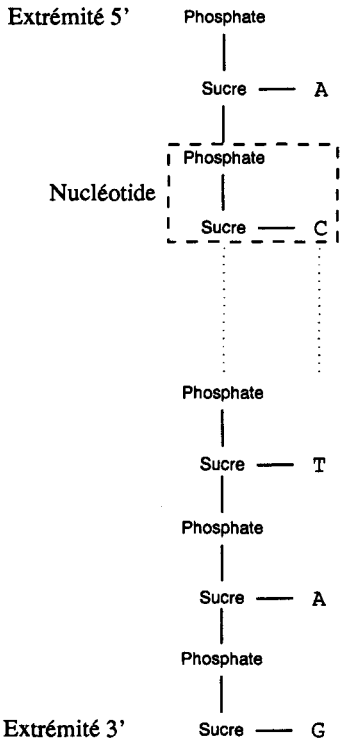


FIG. 2.1 - *Squelette d'une chaîne d'ADN*

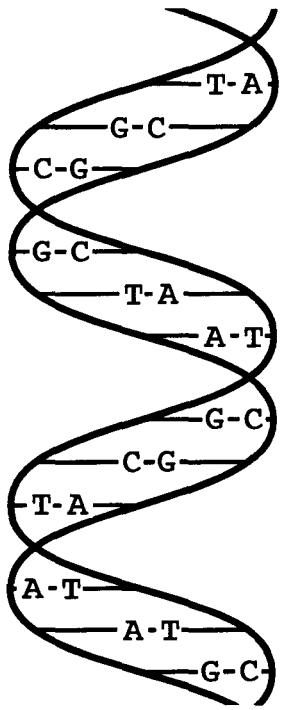


FIG. 2.2 - *Segment d'ADN*

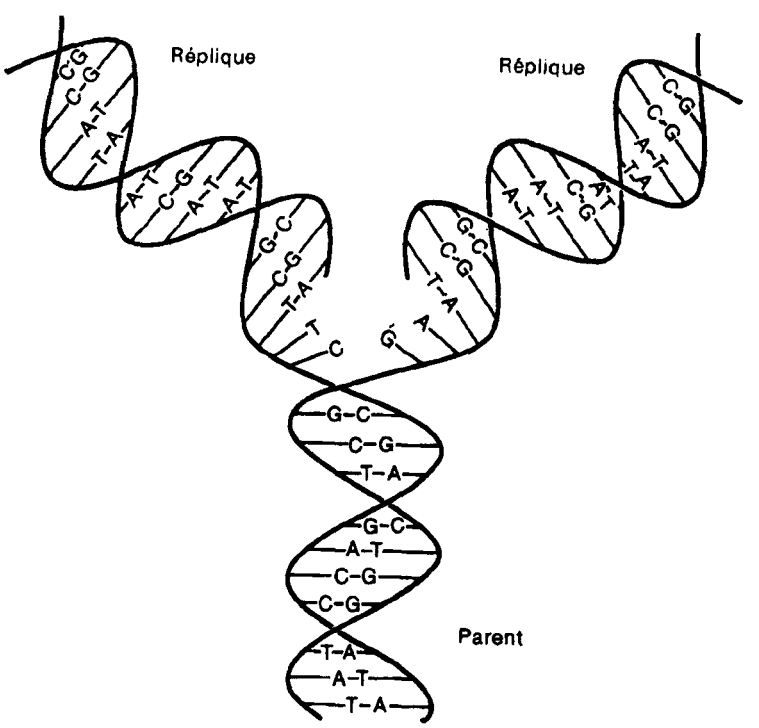


FIG. 2.3 - *Réplication d'ADN*

appelées des *chromatides sœurs*. Elles sont compactées et connectées l'une à l'autre en leur centre pour former la structure en X bien connue du chromosome (voir figure 4.1, page 168) avant la division proprement dite de la cellule.

Puisque dans l'ADN, les bases (ou les nucléotides) vont toujours par paire, l'unité de mesure de la taille d'une molécule d'ADN est la *paire de base* ou *pb*. De la même manière, le "kilo-base", ou *kb*, désigne une succession de 1000 bases ou 1000 pb.

L'ADN est un agent passif de la machinerie de la vie, son rôle se limite à mémoriser l'information génétique. Les agents actifs de la machinerie sont essentiellement les *protéines*. Ces molécules sont responsables de la plupart des fonctions d'une cellule vivante, ce sont elles qui permettent les réactions chimiques qui se déroulent au sein des cellules. Une protéine est une chaîne linéaire d'*acides aminés*. Chaque acide aminé est lié au suivant par une liaison peptide, on dit aussi qu'une protéine est une chaîne polypeptidique. Il existe 20 acides aminés différents, une protéine peut donc être décrite par un mot sur un alphabet à 20 lettres. Cette description est la *structure primaire* de la protéine.

Les protéines sont synthétisées à partir de l'ADN par l'intermédiaire d'une troisième catégorie de molécules: les molécules d'ARN (Acide RiboNucléique). Ce sont des molécules chimiquement très proches de l'ADN. Il s'agit également de chaînes linéaires de nucléotides, les trois différences, d'ordre chimique, entre l'ADN et l'ARN sont:

1. Le sucre utilisé dans chaque nucléotide de l'ARN est un ribose et non pas un désoxyribose.

2. La base azotée Thymine (T) est remplacée par l'*Uracile* (U).

3. Une molécule d'ARN ne contient qu'un seul brin.

Les séquences d'ADN et d'ARN sont toutes deux appelées *séquences nucléiques*. Dans ce travail, nous ne manipulons pas de séquences d'ARN, le terme "séquence nucléique" sera donc toujours utilisé pour désigner une séquence d'ADN.

Un *gène* est une région de l'ADN qui engendre une protéine selon le processus d'*expression génique*. Ce processus est composé des trois étapes suivantes:

1. La *transcription*: la séquence du gène est transcrite en un brin d'ARN. Pour cela, les deux brins d'ADN sont d'abord séparés. L'un des deux brins sert alors de matrice pour la construction, nucléotide par nucléotide, d'un brin d'ARN complémentaire. Le processus est identique à celui de la réplication de l'ADN si ce n'est que la base complémentaire de l'Adénine est l'Uracile. La molécule d'ADN n'est pas modifiée par ce processus, le brin utilisé comme matrice vient se replacer contre le brin complémentaire. La molécule d'ARN ainsi obtenue est appelée le *transcrit primaire*.

2. La *maturation*: certains segments, les *introns* sont supprimés de la molécule d'ARN et les portions restantes, les *exons* sont concaténées pour produire le *transcrit mature* (ou ARN messager, ou encore ARNm). La figure 2.4 décrit le processus de maturation. Ce processus est également appelé l'*épissage* de l'ARN. La molécule d'ARNm traverse alors la membrane du noyau de la cellule pour être *traduite* dans le cytoplasme.

   Ce type de maturation n'existe que chez les eucaryotes car les procaryotes ne possèdent pas d'introns. Une maturation d'ARN chez les procaryotes n'est nécessaire à la traduction que dans des cas exceptionnels.

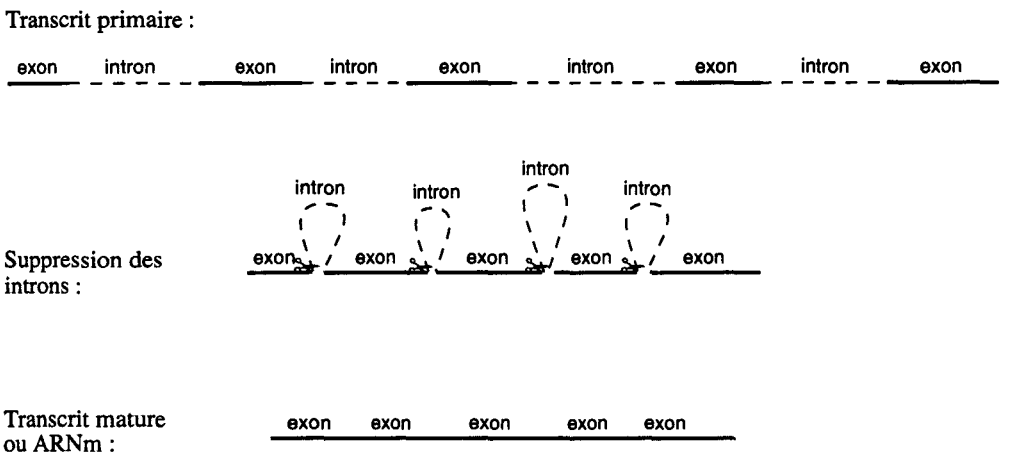


FIG. 2.4 - *Maturation du transcrit primaire*

3. La *traduction* : la séquence d'ARNm est parcourue séquentiellement par groupes de trois nucléotides (les *codons*) à l'aide d'un ribosome. Il construit la protéine en générant un acide aminé pour chaque codon lu, selon le *code génétique* décrit au tableau 2.1 (le code `stop` identifie un "codon stop" auquel n'est associé aucun acide aminé, c'est simplement un signal qui indique la terminaison de la traduction). Prenons par exemple le codon `CGA`. Il est traduit en l'acide aminé Arg (*Arginine* : en gras dans le tableau).

base 1 \base 2	U	C	A	G	base 3
U	Phe	Ser	Tyr	Cys	U
	Phe	Ser	Tyr	Cys	C
	Leu	Ser	*stop*	*stop*	A
	Leu	Ser	*stop*	Trp	G
C	Leu	Pro	His	Arg	U
	Leu	Pro	His	Arg	C
	Leu	Pro	Gln	**Arg**	A
	Leu	Pro	Gln	Arg	G
A	Ile	Thr	Asn	Ser	U
	Ile	Thr	Asn	Ser	C
	Ile	Thr	Lys	Arg	A
	Met	Thr	Lys	Arg	G
G	Val	Ala	Asp	Gly	U
	Val	Ala	Asp	Gly	C
	Val	Ala	Glu	Gly	A
	Val	Ala	Glu	Gly	G

TAB. 2.1 - *Code génétique*

Puisqu'il n'existe que 20 acides aminés et que le nombre de codons possibles est de 61 ($4^3 = 64$ sans compter les trois codons stop), le code est redondant : plusieurs codons correspondent au même acide aminé.

La proportion de gènes dans l'ADN dépend fortement des organismes. Chez les procaryotes, les gènes sont beaucoup plus présents que chez les eucaryotes. Chez l'homme, seulement 3% à 5% de l'ADN constitue des gènes (de 100000 à 200000 gènes).

La science actuelle possède peu d'informations concernant les autres portions d'ADN. On sait qu'elles contiennent certains signaux destinés à guider la machine génétique. Le reste constitue probablement une mémoire de l'évolution de l'espèce.

Remarquons enfin que pour certains gènes, le processus d'expression génique s'arrête après la transcription en ARN. Il s'agit des gènes des ARN *de transfert* (ARNt) et des ARN *ribosomaux* (ARNr) qui ont un rôle bien précis à jouer dans la machinerie génétique elle-même.

2.2 Évolution de l'ADN au cours du temps

Le génome d'une espèce est dynamique, il est en perpétuelle évolution. Il subit principalement trois types de modifications [Riv96] :

1. Lors de la *Reproduction* : le génome de certains organismes est *diploïde* : l'œuf fécondé reçoit le stock de chromosomes de chaque parent. Il s'ensuit un "mélange" d'ADN pour les générations à venir.

2. *Mutations ponctuelles* : l'ADN subit des insertions d'une base, délétion d'une base et substitutions d'une base par une autre.

   L'idée d'*alignement* par programmation dynamique (voir section 2.5) provient de ce modèle d'évolution de l'ADN.

3. Échanges de longs fragments de chromosomes par *recombinaisons homologues*, aussi appelé *crossing-over* (échange de matériel génétique entre chromosomes pendant la phase de division cellulaire qui prélude à la formation des cellules germinales (la *méiose*)).

Mentionnons également certains mécanismes qui provoquent des cassures de longues portions d'ADN : exposition des cellules aux rayons X, rayons gamma,... Ces cassures sont exceptionnelles et sont parfois réparées automatiquement par la machinerie génétique.

L'évolution de l'ADN implique un *polymorphisme*, d'une part entre les individus d'une même espèce (ils n'ont pas tous le même ADN), d'autre part entre les cellules d'un même individu.

À l'intérieur des gènes, les mutations sont très contraintes dans le sens où les protéines pour lesquelles ils codent doivent rester fonctionnelles ou proposer une nouvelle fonction intéressante pour l'organisme. Cette considération rejoint la théorie de l'évolution des espèces introduite par Charles Darwin dans son ouvrage "De l'origine des espèces" (1859). Les modifications que subit l'ADN possèdent une grande part d'aléatoire. Les mutations qui modifient les gènes et rendent les protéines déficientes ne sont pas conservées. Certaines sont même réparées par la machinerie génétique. Ne restent donc que les mutations qui sont "bénéfiques" pour l'organisme. Bien entendu, l'espèce n'évolue que si les modifications d'ADN sont répercutées de génération en génération, c'est-à-dire si elles concernent également les cellules germinales des individus.

Par contre, les mutations "hors gènes" sont peu contraintes, les régions inter-géniques se modifient donc beaucoup plus rapidement que les gènes. Il s'ensuit un polymorphisme inter-génique substantiel.

2.3 Séquençage d'ADN

Les principaux chiffres et détails techniques de cette section sont issus de l'article [Dan93].

Le séquençage d'ADN est la détermination, base par base, de la composition d'un génome ou du moins d'un fragment de chromosome. Bien entendu, la connaissance de tout le génome d'un organisme n'est pas suffisante pour déterminer l'ensemble des fonctions indispensables à la vie de l'organisme. Il est nécessaire d'analyser les séquences d'ADN du génome pour déterminer quels sont les gènes, que sont les protéines pour lesquelles ils codent, et pour comprendre un peu mieux le fonctionnement de la machinerie génétique.

Le séquençage et l'analyse du génome humain sont deux étapes très importantes dans la recherche biologique mais aussi médicale. Le génome humain est énorme, il contient un peu plus de trois milliards de paires de bases. Son séquençage est un travail très ambitieux.

Deux grandes écoles existent concernant le séquençage du génome humain. Tout d'abord l'école "américaine" qui juge que le génome humain est le plus important et que tous les efforts doivent être faits pour le séquencer. Sous l'impulsion de J.D. Watson, un projet de séquençage du génome humain a été mis sur pied. C'est un projet qui est loin d'être terminé.

L'autre école est celle qui juge que les connaissances en génétique ne sont pas encore assez poussées pour s'attaquer à un projet aussi gros que celui du génome humain. Le séquençage de génomes d'organismes modèles est un préalable indispensable au séquençage du génome humain. Le séquençage de tels génomes permettra de développer des techniques de séquençages rapides et aussi d'apprendre à reconnaître ce qui est pertinent dans une séquence d'ADN.

Les plus importants organismes modèles auxquels les biologistes s'intéressent sont :

- *Mus laboratorius* (souris de laboratoire) : Eucaryote. Son génome contient 3000000 kb (contre 3300000 kb pour l'homme). Bien que son génome soit très grand, il constitue un organisme intéressant à étudier car ses gènes sont proches de ceux de l'homme.

- *Drosophila melanogaster* (mouche du vinaigre) : Eucaryote dont la génétique est bien connue depuis les travaux de T.H. Morgan au début du siècle. Son génome comporte 170000 kb.

- *Arabidopsis thaliana* (plante crucifère) : Eucaryote. Son génome comporte 100000 kb. Cet organisme est intéressant pour étudier tout ce qui relève de la photosynthèse.

- *Cœnorhabditis elegans* (vers minuscule des sols) : Eucaryote. Possède peu de cellules (959 cellules pour un adulte mâle). Génome total de 110000 kb environ.

- *Saccharomyces cerevisiae* (levure de boulanger) : Eucaryote. Son génome contient 15000 kb. **C'est le premier organisme eucaryote dont la séquence complète du génome est connue.** Sous l'impulsion d'André Goffeau, professeur à l'université catholique de Louvain, les 16 chromosomes ont été séquencés par l'association de quelques dizaines de laboratoires internationaux. Le séquençage s'est terminé en 1996. La levure est étudiée depuis plus de cinquante ans par des centaines de laboratoires de génétique, c'était notamment un des fleurons de travail de Pasteur.

  Dans le chapitre 4, nous utilisons les chromosomes de la levure comme matériel d'expérimentation pour notre programme de détection de répétitions en tandem approximatives.

- *Escherichia coli* (bactérie qui vit dans l'intestin de l'homme) : Procaryote. Son génome est fait d'un seul chromosome qui contient 4720 kb.

- *Bacillus subtilis* (bactérie qui forme des spores) : Procaryote. Son génome est fait d'un seul chromosome qui contient 4200 kb.

Les bactéries ont un génome extrêmement compact, tout semble y être signifiant. Elles sont étudiées depuis très longtemps, on dispose d'informations génétiques nombreuses à leur sujet.

2.4 Utilité de l'informatique dans le séquençage

Une des conséquence du séquençage massif d'ADN est l'énorme quantité de données à stocker, à analyser et à rendre disponible aux autres chercheurs. L'informatique aide les généticiens dans ces tâches. Cette section est inspirée de l'article [HD94] qui, bien qu'étant complètement dédié à Escherichia coli, effectue un vaste parcours des méthodes informatiques appliquées à la génétique moléculaire.

Beaucoup de domaines de l'informatique sont concernés : les bases de données, l'algorithmique, la combinatoire, le traitement du signal, l'intelligence artificielle, les réseaux,... Autant de domaines qui rendent les projets de séquençage interdisciplinaires [Dan93]. De plus en plus, les génomes sont étudiés expérimentalement, en utilisant les ordinateurs comme dispositif expérimental. Aux approches habituelles de la biologie "in vivo" et "in vitro" vient donc maintenant s'ajouter l'approche "in silico" [HD94].

L'informatique intervient à plusieurs niveaux du séquençage, à chaque niveau correspondent des questions spécifiques à la fois pour les biologistes et pour les informaticiens :

- Lors de l'*acquisition* des séquences : les fragments séquencés sont très petits (quelques centaines de bases) et il convient, grâce à l'informatique, de construire une seule séquence finale par recouvrement continu des morceaux séquencés. Cette séquence finale est appelée un *contig*.

  Il est également important de pouvoir localiser la séquence obtenue dans le génome de l'organisme.

- Lors de l'*exploitation* des séquences : lorsqu'un fragment d'ADN a été séquencé, il convient de localiser les gènes éventuels qui codent pour des protéines, des ARNt ou des ARNr. On parle de *prédiction de gènes*. Beaucoup de méthodes existent mais le problème n'est pas complètement résolu.

  De nombreux signaux peuvent être recherchés dans le fragment, parmi ceux-ci :

  - Les *promoteurs de transcription* : signaux de démarrage de la transcription au début d'un gène ou d'un groupe de gènes.
  - Les *terminateurs de transcription* : signaux de fin de transcription.
  - Les introns dans les gènes.
  - Les régions d'*initiation de traduction*.
  - Les motifs statistiquement significatifs.
  - Les motifs répétés.
  - Les zones régulières (au sens de la complexité de Kolmogorov).
  - Les motifs répartis tout au long de la séquence.
  - etc.

On cherche souvent à déterminer les ressemblances entre deux ou plus de deux séquences (voir la notion d'alignement, section 2.5). De cette façon, la *phylogénie* est la branche de la génétique qui tente de déterminer l'arbre d'évolution des séquences et des espèces en déterminant les ancêtres communs des séquences par alignement.

- Lors de la *gestion* des séquences : les séquences doivent être classées correctement dans des banques de données. Les deux grandes banques de séquences nucléiques utilisées sont les banques de l'*EMBL* (*European Molecular Biology Laboratory*) et *Genbank* (*Genetic sequences data Bank*). Elles contiennent tous les fragments d'ADN publiés. Le 15 mars 1997, l'EMBL contenait près de 200000000 pb et Genbank 248500000 pb. Ces chiffres changent fréquemment, la quantité de paires de bases contenues dans les banques double environ tous les 18 mois.

  Les banques sont distribuées sur beaucoup de serveurs du réseau internet pour être interrogeables par toute la communauté scientifique.

  Beaucoup de programmes existent pour extraire les informations contenues dans les banques de séquences (ACNUC, ATLAS, SRS et Entrez sont les plus importants). Ils diffèrent quant aux types de questions que l'on peut leur poser. Souvent, on désire extraire des séquences d'ADN (ou des séquences protéiques, des banques de séquences protéiques existent également) qui vérifient certains critères tels que : appartenance à un organisme donné, ressemblance (voir la notion d'alignement, section 2.5) à une séquence donnée, possession une occurrence d'un motif donné,...

  Il existe même des banques spécialisées dédiées à un organisme spécifique.

2.5 Alignements de séquences

Dans cette section, nous présentons la notion d'*alignement* qui permet d'évaluer la ressemblance entre deux séquences.

Lorsqu'un nouveau fragment F d'ADN a été séquencé, il est important de voir s'il ressemble à une séquence déjà séquencée. S'il ressemble à une séquence S déjà répertoriée, alors S et F sont deux séquences qui ont probablement évolué indépendamment à partir d'une même séquence ancestrale (pour déduire cela, les longueurs des deux séquences S et F doivent bien entendu être suffisantes et la ressemblance doit être forte).

Si S est un gène, alors il y a de grandes chances pour que F soit également un gène et que la fonction de la protéine codée par F soit proche de la fonction de la protéine codée par S. Cette constatation n'est pas vraie dans tous les cas, il faut se prémunir contre de mauvaises interprétations : les mutations indépendantes des deux gènes peut les avoir fait changer totalement de fonction, c'est un des rôles de l'évolution.

Nous nous restreignons à la méthode classique de recherche de l'alignement optimal par programmation dynamique. Pour en savoir plus sur l'alignement de séquences, se référer aux ouvrages [SK83, Wat89, Mye91, CR94, Ste94, Sag96]. La méthode d'alignement par programmation dynamique est une méthode **exacte** dont la complexité est quadratique. Il existe un certain nombre de méthodes **approchées** qui ne garantissent pas de trouver le meilleur alignement mais qui fournissent le résultat plus rapidement. Elles sont utilisées pour effectuer du *criblage de banque*, c'est-à-dire la comparaison systématique d'une séquence donnée avec toutes les séquences d'une banque de séquences. Les plus connues de ces méthodes sont d'une

part FASTN, FASTP, FASTA [LP85, PL88, Pea90] et, d'autre part, BLAST [AGM+90]. Nous n'en parlerons pas ici.

Nous nous restreignons à l'alphabet $\mathcal{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ des nucléotides mais tout peut être facilement généralisé à d'autres alphabets.

2.5.1 Notion d'alignement

Une image intéressante pour présenter la comparaison de séquences nucléiques par alignement est l'image du *collier de perles* [Sag96]. Supposons qu'une séquence soit un collier de perles de couleurs différentes, les perles étant les nucléotides. Chaque perle peut donc être d'une des quatre "couleurs" : A, C, G ou T. Les perles du collier peuvent glisser le long du fil mais l'ordre des perles est toujours respecté. Une façon de comparer deux séquences est de les allonger l'une en-dessous de l'autre et d'examiner chaque paire de perles mises ainsi face à face pour voir lesquelles sont identiques et lesquelles diffèrent. Si l'on autorise toutes les configurations possibles, il se peut que des perles d'un des colliers ne soient pas face à des perles de l'autre collier.

Exemple 2.1 *Soit* $s =$ ACGGATCTAAGC *et* $t =$ ATGGACTGAAAGT *deux séquences sur* $\mathcal{N}$ *avec* $|s| = 12$ *et* $|t| = 13$. *La figure 2.5 représente un alignement des deux colliers de perles correspondant aux séquences* s *et* t.

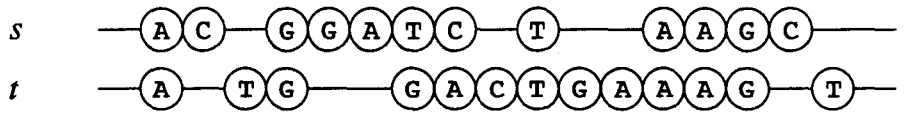


FIG. 2.5 - *Alignement des deux colliers de perles* s *et* t

Le processus qui consiste à placer les deux séquences l'une au-dessous de l'autre et à mettre certaines bases en face d'autres est ce que l'on appelle réaliser un *alignement*. La ressemblance entre les deux séquences peut être mesurée simplement en comptant le nombre de bases mises face à face qui sont identiques. Inversément, on peut mesurer la dissimilarité entre les deux séquences en mesurant le nombre bases mises face à face qui sont différentes et le nombre de bases mises en face de rien du tout.

Définissons de manière plus formelle l'alignement entre deux séquences s et t.

Définition 2.1 *Soient* $s = s_1 s_2 \dots s_n \in \mathcal{N}^+$ *et* $t = t_1 t_2 \dots t_m \in \mathcal{N}^+$ *deux séquences. Soit* $\overline{\mathcal{N}} = \mathcal{N} \cup \{\epsilon\}$. *Un alignement de* s *et* t *est une suite* $(\bar{s}_1, \bar{t}_1)(\bar{s}_2, \bar{t}_2) \dots (\bar{s}_p, \bar{t}_p)$ *définie sur* $\overline{\mathcal{N}} \times \overline{\mathcal{N}}$, *avec* $\max(n, m) \leq p \leq n + m$ *et telle que :*

- $\bar{s}_i = s_j$ *ou* $\bar{s}_i = \epsilon$ *pour* $i, j : 1 \leq i \leq p$ *et* $1 \leq j \leq n$.
- $\bar{t}_i = t_j$ *ou* $\bar{t}_i = \epsilon$ *pour* $i, j : 1 \leq i \leq p$ *et* $1 \leq j \leq m$.
- $\bar{s}_1 \bar{s}_2 \dots \bar{s}_p = s$ *: l'ordre des bases est respecté et toutes les bases de* s *sont présentes dans l'alignement. Seuls des* ϵ *ont été intercalés entre deux bases de* s.
- $\bar{t}_1 \bar{t}_2 \dots \bar{t}_p = t$ *: l'ordre des bases est respecté et toutes les bases de* t *sont présentes dans l'alignement. Seuls des* ϵ *ont été intercalés entre deux bases de* t.

- *Pour tout* i : $1 \leq i \leq p$, *on n'a jamais* $\bar{s}_i = \bar{t}_i = \epsilon$: *à chaque position alignée* i *se trouve une base dans au moins une des séquences.*

Chaque couple $(\bar{s}_i, \bar{t}_i) \in \overline{\mathcal{N}} \times \overline{\mathcal{N}}$ *est appelé une* paire alignée. *Pour mettre en évidence les paires alignées, on note l'alignement* $(\bar{s}_1, \bar{t}_1)(\bar{s}_2, \bar{t}_2) \ldots (\bar{s}_p, \bar{t}_p)$ *par :*

$$\binom{\bar{s}_1}{\bar{t}_1}\binom{\bar{s}_2}{\bar{t}_2}\cdots\binom{\bar{s}_p}{\bar{t}_p}$$

Une paire du type $\binom{s_j}{\epsilon}$ représente une perle, sur le collier du haut, en face de laquelle il n'y a pas de perle sur le collier du bas. De la même manière, une paire du type $\binom{\epsilon}{t_j}$ représente une perle, sur le collier du bas, en face de laquelle il n'y a pas de perle sur le collier du haut.

Exemple 2.2 *Reprenons les séquences* s *et* t *de l'exemple 2.1. L'alignement représenté par la figure 2.5 est :*

$$\binom{A}{A}\binom{C}{\epsilon}\binom{\epsilon}{T}\binom{G}{G}\binom{G}{\epsilon}\binom{A}{\epsilon}\binom{T}{G}\binom{C}{A}\binom{\epsilon}{C}\binom{T}{T}\binom{\epsilon}{G}\binom{\epsilon}{A}\binom{A}{A}\binom{A}{A}\binom{G}{G}\binom{C}{\epsilon}\binom{\epsilon}{T}$$

Il contient 17 *paires alignées.*

D'une façon générale, un alignement entre deux séquences s et t représente une suite de *transformations élémentaires* qui permettent de passer de s à t :

- Une paire alignée $\binom{s_i}{t_j}$, avec $s_i, t_j \in \mathcal{N}$ et $s_i \neq t_j$ représente la *substitution* de s_i par t_j.
- Une paire alignée $\binom{s_i}{\epsilon}$, avec $s_i \in \mathcal{N}$ représente la *délétion* de s_i.
- Une paire alignée $\binom{\epsilon}{t_j}$, avec $t_i \in \mathcal{N}$ représente l'*insertion* de t_j.
- Une paire alignée $\binom{s_i}{t_j}$, avec $s_i, t_j \in \mathcal{N}$ et $s_i = t_j$ représente la *coïncidence* entre s_i et t_j.

Exemple 2.3 *Reprenons le même alignement que celui de l'exemple 2.2. Il exprime que la transformation de la séquence* s *en* t *consiste en :*

- $\binom{A}{A}$: *on laisse telle quelle la première base* A.
- $\binom{C}{\epsilon}$: *on supprime* C
- $\binom{\epsilon}{T}$: *on insère* T

- $\begin{pmatrix} G \\ G \end{pmatrix}$: *on laisse G*
- $\begin{pmatrix} G \\ \epsilon \end{pmatrix}$: *on supprime G*
- $\begin{pmatrix} A \\ \epsilon \end{pmatrix}$: *on supprime A*
- $\begin{pmatrix} T \\ G \end{pmatrix}$: *on substitue le T par un G*
- *etc.*

À chaque paire potentielle $\begin{pmatrix} a \\ b \end{pmatrix}$, avec $a, b \in \overline{\mathcal{N}}$ telle que l'on n'ait pas $a = b = \epsilon$, on attribue un *coût* que l'on note $\omega\begin{pmatrix} a \\ b \end{pmatrix}$.

On peut voir ces coûts comme faisant partie d'une *matrice de coûts* :

$$\Omega = \begin{pmatrix} \lambda & \omega\begin{pmatrix} \mathtt{A} \\ \epsilon \end{pmatrix} & \omega\begin{pmatrix} \mathtt{C} \\ \epsilon \end{pmatrix} & \omega\begin{pmatrix} \mathtt{G} \\ \epsilon \end{pmatrix} & \omega\begin{pmatrix} \mathtt{T} \\ \epsilon \end{pmatrix} \\ \omega\begin{pmatrix} \epsilon \\ \mathtt{A} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{A} \\ \mathtt{A} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{C} \\ \mathtt{A} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{G} \\ \mathtt{A} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{T} \\ \mathtt{A} \end{pmatrix} \\ \omega\begin{pmatrix} \epsilon \\ \mathtt{C} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{A} \\ \mathtt{C} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{C} \\ \mathtt{C} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{G} \\ \mathtt{C} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{T} \\ \mathtt{C} \end{pmatrix} \\ \omega\begin{pmatrix} \epsilon \\ \mathtt{G} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{A} \\ \mathtt{G} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{C} \\ \mathtt{G} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{G} \\ \mathtt{G} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{T} \\ \mathtt{G} \end{pmatrix} \\ \omega\begin{pmatrix} \epsilon \\ \mathtt{T} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{A} \\ \mathtt{T} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{C} \\ \mathtt{T} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{G} \\ \mathtt{T} \end{pmatrix} & \omega\begin{pmatrix} \mathtt{T} \\ \mathtt{T} \end{pmatrix} \end{pmatrix}$$

La valeur λ est sans importance puisque la paire $\begin{pmatrix} \epsilon \\ \epsilon \end{pmatrix}$ n'apparaît jamais.

On définit alors le *coût d'un alignement.*

Définition 2.2 *Soient* $s, t \in \mathcal{N}^+$ *et* $A = \begin{pmatrix} \bar{s}_1 \\ \bar{t}_1 \end{pmatrix}\begin{pmatrix} \bar{s}_2 \\ \bar{t}_2 \end{pmatrix}\ldots\begin{pmatrix} \bar{s}_p \\ \bar{t}_p \end{pmatrix}$ *un alignement de s et t. Le* coût de l'alignement *A est la somme des coûts de ses paires alignées :*

$$\omega(A) = \omega\begin{pmatrix} \bar{s}_1 \\ \bar{t}_1 \end{pmatrix} + \omega\begin{pmatrix} \bar{s}_2 \\ \bar{t}_2 \end{pmatrix} + \ldots + \omega\begin{pmatrix} \bar{s}_p \\ \bar{t}_p \end{pmatrix}$$

Exemple 2.4 *Supposons que la matrice de coûts soit (les coïncidences ont un coût nul, les délétions et les substitutions un coût de* 1*, les insertions un coût de* 2*) :*

$$\Omega = \begin{pmatrix} \lambda & 1 & 1 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Le coût de l'alignement de l'exemple 2.2 est:

$$\omega\begin{pmatrix} A \\ A \end{pmatrix} + \omega\begin{pmatrix} C \\ \epsilon \end{pmatrix} + \omega\begin{pmatrix} \epsilon \\ T \end{pmatrix} + \omega\begin{pmatrix} G \\ G \end{pmatrix} \ldots = 0 + 1 + 2 + 0 + \ldots = 16$$

Remarque 2.1 *Souvent, en biologie moléculaire, on considère que les délétions (resp. insertions) de plusieurs bases consécutives ne sont pas indépendantes, elles proviennent d'un même événement mutationnel. Dès lors, le coût de délétion (resp. d'insertion) de plusieurs bases adjacentes n'est pas égal à la somme des coûts individuels de délétions (resp. d'insertions) successives des bases une à une. Dans ce cas, on considère que (pour la délétion multiple):*

$$\omega\left(\begin{pmatrix} s_i \\ \epsilon \end{pmatrix}\begin{pmatrix} s_{i+1} \\ \epsilon \end{pmatrix}\ldots\begin{pmatrix} s_k \\ \epsilon \end{pmatrix}\right) \leq \omega\begin{pmatrix} s_i \\ \epsilon \end{pmatrix} + \omega\begin{pmatrix} s_{i+1} \\ \epsilon \end{pmatrix} + \ldots + \omega\begin{pmatrix} s_k \\ \epsilon \end{pmatrix}$$

Nous ne nous intéressons pas à ce cas ici, nous considérons uniquement le modèle d'évolution dans lequel les délétions (resp. insertions) sont ponctuelles.

Le coût d'une délétion est toujours égal au coût d'une insertion; il n'est pas possible d'établir dans quel sens s'est produite l'évolution. On parle plus généralement de indel *ou de* gap *pour désigner l'insertion ou la délétion. En pratique, la matrice Ω est donc symétrique.*

2.5.2 Ressemblance par alignement

Maintenant que l'on a pu associer un coût à un alignement, on peut définir une *mesure de ressemblance* entre deux séquences. C'est le coût de l'alignement optimal.

Définition 2.3 *Soient $s, t \in \mathcal{N}^+$ et la fonction de coût ω. Soit R_ω la* mesure de ressemblance, *relative à ω, entre s et t. Elle se calcule par:*

$$R_\omega(s,t) = \min\{\omega(A) \mid A \text{ est un alignement de } s \text{ et } t\}$$

Plus petite est la valeur de $R_\omega(s,t)$, plus s "ressemble" à t.

Exemple 2.5 *Reprenons les séquences s et t de l'exemple 2.1 ainsi que la fonction de coût ω déterminée par la matrice de coût Ω de l'exemple 2.4. La mesure de la ressemblance entre s et t est: $R_\omega(s,t) = 6$. Un des alignements optimaux est:*

$$\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} C \\ T \end{pmatrix}\begin{pmatrix} G \\ G \end{pmatrix}\begin{pmatrix} G \\ G \end{pmatrix}\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} \epsilon \\ C \end{pmatrix}\begin{pmatrix} T \\ T \end{pmatrix}\begin{pmatrix} C \\ G \end{pmatrix}\begin{pmatrix} T \\ A \end{pmatrix}\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} G \\ G \end{pmatrix}\begin{pmatrix} C \\ T \end{pmatrix}$$

Remarque 2.2 *Lorsque la fonction de coût est: $\omega\begin{pmatrix} a \\ a \end{pmatrix} = 0$, $\omega\begin{pmatrix} a \\ \epsilon \end{pmatrix} = \omega\begin{pmatrix} \epsilon \\ a \end{pmatrix} = \omega\begin{pmatrix} a \\ b \end{pmatrix} = 1$, pour tous les $a, b \in \mathcal{N}$ et $a \neq b$, alors $R_\omega(s,t)$ est appelée la* distance d'édition *entre s et t ou la* distance de Levenshtein *[Lev66]. Dans l'exemple 2.1, la distance de Levenshtein entre s et t est* 5.

2.5.3 Alignement optimal par programmation dynamique

Nous nous intéressons maintenant au problème du calcul de la ressemblance entre deux séquences et de la recherche d'un alignement optimal.

L'algorithme de programmation dynamique que nous présentons a été découvert et publié de multiples fois de manière indépendante. Le problème d'alignement ne concerne pas que la biologie moléculaire, il concerne également le domaine du traitement de la parole, de la traduction automatique et d'une façon générale, la recherche de ressemblance entre mots ou textes. Les versions les plus connues de l'algorithme de programmation dynamique sont celles de Needleman et Wunsch (en biologie moléculaire) [NW70], Wagner et Fisher (en informatique: édition de mots) [WF75] et Sellers (en informatique: édition de mots) [Sel74].

Le principe de la programmation dynamique est d'évaluer la ressemblance entre des préfixes de plus en plus longs des deux séquences jusqu'à ce que la ressemblance des deux chaînes complètes ait été trouvée.

On désire trouver un alignement optimal $\bar{A}$ entre $s = s_1 s_2 \dots s_n$ et $t = t_1 t_2 \dots t_m$ pour la fonction de coût ω, c'est-à-dire trouver $\bar{A}$ tel que $\omega(\bar{A}) = \min\{\omega(A) \mid A \text{ est un alignement de } s \text{ et } t\}$.

L'idée de l'algorithme est la suivante.

Puisqu'une paire alignée $\begin{pmatrix} \epsilon \\ \epsilon \end{pmatrix}$ n'est pas autorisée, l'alignement $\bar{A}$ est d'une des trois formes :

- $\bar{A} = A_{n,m-1}\begin{pmatrix} \epsilon \\ t_m \end{pmatrix}$ avec $A_{n,m-1}$ un alignement entre s et $t_{..m-1}$.
- $\bar{A} = A_{n-1,m}\begin{pmatrix} s_n \\ \epsilon \end{pmatrix}$ avec $A_{n-1,m}$ un alignement entre $s_{..n-1}$ et t.
- $\bar{A} = A_{n-1,m-1}\begin{pmatrix} s_n \\ t_m \end{pmatrix}$ avec $A_{n-1,m-1}$ un alignement entre $s_{..n-1}$ et $t_{..m-1}$.

Grâce aux trois propositions suivantes, nous allons pouvoir définir une récurrence qui nous permettra de rechercher l'alignement optimal $\bar{A}$.

Proposition 2.1 *Si* $\bar{A} = A_{n,m-1}\begin{pmatrix} \epsilon \\ t_m \end{pmatrix}$, *alors l'alignement* $A_{n,m-1}$ *est un alignement optimal entre* s *et* $t_{..m-1}$.

Preuve. Supposons que l'alignement $A_{n,m-1}$ ne soit pas optimal, c'est-à-dire qu'il existe un alignement $A'_{n,m-1}$ entre s et $t_{..m-1}$ tel que $\omega(A'_{n,m-1}) < \omega(A_{n,m-1})$. Alors l'alignement $\bar{A}' = A'_{n,m-1}\begin{pmatrix} \epsilon \\ t_m \end{pmatrix}$ entre s et t est tel que $\omega(\bar{A}') < \omega(\bar{A})$ ce qui est impossible puisque $\bar{A}$ est optimal. □

Proposition 2.2 *Si* $\bar{A} = A_{n-1,m}\begin{pmatrix} s_n \\ \epsilon \end{pmatrix}$, *alors l'alignement* $A_{n-1,m}$ *est un alignement optimal entre* $s_{..n-1}$ *et* t.

Preuve : Identique à la preuve de la proposition 2.1. □

Proposition 2.3 *Si* $\bar{A} = A_{n-1,m-1}\begin{pmatrix} s_n \\ t_m \end{pmatrix}$, *alors l'alignement* $A_{n-1,m-1}$ *est un alignement optimal entre* $s_{..n-1}$ *et* $t_{..m-1}$.

Preuve : Identique à la preuve de la proposition 2.1. □

On calcule le coût de l'alignement optimal $\bar{A}$ en choisissant le coût minimal parmi les trois alternatives possibles :

$$\omega(\bar{A}) = \min\left\{\omega(A_{n,m-1}) + \omega\begin{pmatrix}\epsilon\\ t_m\end{pmatrix}, \omega(A_{n-1,m}) + \omega\begin{pmatrix}s_n\\ \epsilon\end{pmatrix}, \omega(A_{n-1,m-1}) + \omega\begin{pmatrix}s_n\\ t_m\end{pmatrix}\right\} \quad (2.1)$$

Les coûts $\omega(A_{n,m-1}), \omega(A_{n-1,m})$ et $\omega(A_{n-1,m-1})$ se calculent de la même manière grâce aux alignements optimaux $A_{n,m-2}$, $A_{n-1,m-1}$, $A_{n-1,m-2}$, $A_{n-1,m-1}$, $A_{m-2,m}$, $A_{n-2,m-1}, \cdots$

Chaque alignement optimal $A_{i,j}$ (avec $i \geq 1$ et $j \geq 1$) se calcule donc grâce aux trois alignements optimaux $A_{i-1,j}$, $A_{i,j-1}$ et $A_{i-1,j-1}$. Son coût est donné par la formule de récurrence de l'équation 2.2.

$$\omega(A_{i,j}) = \min\left\{\omega(A_{i,j-1}) + \omega\begin{pmatrix}\epsilon\\ t_j\end{pmatrix}, \omega(A_{i-1,j}) + \omega\begin{pmatrix}s_i\\ \epsilon\end{pmatrix}, \omega(A_{i-1,j-1}) + \omega\begin{pmatrix}s_i\\ t_j\end{pmatrix}\right\} \quad (2.2)$$

L'amorçage de la récurrence est réalisé grâce aux alignements $A_{0,j}$ et $A_{i,0}$ définis par :

- $A_{0,j}$ est l'alignement optimal entre ϵ et $t_{..j}$, avec $j \geq 1$.
  C'est-à-dire $A_{0,j} = \begin{pmatrix}\epsilon\\ t_1\end{pmatrix}\begin{pmatrix}\epsilon\\ t_2\end{pmatrix}\ldots\begin{pmatrix}\epsilon\\ t_j\end{pmatrix}$ et donc $\omega(A_{0,j}) = \sum_{k=1}^{j}\omega\begin{pmatrix}\epsilon\\ t_k\end{pmatrix}$, ou de manière récurrente, $\omega(A_{0,j}) = \omega(A_{0,j-1}) + \omega\begin{pmatrix}\epsilon\\ t_j\end{pmatrix}$.
- $A_{i,0}$ est l'alignement optimal entre $s_{..i}$ et ϵ, avec $i \geq 1$.
  C'est-à-dire $A_{i,0} = \begin{pmatrix}s_1\\ \epsilon\end{pmatrix}\begin{pmatrix}s_2\\ \epsilon\end{pmatrix}\ldots\begin{pmatrix}s_i\\ \epsilon\end{pmatrix}$ et donc $\omega(A_{i,0}) = \sum_{k=1}^{i}\omega\begin{pmatrix}s_k\\ \epsilon\end{pmatrix}$, ou de manière récurrente, $\omega(A_{i,0}) = \omega(A_{i-1,0}) + \omega\begin{pmatrix}s_i\\ \epsilon\end{pmatrix}$.
- Par convention, $A_{0,0}$ est la suite vide et $\omega(A_{0,0}) = 0$.

Remarque 2.3 *Les algorithmes de programmation dynamique sont souvent basés sur une relation de récurrence du type de celle de l'équation 2.2. Une telle équation ne met pas très bien en évidence les mécanismes d'évolution sous-jacents. Il est parfois préférable de présenter les algorithmes de programmation dynamique sous la forme d'*automates finis *(de* transducteurs*) [SM95, Lef96]. Cette approche possède l'avantage de mieux mettre en évidence les relations entre les valeurs à calculer et de faciliter toute adaptation de l'algorithme à un autre modèle d'évolution. À titre d'exemple, le chapitre 4 contient la présentation, sous forme de transducteur, d'un algorithme de* programmation dynamique cyclique (wraparound dynamic programming).

Soit $c[0 \to n, 0 \to m]$ le tableau des coûts des alignements. Pour chaque cellule i, j, telle que $0 \leq i \leq n$ et $0 \leq j \leq m$, il est défini par : $c[i, j] = \omega(A_{i,j})$.

Le calcul de toutes les valeurs du tableau c est réalisé grâce à deux boucles imbriquées qui parcourent tous les indices i et j. La fonction CALCULCOUTALIGNEMENTS réalise ce travail, son algorithme est donné à la figure 2.6.

CALCULCOUTALIGNEMENTS(s, n, t, m, ω)

1 $c[0,0] \leftarrow 0$
2 **Pour** $i \leftarrow 1$ **Jusque** n
3 **Faire** $c[i,0] \leftarrow c[i-1,0] + \omega\begin{pmatrix} s_i \\ \epsilon \end{pmatrix}$
4 **Pour** $j \leftarrow 1$ **Jusque** m
5 **Faire** $c[0,j] \leftarrow c[0,j-1] + \omega\begin{pmatrix} \epsilon \\ t_j \end{pmatrix}$
6 **Pour** $i \leftarrow 1$ **Jusque** n
7 **Faire** $c[i,j] \leftarrow \min\left\{c[i,j-1] + \omega\begin{pmatrix} \epsilon \\ t_j \end{pmatrix}, c[i-1,j] + \omega\begin{pmatrix} s_i \\ \epsilon \end{pmatrix}, c[i-1,j-1] + \omega\begin{pmatrix} s_i \\ t_j \end{pmatrix}\right\}$
8 **Retourner** c

FIG. 2.6 - *Algorithme de calcul des coûts de tous les alignements optimaux par programmation dynamique*

Après application de cet algorithme, la valeur de $c[n,m] = \omega(A_{n,m}) = \omega(\bar{A})$ est le coût d'un alignement optimal et donc par définition : $R_\omega(s,t) = c[n,m]$.

Exemple 2.6 *L'alignement optimal de l'exemple 2.5 a été trouvé grâce à l'algorithme de programmation dynamique* CALCULCOUTALIGNEMENTS. *Les valeurs de c sont représentées dans le tableau 2.2.*

$i\backslash j$		0	1	2	3	4	5	6	7	8	9	10	11	12	13
	$s\backslash t$		A	T	G	G	A	C	T	G	A	A	A	G	T
0		0	2	4	6	8	10	12	14	16	18	20	22	24	26
1	A	1	0	2	4	6	8	10	12	14	16	18	20	22	24
2	C	2	1	1	3	5	7	8	10	12	13	15	17	19	21
3	G	3	2	2	1	3	5	7	9	10	11	13	15	17	19
4	G	4	3	3	2	1	3	5	7	9	10	12	14	15	17
5	A	5	4	4	3	2	1	3	5	7	9	10	12	14	16
6	T	6	5	4	4	3	2	2	3	5	7	9	11	13	14
7	C	7	6	5	5	4	3	2	3	4	5	7	9	11	14
8	T	8	7	6	6	5	4	3	2	4	5	6	8	10	11
9	A	9	8	7	7	6	5	4	3	3	4	5	6	8	10
10	A	10	9	8	8	7	6	5	4	4	3	4	5	7	9
11	G	11	10	9	8	8	7	6	5	4	4	4	5	5	7
12	C	12	11	10	9	9	8	7	6	5	5	5	5	6	6

TAB. 2.2 - *Programmation dynamique : tableau des coûts $c[i,j]$ pour les séquences s et t de l'exemple 2.5*

La mesure de ressemblance entre s et t est $R_\omega(s,t) = c[12,13] = 6$.

La complexité en temps de l'algorithme CALCULCOUTALIGNEMENTS est $\theta(n.m)$. La complexité en espace est également $\theta(n.m)$.

Lorsque toutes les valeurs $c[i,j]$ ont été calculées, il est facile de construire un alignement optimal. Il se construit par récurrence de la dernière paire alignée vers la première.

L'alignement que l'on recherche est $A_{n,m}$. Grâce à la valeur de $c[n,m]$ et aux valeurs de $c[n-1,m]$, $c[n,m-1]$ et $c[n-1,m-1]$, on sait trouver la dernière paire alignée de $A_{n,m}$:

- Si $c[n,m] = c[n,m-1] + \omega\begin{pmatrix}\epsilon\\ t_m\end{pmatrix}$ alors $A_{n,m} = A_{n,m-1}\begin{pmatrix}\epsilon\\ t_m\end{pmatrix}$.
- Si $c[n,m] = c[n-1,m] + \omega\begin{pmatrix}s_n\\ \epsilon\end{pmatrix}$ alors $A_{n,m} = A_{n-1,m}\begin{pmatrix}s_n\\ \epsilon\end{pmatrix}$.
- Si $c[n,m] = c[n-1,m-1] + \omega\begin{pmatrix}s_n\\ t_m\end{pmatrix}$ alors $A_{n,m} = A_{n-1,m-1}\begin{pmatrix}s_n\\ t_m\end{pmatrix}$.

Si plus d'une des trois conditions est satisfaite, alors il existe plus d'un alignement optimal et il suffit d'en choisir arbitrairement un.

Soit $A_{i,j}$ l'alignement choisi et $\begin{pmatrix}a\\ b\end{pmatrix}$ la dernière paire alignée : $A_{n,m} = A_{i,j}\begin{pmatrix}a\\ b\end{pmatrix}$ avec $i \in \{n-1,n\}, j \in \{m-1,m\}, a \in \{\epsilon, s_n\}$ et $b \in \{\epsilon, t_m\}$.

En procédant de la même manière, on détermine la dernière paire alignée $\begin{pmatrix}a'\\ b'\end{pmatrix}$ de $A_{i,j}$: $A_{i,j} = A_{i',j'}\begin{pmatrix}a'\\ b'\end{pmatrix}$. Deux paires de $A_{n,m}$ sont alors connues : $A_{n,m} = A_{i',j'}\begin{pmatrix}a'\\ b'\end{pmatrix}\begin{pmatrix}a\\ b\end{pmatrix}$. De proche en proche, toutes les paires alignées de $A_{n,m}$ sont ainsi construites.

Cela revient, dans le tableau c, à rechercher le chemin qui mène de la position $[n,m]$ à la position $[0,0]$ en suivant, pour chaque arc, la direction imposée par les valeurs de c. Les trois directions possibles sont $\leftarrow$, $\uparrow$ et $\nwarrow$ et chaque déplacement est limité à une des trois positions adjacentes (voir figure 2.7).

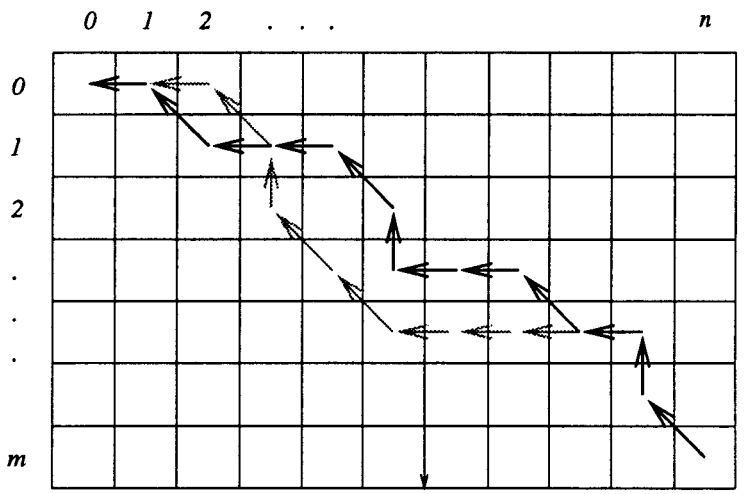


FIG. 2.7 - *Chemins qui correspondent aux alignements optimaux*

Chaque déplacement correspond à une paire alignée :

$\leftarrow$ pour une paire $\begin{pmatrix}\epsilon\\ t_j\end{pmatrix}$.

$\uparrow$ pour une paire $\begin{pmatrix} s_i \\ \epsilon \end{pmatrix}$.

$\nwarrow$ pour une paire $\begin{pmatrix} s_i \\ t_j \end{pmatrix}$.

Le chemin parcourt les paires alignées dans l'ordre inverse de celui de l'alignement. S'il existe plusieurs alignements optimaux, chacun d'eux correspond à un chemin différent. La recherche du chemin correspondant à un alignement optimal se fait en temps $\theta(p)$ où p est le nombre de paires de l'alignement. Puisque $p \leq n + m$, la recherche du chemin se fait en temps $O(n + m)$.

Exemple 2.7 *L'application de ce processus au tableau c de l'exemple 2.6 nous donne l'alignement optimal*

$$A_{12,13} = \begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} C \\ T \end{pmatrix}\begin{pmatrix} G \\ G \end{pmatrix}\begin{pmatrix} G \\ G \end{pmatrix}\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} \epsilon \\ C \end{pmatrix}\begin{pmatrix} T \\ T \end{pmatrix}\begin{pmatrix} C \\ G \end{pmatrix}\begin{pmatrix} T \\ A \end{pmatrix}\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} A \\ A \end{pmatrix}\begin{pmatrix} G \\ G \end{pmatrix}\begin{pmatrix} C \\ T \end{pmatrix}$$

de l'exemple 2.5 (voir tableau 2.3).

$i\backslash j$		0	1	2	3	4	5	6	7	8	9	10	11	12	13
	$s\backslash t$		A	T	G	G	A	C	T	G	A	A	A	G	T
0		**0**	2	4	6	8	10	12	14	16	18	20	22	24	26
1	A	1	**0**	2	4	6	8	10	12	14	16	18	20	22	24
2	C	2	1	**1**	3	5	7	8	10	12	13	15	17	19	21
3	G	3	2	2	**1**	3	5	7	9	10	11	13	15	17	19
4	G	4	3	3	2	**1**	3	5	7	9	10	12	14	15	17
5	A	5	4	4	3	2	**1**	**3**	5	7	9	10	12	14	16
6	T	6	5	4	4	3	2	2	**3**	5	7	9	11	13	14
7	C	7	6	5	5	4	3	2	3	**4**	5	7	9	11	14
8	T	8	7	6	6	5	4	3	2	4	**5**	6	8	10	11
9	A	9	8	7	7	6	5	4	3	3	4	**5**	6	8	10
10	A	10	9	8	8	7	6	5	4	4	3	4	**5**	7	9
11	G	11	10	9	8	8	7	6	5	4	4	4	5	**5**	7
12	C	12	11	10	9	9	8	7	6	5	5	5	5	6	**6**

TAB. 2.3 - *Chemin qui détermine l'alignement optimal de l'exemple 2.5*

2.5.4 Alignement local

Un problème annexe à celui de l'alignement de deux séquences est l'alignement local d'une courte séquence dans une séquence plus longue. Soit $s = s_1 s_2 \dots s_n$ et $t = t_1 t_2 \dots t_m$ deux séquences de $\mathcal{N}^+$ avec $n < m$ et ω une fonction de coût. On désire trouver le facteur $t_{k..l}$ de t qui minimise $R_\omega(s, t_{k..l})$. C'est-à-dire que l'on désire trouver le facteur de t qui ressemble le plus à la séquence s. Le problème est identique à celui de la recherche du meilleur alignement global si ce n'est que l'alignement peut commencer à une position $k \neq 1$ de t et se terminer à une position $l \neq m$ de t. La formule de récurrence pour résoudre ce problème est exatement

la même que dans le cas global (équation 2.2), le fait que l'alignement puisse commencer à n'importe quelle position k se traduit par un coût $\omega(A_{0,j}) = 0$, pour toute position j de t. Les valeurs $c[0,j]$ du tableau de programmation dynamique sont donc toutes nulles. L'algorithme de la figure 2.6 est simplement modifié en remplaçant la ligne 5 par :

$$5 \quad \textbf{Faire } c[0,j] \leftarrow 0$$

L'alignement optimal se construit à partir du tableau c de la même manière que dans le cas global, si ce n'est que la valeur à considérer n'est pas $c[n,m]$ mais plutôt la valeur $c[n,l] = \min\{c[n,j] | 1 \leq j \leq m\}$. Cela exprime que l'alignement se termine à la position l de t. On recherche donc, dans le tableau c, le chemin qui mène de la position $[n,l]$ à une position $[0,k-1]$. Le facteur de t qui ressemble le plus à s est alors $t_{k..l}$.

Exemple 2.8 *Considérons* $t =$ *ACTACTAATGACCTAGTTTAATCCG et* $s =$ *TATAAT. On désire rechercher le facteur de t qui ressemble le plus au motif TATAAT. Prenons comme fonction de coût celle qui définit la distance de Levenshtein (remarque 2.2):* $\omega\binom{a}{a} = 0$, $\omega\binom{a}{\epsilon} = \omega\binom{\epsilon}{a} = \omega\binom{a}{b} = 1$, *pour tous les* $a, b \in \mathcal{N}$ *et* $a \neq b$.

Le tableau 2.4 représente les valeurs $c[i,j]$. *Deux valeurs minimales sont trouvées sur la dernière ligne aux colonnes 9 et 22, ce qui suggère que deux facteurs de t "ressemblent" à s :* $t_{3..9} =$ *TAcTAAT et* $t_{17..22} =$ *TtTAAT.*

Remarque 2.4 *La méthode d'alignement local permet de trouver le facteur qui ressemble le plus au motif et non tous les facteurs qui ressemblent au motif. Si dans l'exemple, deux facteurs ressemblant au motif ont été trouvés, il s'agit d'une pure coïncidence. Ils ont tous deux un score optimal d'alignement avec le motif (le score est de 1).*

Si l'on désire obtenir tous les mots ressemblant au motif, alors il faut établir un seuil sur le score d'alignement et rechercher, dans la dernière ligne du tableau de programmation dynamique c, tous les scores qui sont sous ce seuil. La définition du seuil introduit donc de l'arbitraire dans l'alignement, elle influe directement sur la notion de ressemblance entre le motif et les facteurs recherchés.

2.6 Notations

Nous précisons ici quelques notations qui seront utilisées dans les trois derniers chapitres de la thèse.

Pour commencer, rappelons que $\mathcal{N} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ est l'*alphabet des nucléotides*, toutes les séquences d'ADN sont des mots sur $\mathcal{N}$.

Pour faciliter la compréhension d'un alignement, on le représente souvent simplement en plaçant les deux séquences l'une au-dessus de l'autre de telle sorte que les bases alignées soient l'une en face de l'autre. Des tirets (-) sont introduits dans les séquences pour représenter les insertions et les délétions. De cette façon, l'alignement

$$\binom{\mathtt{A}}{\mathtt{A}}\binom{\mathtt{C}}{\epsilon}\binom{\epsilon}{\mathtt{T}}\binom{\mathtt{G}}{\mathtt{G}}\binom{\mathtt{G}}{\epsilon}\binom{\mathtt{A}}{\epsilon}\binom{\mathtt{T}}{\mathtt{G}}\binom{\mathtt{C}}{\mathtt{A}}\binom{\epsilon}{\mathtt{C}}\binom{\mathtt{T}}{\mathtt{T}}\binom{\epsilon}{\mathtt{G}}\binom{\epsilon}{\mathtt{A}}\binom{\mathtt{A}}{\mathtt{A}}\binom{\mathtt{A}}{\mathtt{A}}\binom{\mathtt{G}}{\mathtt{G}}\binom{\mathtt{C}}{\epsilon}\binom{\epsilon}{\mathtt{T}}$$

est représenté par :

```
AC-GGATC-T--AAGC-
A-TG--GACTGAAAG-T
```

$i\backslash j$		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	$s\backslash t$		A	C	T	A	C	T	A	A	T	G	A	C	C	T	A	G	T	T	T	A	A	T	C	C	G
0		0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0
1	T	1	1	1	**0**	1	1	0	1	1	0	1	1	1	1	0	1	1	**0**	0	0	1	1	0	1	1	1
2	A	2	1	2	1	**0**	**1**	1	0	1	1	1	1	2	2	1	0	1	1	**1**	1	0	1	1	1	2	2
3	T	3	2	2	2	1	1	**1**	1	1	1	2	2	2	3	2	1	1	1	1	**1**	1	1	1	2	2	3
4	A	4	3	3	3	2	2	2	**1**	1	2	2	2	3	3	3	2	2	2	2	2	**1**	1	2	2	3	3
5	A	5	4	4	4	3	3	3	2	**1**	2	3	2	3	4	4	3	3	3	3	3	2	**1**	2	3	3	4
6	T	6	5	5	4	4	4	3	3	2	**1**	2	3	3	4	4	4	4	3	3	3	3	2	**1**	2	3	4

TAB. 2.4 - *Alignement local de s = TATAAT et t = ACTACTAATGACCTAGTTTAATCCG*

Pour bien mettre en évidence les différences entre les deux séquences, on utilise la distinction entre les majuscules et les minuscules pour faire ressortir les paires de coïncidence :

```
Ac-Ggatc-T--AAGc-
A-tG--gacTgaAAG-t
```

On se permettra d'utiliser quelques artifices de présentation (caractères gras, italiques, soulignés...) pour faire ressortir au mieux les informations pertinentes des alignements.

Enfin, d'un point de vue codage, puisque $\#\mathcal{N} = 4$, toute base peut être codée sur deux bits. Soit $NUC : \mathcal{N} \to \mathcal{B}^+$ le code auto-délimité qui associe deux bits à chaque base de la façon suivante :

$$NUC(\mathtt{A}) = 00 \quad NUC(\mathtt{C}) = 01 \quad NUC(\mathtt{G}) = 10 \quad NUC(\mathtt{T}) = 11$$

Le code homomorphe $NUC^* : \mathcal{N}^* \to \mathcal{B}^*$ qui applique $s_1 s_2 \ldots s_n$ sur $NUC^*(s_1 s_2 \ldots s_n) = NUC(s_1)NUC(s_2)\ldots NUC(s_n)$ est utilisable pour coder n'importe quelle séquence d'ADN.

Chapitre 3

Compression de séquences nucléiques

Puisque les concepts de compression et de compréhension sont intimement liés, la problématique de compression de séquences d'ADN est très intéressante. Elle constitue une nouvelle approche de l'analyse de séquences.

Ce chapitre tente un rapide survol des méthodes de compression appliquées aux séquences nucléiques. La compression doit seulement être vue ici comme outil d'analyse, elle n'est nullement utilisée pour réduire l'espace de stockage ou le temps de transmission des séquences. Même si une méthode de compression appliquée à une séquence produit un faible taux, voir même un taux négatif (un allongement), l'expérimentation est riche en apprentissage. Il est toujours intéressant de savoir pourquoi une méthode fonctionne ou ne fonctionne pas sur une séquence.

La première idée qui vient à l'esprit dans le contexte d'analyse de séquences nucléiques par compression est l'application des méthodes de compression existantes aux séquences. Cette idée fait l'objet de la première section du chapitre. Un constat frappant est l'inadéquation des méthodes classiques de compression (codage de Huffman, codage arithmétique, LZ77 et LZ78). Les raisons à cela sont exposées, ce qui met en évidence le besoin de méthodes dédiées, c'est-à-dire de méthodes qui exploitent des propriétés propres aux séquences biologiques.

Les sections suivantes exposent les quelques méthodes dédiées à l'ADN. Nous nous restreignons aux méthodes exploitant les répétitions exactes, l'exploitation des répétitions approximatives fait l'objet des deux chapitres suivants. Les méthodes sont exposées par ordre chronologique. Le but n'est pas de les rendre directement implémentables mais plutôt d'en présenter les points marquants.

La première méthode exposée, celle d'étude de complexité linguistique de Trifonov [Tri90] n'est pas à proprement parler une méthode de compression. C'est un préalable au reste. La raison de sa présence ici est son analogie avec la complexité de Kolmogorov : elle définit une complexité des séquences d'ADN dont le but est de marquer la différence entre les séquences aléatoires et les séquences non aléatoires. Vient ensuite la méthode de significativité algorithmique de Milosavljević et Jurka [MJ93, Mil93]. Ils proposent une méthode de compression basée sur l'exploitation des répétitions dans les séquences. La méthode concerne non seulement la recherche de la plus courte description d'une séquence (par rapport à un schéma de codage particulier) mais également un test permettant de conjecturer si la séquence est aléatoire ou non avec un seuil de significativité déterminé. Bien que développée indépendam-

ment, la méthode BIOCOMPRESS de Grumbach et Tahi [GT93a, GT93b, GT95] est semblable à la méthode de Milosavljević et Jurka. Les deux méthodes ont été publiées pratiquement en même temps. L'innovation proposée par BIOCOMPRESS réside dans l'exploitation d'un nouveau type de régularité présente dans les séquences d'ADN : les *palindromes génétiques*. La version BIOCOMPRESS-2 intègre également une compression à l'aide d'un codage arithmétique. Malheureusement, la méthode n'est pas toujours capable d'exploiter les longues répétitions. Cette lacune est fâcheuse car la détection de longues répétitions est un problème important dans le contexte de l'étude d'évolution de séquences. La méthode suivante, CFACT, de Rivals concentre ses efforts sur les longues répétitions [Riv94, Riv96, RDDD96, RDDD97]. Elle possède un équivalent pour les palindromes génétiques : CPAL. Le chapitre se termine avec la méthode CBI développée par Loewenstern, Hirsh, Yianilos et Noordewier [LHYN95]. Elle concerne la classification de séquences d'ADN à l'aide de la compression. C'est une utilisation légèrement en marge du reste. Elle montre que la compression peut être utilisée pour résoudre des problèmes très diversifiés d'analyse de séquences.

3.1 Inadéquation des méthodes classiques de compression

Il est surprenant de constater que les méthodes classiques de compression (codage de Huffman, codage arithmétique, LZ77 et LZ78) sont quasiment incapables de comprimer les séquences nucléiques. Bien entendu, ce constat suppose que l'on compare la taille de la séquence comprimée avec la taille de la séquence initiale, codée sur l'alphabet $\mathcal{B}$ à l'aide du code NUC^* (voir remarque 2.1, page 29). Il ne faut pas commettre l'erreur courante de comparer la taille des deux fichiers informatiques "avant compression" et "après compression" alors que la séquence initiale était codée une base par octet.

De nombreuses expérimentations ont été réalisées sur des séquences nucléiques [GT93b, Riv94, Riv96] et les taux de compression sont faibles, en moyenne inférieurs à 1%. Dans beaucoup de cas même, la séquence est allongée plutôt que raccourcie. Ce comportement semble très étrange lorsque l'on sait que ces méthodes atteignent habituellement 50% de compression pour les fichiers courants (textes en langage naturel, programmes sources, images,...). Nous allons voir, au cas par cas, pourquoi il en est ainsi.

3.1.1 Les méthodes statistiques

Pour qu'une méthode statistique ait une chance de comprimer une séquence nucléique, la répartition des quatre bases `A`, `C`, `G` et `T` doit être disproportionnée. Dans les séquences d'ADN, les fréquences d'apparition des bases sont similaires, il n'y a pas de différences marquantes entre elles.

Sans aucune compression, chaque base est codée sur deux bits. Une compression n'est possible par une méthode statistique que si l'entropie est inférieure à 2.

Le codage de Huffman : Puisque que le codage de Huffman attribue un nombre entier de bits à chaque base, deux bases de fréquences similaires sont souvent codées en utilisant le même nombre de bits. De ce fait, une faible différence de répartition n'est pas exploitée. Au contraire, une telle disproportion peut provoquer un allongement considérable de la séquence (10% d'allongement en moyenne [Riv96]).

Cette remarque est également valable pour les codages de Huffman d'ordres supérieurs.

Le codage arithmétique : Ici, chaque base est codée sur un nombre fractionnaire de bits. La moindre disproportion dans la répartition des bases est exploitée par le codage. Pour des séquences suffisamment longues, la longueur moyenne des mots code est très proche de l'entropie qui est elle-même inférieure ou égale à 2.

Le codage arithmétique produit la meilleure compression parmi les quatre méthodes classiques. La moindre propriété statistique concernant les bases est exploitée (exemple de propriété : les *isochores* qui sont des fragments d'ADN le long desquels la proportion de C+G est très différente de la proportion de A+T). Pour exploiter une propriété statistique locale, telle que les isochores, il faut utiliser un codage arithmétique adaptatif (voir partie I, section 2.3.1.3.3).

Des codages arithmétiques d'ordres supérieurs peuvent être utilisés, ils exploitent des biais dans l'utilisation des dinucléotides, des codons, des oligonucléotides d'une façon générale.

Le compresseur arithmétique qui donne les meilleurs résultats sur l'ADN est un compresseur adaptatif qui exploite les biais de répartition des dinucléotides (d'ordre 1, d'après la définition donnée page 53, section 2.3.1.4).

Il faut cependant remarquer que le taux de compression moyen, des expérimentations réalisées dans [Riv96] sur des séquences de diverses espèces, à l'aide du codage arithmétique, n'atteint pas 4% !

3.1.2 Les méthodes par substitutions de facteurs

Les méthodes par substitutions exploitent les répétitions de facteurs. La deuxième occurrence d'un facteur (de gauche à droite) est codée par un "pointeur" qui référence la première occurrence. Dans les séquences d'ADN, les facteurs répétés peuvent être très longs. De plus, les occurrences d'un facteur répété peuvent être très distantes l'une de l'autre. Ces deux constats empêchent les deux méthodes LZ77 et LZ78 de comprimer des séquences d'ADN. **Les méthodes par substitutions de facteurs sont quasiment toujours incapables de comprimer des séquences d'ADN.**

La méthode LZ77 : **C'est la méthode qui produit la moins bonne compression parmi les quatre méthodes classiques** (40% d'allongement en moyenne [Riv96]). C'est pourtant le compresseur le plus utilisé pour des données à caractère général (PKZIP et ARJ sous MS-DOS, GZIP sous UNIX,...) ! L'explication de ce phénomène est assez simple. La fenêtre qui coulisse sur le texte de gauche à droite est de taille limitée. Cette approche est bien adaptée à la plupart des textes mais pas aux séquences d'ADN : si deux occurrences d'un facteur répété sont trop éloignées (plusieurs centaines ou milliers de bases), LZ77 n'est pas capable d'exploiter la répétition car la première occurrence ne fait plus partie de la fenêtre lorsque la deuxième est rencontrée.

La méthode LZ78 : La perte en compression de LZ78 est généralement plus faible que celle de LZ77 (11% d'allongement en moyenne [Riv96]).

Les séquences d'ADN contiennent de nombreuses répétitions de mots de petites tailles : pour une longueur de séquence suffisante, pratiquement tous les mots de petites tailles sont présents et répétés. De ce fait, le dictionnaire est rapidement rempli par des répétitions insignifiantes avec, pour conséquence, une forte augmentation de la longueur du codage des indices dans le dictionnaire [Riv96]. Il en résulte une mauvaise compression.

Le compresseur LZ78 parvient rarement à exploiter une longue répétition car elle se trouve souvent décomposée, dans le dictionnaire, selon les facteurs qui la composent. Ils sont en effet plus courts et sont souvent répétés également dans d'autres contextes que celui de la longue répétition.

D'autre part, d'un point de vue biologique, la décomposition de la séquence en petits facteurs répétés n'est intéressante que pour des mots spécifiques connus tels que la "TATA box" (signal particulier présent dans la majorité des promoteurs de transcription) par exemple. De tels motifs peuvent être recherchés par des algorithmes spécifiques de "string-matching". De plus, le biologiste préfère connaître la longue répétition qui traduit un seul événement mutationnel [RDDD97] plutôt que la série de courtes répétitions qui ne traduit rien.

Dans le cadre d'utilisation de la compression comme outil d'analyse, les quatre méthodes classiques se montrent bien impuissantes : la seule conclusion que l'on puisse tirer est qu'elles ne sont pas capables de "comprendre" la structure des séquences.

3.2 Complexité linguistique (E.N. Trifonov, 1990)

La méthode développée par E.N. Trifonov dans l'article [Tri90] n'est pas une méthode de compression mais elle est intéressante dans la mesure ou elle définit une complexité des séquences d'ADN. Son rôle, tout comme celui de la complexité de Kolmogorov, est de marquer la différence entre les séquences "aléatoires" et les séquences "non-aléatoires". Cette complexité est basée sur le vocabulaire des séquences, c'est-à-dire sur la quantité de mots utilisés dans les séquences. Elle s'inspire d'une analogie avec les textes en langage naturel : tout texte qui a un sens utilise uniquement un sous-ensemble limité (un *vocabulaire*) de toutes les combinaisons possibles de lettres. Pour des textes suffisamment longs, un certain nombre de mots sont donc répétés. Un texte complexe, c'est-à-dire écrit par un écrivain professionnel devrait posséder un vocabulaire plus riche qu'un texte simple écrit par une personne non qualifiée. Dès lors, un texte complexe d'un point de vue linguistique devrait posséder moins de répétitions qu'un texte simple.

Toute séquence d'ADN est caractérisée par son vocabulaire, c'est-à-dire par l'utilisation qu'elle fait de tous les mots de longueur 1, 2, 3,...

Soit $S = s_1 s_2 \dots s_n \in \mathcal{N}^+$ une séquence et soit $l > 0$ une longueur de mots.

Définition 3.1 *L'usage du vocabulaire des mots de longueur l de S, noté $U_l(S)$, est le quotient entre le nombre de mots de longueur l rencontrés dans S et le nombre maximal de mots de longueur l que l'on peut rencontrer dans une séquence de même longueur que S. Les mots comptabilisés sont chevauchants.*

Exemple 3.1 *Soit $S1$ = ACGGTAAGCTGATTCCA une séquence d'ADN de longueur 17.*
Tous les nucléotides sont utilisés dans la séquence : $U_1(S1) = \frac{17}{17} = 1$.
Tous les dinucléotides sont utilisés dans la séquence : $U_2(S1) = \frac{16}{16} = 1$.

Exemple 3.2 *Soit $S2$ = ACACACACACACACACA une autre séquence de longueur 17.*
Tous les nucléotides ne sont pas utilisés : $U_1(S2) = \frac{9}{17}$ car sur les 17 mots possibles de longueur 1, il en manque 8 (4 G et 4 T).
Seuls les deux dinucléotides AC et CA sont utilisés dans la séquence. L'usage du vocabulaire des dinucléotides de $S2$ est $U_2(S2) = \frac{2}{16} = \frac{1}{8}$.

La *complexité linguistique* de la séquence S est une synthèse des usages de vocabulaire pour toutes les longueurs possibles.

Définition 3.2 *La* complexité linguistique *de la séquence S est la grandeur :*

$$C(S) = \prod_{l=1}^{n-1} U_l(S)$$

Exemple 3.3 *Soit $S1$ = ACGGTAAGCTGATTCCA. La complexité linguistique de $S1$ est* $C(S1) = \frac{17}{17} \cdot \frac{16}{16} \ldots \frac{2}{2} = 1$.

Exemple 3.4 *Soit $S2$ = ACACACACACACACACA. La complexité linguistique de $S2$ est* $C(S2) = \frac{9}{17} \cdot \frac{2}{16} \ldots \frac{2}{2} \approx 0$.

Plus $C(S)$ est proche de 1, plus la séquence est complexe d'un point de vue linguistique.

Le calcul de $C(S)$ se fait en temps $O(n^3)$ car il comporte trois boucles imbriquées : celle qui parcourt les longueurs l de mots, celle qui parcourt tous les mots de S pour une longueur l et celle qui parcourt tous les symboles d'un mot. La méthode n'est donc pas utilisable pour rechercher la complexité de longues séquences. Dès lors, l'auteur a préféré calculer les complexités linguistiques de fenêtres de longueurs 20 et de prendre comme complexité pour la séquence complète, la moyenne des complexités des fenêtres.

Il a expérimenté sa méthode sur un grand nombre de séquences de divers types et de diverses espèces. Le résultat le plus marquant est la **grande complexité des séquences codantes (séquences qui sont traduites en protéines) par rapport à la complexité des séquences non codantes.**

3.3 Significativité algorithmique (A. Milosavljević et J. Jurka, 1993)

À notre connaissance, A. Milosavljević et J. Jurka ont été les premiers, en 1993, à s'intéresser à l'analyse de séquences d'ADN par compression (article [MJ93]). Ils utilisent la compression pour détecter des séquences d'ADN simples. L'idée est de remplacer la deuxième occurrence d'un facteur répété par un pointeur qui référence la première occurrence. Ce référençage n'est effectué que si cela apporte une réelle compression, dans le cas contraire la deuxième occurrence est codée telle quelle.

Grâce à une relation de programmation dynamique, la recherche de la *longueur minimale d'encodage* utilisant ce schéma de codage est réalisée en temps linéaire. Les auteurs donnent ensuite un test de *significativité algorithmique* leur permettant d'accepter ou de rejeter l'hypothèse de séquence aléatoire avec une significativité précise. Dans [Mil93], A. Milosavljević utilise la significativité algorithmique pour découvrir la similarité entre séquences.

3.3.1 Longueur minimale et significativité algorithmique [MJ93]

Une séquence comprimée est une suite de nucléotides codés tels quels et de pointeurs (pos, lg) où pos est la position de la première occurrence et lg est la longueur du facteur répété.

Exemple 3.5 *La séquence S = AGTCAGTTTT se comprime en S' = AGTC(1,3)(7,3).*

Il n'est pas possible d'utiliser le code NUC^* pour coder les nucléotides car il ne laisse pas la possibilité de réserver un mot code "annonce de pointeur" pour annoncer que ce qui suit est un pointeur (l'annonce de pointeur est un concept analogue à l'annonce de rupture $a_{\mathcal{R}}$, voir chapitre 1). Dès lors, l'annonce de pointeur ainsi que les nucléotides sont tous les cinq codés sur 3 bits. Les deux entiers d'un pointeur (pos, lg) sont codés en format fixe: $\lceil \log n \rceil$ bits pour chacun d'eux (n est la longueur de la séquence). Le coût de codage d'un pointeur est donc $c_p = 3 + 2\lceil \log n \rceil$ bits et le coût de codage d'un nucléotide est $c_{\mathcal{N}} = 3$ bits.

Soit $S = s_1 s_2 \ldots s_n$ une séquence d'ADN. Étant donné le schéma de codage décrit au paragraphe précédent, soit $I(S)$ la *longueur minimale d'encodage* de S. On calcule cette longueur minimale grâce aux longueurs minimales $I(s_{i..})$ d'encodage des suffixes de S par la formule de récurrence suivante (la preuve se trouve dans [Sto88]) :

$$I(s_{i..}) = \min \left\{ c_{\mathcal{N}} + I(s_{i+1..}), c_p + I(s_{i+l(i)..}) \right\}$$

où $l(i)$ est la longueur du plus long facteur qui débute à la position i et qui possède une occurrence à une position $j < i$. Si la lettre à la position i n'apparaît à aucune position $j < i$, alors $l(i) = 0$. Par convention, $I(\epsilon) = 0$.

Cette relation de récurrence suggère que la longueur minimale $I(S) = I(s_{1..})$ soit calculée en temps linéaire par l'algorithme de programmation dynamique qui calcule les $I(s_{i..})$ pour i allant de n à 1. Pour que ce soit possible, les valeurs $l(i)$ doivent être connues. Elles sont calculées dans une phase de pré-traitement par un parcours de gauche à droite de la séquence en utilisant l'*automate des facteurs de* S (*DAWG*, voir [BBE+85]). C'est une structure compacte qui joue le même rôle que l'arbre des suffixes. Son utilisation permet de calculer les valeurs $l(i)$ en temps linéaire; le calcul de $I(S)$ est donc réalisé en un temps global linéaire.

On dispose donc d'un algorithme qui fournit la compression optimale d'une séquence pour le schéma de codage décrit, en temps $O(n)$.

La deuxième contribution importante de A. Milosavljević et J. Jurka est le concept de significativité algorithmique qui permet de réfuter ou d'accepter, grâce à la longueur de la version comprimée d'une séquence, qu'elle ait été générée aléatoirement.

Soit $S = s_1 s_2 \ldots s_n$ une séquence d'ADN. Soit P_0 une distribution de probabilité sur les séquences si on suppose qu'elles sont générées aléatoirement. L'hypothèse nulle H_0 est: "la séquence S a été générée aléatoirement selon la loi P_0".

Soit $I_A(S)$ la longueur, en bits, d'une version comprimée de S à l'aide d'un algorithme A (par exemple, $I_A(S) = I(S)$ est la longueur minimale d'encodage de S décrite précédemment). Alors, le théorème 3.1 permet d'accepter ou de rejeter l'hypothèse H_0 avec un seuil de significativité 2^{-d}.

Théorème 3.1 *Pour toute distribution de probabilité P_0 et pour tout algorithme de compression A :*

$$\Pr(-\log p_0(S) - I_A(S) \geq d) \leq 2^{-d}$$

avec $p_0(S)$ la probabilité de la séquence S selon la loi P_0 [1].

La preuve est omise. Elle se trouve dans [MJ93].

Ce théorème montre qu'il est peu probable, si S suit la loi de probabilité P_0, qu'une version comprimée soit d bits plus courte que $-\log p_0(S)$. Si c'est le cas, alors l'hypothèse H_0

1. $\Pr(E)$ désigne la probabilité pour que l'événement E se produise.

est rejetée avec le seuil de significativité 2^{-d}. La séquence est alors dite *simple.*

Exemple 3.6 *On désire tester si une séquence $S \in \mathcal{N}^n$ suit la distribution uniforme P_0 avec un seuil de significativité* 0.01. *Prenons $d = 7$, on a $2^{-d} = 0.0078 \leq 0.01$. La loi uniforme P_0 donne: $p_0(S) = 4^{-|S|}$, car il existe 4 types de nucléotides, et donc $-\log p_0(S) = 2|S|$. Si un algorithme de compression A fournit une version comprimée de longueur $I_A(S) \leq 2|S| - 7$ alors on réfute l'hypothèse de génération aléatoire uniforme de S avec un seuil de significativité* 0.01.

Les auteurs ont expérimenté la méthode sur la séquence complète du gène humain TPA (*Tissue Plasminogen Activator* [FDRR86]). La séquence a été considérée une fenêtre à la fois avec une largeur de fenêtre de 128 et un chevauchement entre les fenêtres adjacentes de 64. Le seuil de significativité considéré était fixé à 0.01. Quatre segments simples ont été trouvés. Deux d'entre eux étaient déjà signalés dans Genbank.

3.3.2 Similarité entre séquences

Dans [Mil93], A. Milosavljević utilise la significativité algorithmique pour déduire rigoureusement la similarité entre séquences. Une séquence *cible* est dite *similaire* à une séquence *source* si elle peut être codée de manière concise en remplaçant certains facteurs par des pointeurs sur les occurrences des mêmes facteurs dans la séquence source (les pointeurs sont des couples (pos, lg) comme dans la section 3.3.1 mais ici, *pos* désigne une position dans la séquence source). La similarité entre les deux séquences est acceptée ou rejetée avec un seuil de significativité qui dépend de la même manière que dans [MJ93], du nombre minimal de bits économisés.

3.4 BIOCOMPRESS (S. Grumbach et F. Tahi, 1993)

L'algorithme BIOCOMPRESS développé par S. Grumbach et F. Tahi [GT93a] suit un schéma semblable à celui de A. Milosavljević et J. Jurka.

La principale innovation de BIOCOMPRESS est la détection et l'exploitation des *palindromes génétiques* d'une séquence pour la comprimer. Voyons ce qu'est un palindrome génétique.

Dans le chapitre précédent, nous avons mentionné que les bases d'une molécule d'ADN avaient la possibilité de s'apparier chimiquement deux à deux : `A` avec `T` et `C` avec `G`. Ce sont des appariements très stables qui sont responsables de la structure à deux brins complémentaires de l'ADN. Dans les molécules d'ARN, ces appariements sont également possibles : `A` avec `U` et `C` avec `G`[2]. De ce fait, une molécule d'ARN, qui ne possède elle qu'un seul brin, se replie sur elle-même pour apparier des facteurs complémentaires. La figure 3.1 représente le repliement, dans le plan, d'une molécule d'ARN de transfert. Une telle représentation est appelée la *structure secondaire* de la molécule d'ARN. Elle est constituée d'un ensemble de *bras*, formés par l'appariement de deux facteurs complémentaires, et de *boucles*, formées par des facteurs non-appariés. Les deux facteurs appariés d'un bras constituent, dans la molécule originale

2. Une molécule d'ARN peut également contenir des appariements moins stables tels que `A-C` ou `G-U`. Nous nous restreignons, dans la définition d'un palindrome génétique, aux appariements classiques `A-T` et `C-G`.

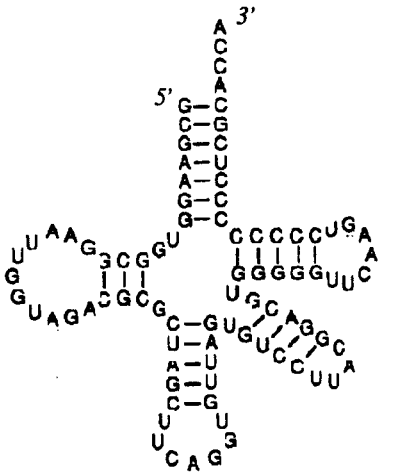


FIG. 3.1 - *Structure secondaire en "feuille de trèfle" d'un ARNt*

d'ADN, un *palindrome génétique.* Un des deux facteurs est le "complémentaire et renversé" de l'autre.

Définition 3.3 *Soit* $Comp : \mathcal{N} \to \mathcal{N}$ *le code défini par :*

$$Comp(A) = T, Comp(C) = G, Comp(G) = C \text{ et } Comp(T) = A$$

Le code homomorphe $Comp^* : \mathcal{N}^* \to \mathcal{N}^* : s_1 s_2 \dots s_n \mapsto Comp(s_1)Comp(s_2)\dots Comp(s_n)$ *applique une séquence sur sa* séquence complémentaire.

Définition 3.4 *Soit* $Rv : \mathcal{N}^* \to \mathcal{N}^* : s_1 s_2 \dots s_n \mapsto s_n \dots s_2 s_1$ *le code qui* renverse *une séquence.*

Nous pouvons maintenant définir formellement un palindrome génétique.

Définition 3.5 *Soit* $S \in \mathcal{N}^+$ *une séquence. Soient* $u = S_{i..i+l-1}$ *et* $u' = S_{j..j+l-1}$ *deux facteurs de longueur* l *de* S *tels que :*

- $u = Comp^*(Rv(u'))$ *: u est le complémentaire renversé de u'.*
- $i + l - 1 \leq j$ *: u et u' ne se chevauchent pas.*

Le couple (u, u') *est un* palindrome génétique *de S. On dit que la longueur du palindrome génétique* (u, u') *est égal à celle de u.*

Exemple 3.7 *Soit*

$$S = accaccaAGTTTAGACCAGTggtcacatACTGGTCTAAACTaaccccccaca$$

Le couple $(AGTTTAGACCAGT, ACTGGTCTAAACT)$ *est un palindrome génétique de S.*

La présence de palindromes génétiques est une propriété qui peut être exploitée pour comprimer les séquences d'ADN : si (u, u') est un palindrome génétique de S, alors le facteur u' peut être remplacé par un pointeur (pos, lg) où pos est l'indice du début de u dans S et lg la longueur de u.

3.4.1 BIOCOMPRESS

Étant donné une séquence $S \in \mathcal{N}^n$, BIOCOMPRESS effectue un parcours de gauche à droite. Pour la position courante i, il recherche dans la partie de la séquence déjà parcourue, le plus long facteur ou palindrome génétique qui concorde avec le facteur débutant en i. Soit pos sa position, avec $1 \leq pos < i$ et soit $lg \geq 0$ sa longueur. Soit (pos, lg) le codage auto-délimité des deux nombres pos et lg. L'algorithme choisit de coder le pointeur (pos, lg) si cela produit une réelle compression, c'est-à-dire si le nombre de bits qu'il occupe est inférieur au nombre de bits du codage $NUC^*(S_{i..i+lg-1})$. Dans ce cas, i est augmenté de lg pour poursuivre la compression juste après la répétition (ou le palindrome). Si ce n'est pas le cas, une seule base est codée grâce à NUC et l'algorithme continue une position plus loin.

La séquence comprimée est donc composée de nucléotides codés à l'aide de NUC et de pointeurs (pos, lg). Remarquons tout d'abord que chaque pointeur doit être précédé d'un bit qui spécifie s'il s'agit d'un facteur répété (0) ou d'un palindrome (1). Remarquons également, pour que le décodage soit possible et non-ambigu, que chaque codage $NUC^*(s_i s_{i+1} \ldots)$ d'une suite de nucléotides doit être précédé du codage du nombre de nucléotides codés. Il en est de même pour chaque suite $1(pos_1, lg_1)0(pos_2, lg_2)\ldots$ de pointeurs: elle doit être précédée du codage du nombre de pointeurs. Ces deux nombres sont codés de manière auto-délimitée (BIOCOMPRESS utilise le code auto-délimité *Fibo*).

Exemple 3.8 *Considérons la séquence S suivante (les chiffres représentent les positions et les majuscules sont utilisées pour mettre en évidence les facteurs répétés et les palindromes) :*

```
12345678901234567890123456789012345678901234567890
aacaAGTACTggaTTTACACAGgaatAGTACTacagtCTGTGTAAAcaga
```

La séquence S contient un facteur répété $(s_{5..10} = s_{27..32})$ et un palindrome génétique: $(s_{14..22}, s_{38..46})$. La séquence comprimée est (les pointeurs sont représentés par les couples (pos, lg) précédés du bit indiquant s'il s'agit d'une copie de facteurs ou d'un palindrome) :

$Fibo(26)NUC^*(AACAAGTACTGGATTTACACAGGAAT)Fibo(1)0(5,6)Fibo(5)NUC^*(ACAGT)$
$Fibo(1)1(14,9)Fibo(4)NUC^*(CAGA)$

La recherche des répétitions dans la séquence est réalisée à l'aide d'un automate des facteurs. Pour des raisons d'économie mémoire, l'automate est limité aux facteurs de longueurs inférieures à h (h est choisit arbitrairement, en pratique $h = 8$). Pour la position courante i, l'automate permet de rechercher toutes les positions d'occurrence du facteur $s_{i..i+h-1}$ dans la partie de la séquence déjà codée. Parmi ces positions, celle qui débute le facteur le plus long est trouvée par comparaison, symbole par symbole des caractères au delà du $h^{\text{ème}}$. Cette recherche séquentielle empêche BIOCOMPRESS d'être linéaire. La recherche des palindromes génétiques est réalisée en utilisant une procédure similaire.

Lorsque BIOCOMPRESS teste s'il vaut mieux coder un facteur par un pointeur (pos, lg) (précédé du bit identifiant le type de pointeur), ou le laisser tel quel, il ne tient pas compte du nombre auto-délimité qui précède toute suite de pointeurs et toute suite utilisant NUC^*. Suite à cette lacune, il se peut que le codage d'un pointeur (pos, lg) entraîne une perte en compression plutôt qu'un gain. Dans [MJ93], le test correct a pu être effectué car toutes les informations sont codées en format fixe. Le choix de laisser un facteur tel quel ou de le remplacer par un pointeur n'interfèrait avec le codage d'aucun autre facteur. Dans le cas de BIOCOMPRESS, c'est différent. Les codages auto-délimités des nombres de pointeurs rendent

tous les tests effectués dépendants les uns des autres. Il n'est donc pas possible de garantir une compression par décisions individuelles position par position.

Par contre, BIOCOMPRESS code chacun des nucléotides sur deux bits alors que trois étaient nécessaires dans [MJ93].

3.4.2 BIOCOMPRESS-2

L'algorithme BIOCOMPRESS-2 présenté dans [GT93b, GT95] est une amélioration de BIOCOMPRESS. Les suites de nucléotides que BIOCOMPRESS avait choisi de coder à l'aide de NUC^* sont maintenant comprimées à l'aide d'un codage arithmétique d'ordre 2. Le gain de compression global est ainsi amélioré.

Dans [GT93b], les auteurs testent BIOCOMPRESS-2 sur un petit nombre de séquences. Il semble que les palindromes jouent un rôle important, en particulier la séquence CHNTXX[3] contient un très long palindrome, le taux de compression qu'elle engendre est de 19.14%. Un autre résultat frappant est le taux de compression de la séquence HUMGHCSA[4] : 34.63% alors que le meilleur résultat obtenu, à l'aide d'une méthode classique, était de 3.11% pour un codage arithmétique d'ordre 2. Cette séquence a la propritété de posséder de nombreuses répétitions. Elles sont ordonnées de telle façon que l'on puisse penser qu'elles proviennent d'une recopie d'un fragment de plusieurs milliers de bases et ensuite d'évolution indépendante des deux fragments. Cette séquence sera reconsidérée, dans le chapitre 5, lors de la recherche de *longues répétitions approximatives.*

3.5 CFACT et CPAL (E. Rivals, 1994)

Nous avons signalé, à la section 3.1.2, que les répétitions contenues dans les séquences d'ADN peuvent être très longues et que leurs occurrences peuvent être très distantes les unes des autres. Il est donc indispensable que les méthodes recherchent les répétitions dans toute la séquence et non dans une fenêtre de taille limitée. C'est le cas de la méthode LZ78, de la longueur minimale d'encodage [MJ93] et de BIOCOMPRESS-2. Malheureusement, LZ78 et BIOCOMPRESS-2 ne sont pas toujours capables d'exploiter les longues répétitions. La méthode CFACT, développée par E. Rivals [Riv94, Riv96, RDDD96, RDDD97], abroge cette limitation.

Les méthodes LZ78 et BIOCOMPRESS-2 parcourent la séquence de gauche à droite. Soit $S \in \mathcal{N}^n$ la séquence. Supposons que la méthode décide de coder le plus long facteur, débutant à la position courante i et déjà rencontré dans la partie comprimée de la séquence, par un pointeur. Ce codage n'est jamais remis en question. Soit A ce facteur et $l = |A|$ sa longueur. Après le codage du pointeur, la compression se poursuit à la position $i+l$ (à la position $i+l+1$ dans le cas de LZ78 car un caractère supplémentaire est codé avec le pointeur). Supposons qu'il existe un plus long facteur répété $B = s_{j..j+l'-1}$ avec $i < j < i + l$ et $l < l'$ (voir figure 3.2). La première occurrence de la répétition de B se trouve à une position $pos < j$. Puisque $j < i + l$, la répétition du facteur B n'est pas exploitée, l'algorithme préfère considérer les répétitions consécutives du facteur A et du facteur C plutôt que la longue répétition de B.

Remarque 3.1 *La méthode de longueur minimale d'encodage [MJ93] exploite bien la longue répétition de B car la programmation dynamique envisage tous les cas et choisit les répétitions les plus intéressantes.*

3. Chloroplaste du tabac, 155855 bases.

4. Human growth hormone and chorionic somatomammotropin genes, 66495 bases.

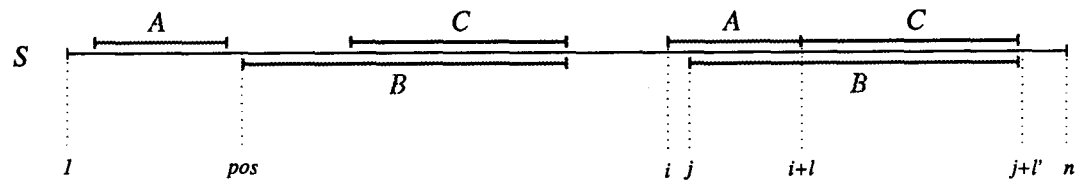


FIG. 3.2 - *LZ78 et* BIOCOMPRESS-2 *n'exploitent pas la longue répétition du facteur B*

E. Rivals a développé, en 1994 [Riv94], un algorithme appelé COMPGD+ qui exploite les répétitions non pas dans l'ordre gauche-droite comme c'est le cas pour LZ78 et BIOCOMPRESS-2, mais plutôt de la plus longue à la plus courte. De cette façon, les longues répétitions sont exploitées car elles sont envisagées avant les plus courtes. Dans le cas de l'exemple de la figure 3.2, la répétition du facteur B est considérée avant celle de A. Cet algorithme a été amélioré au cours des années, il porte maintenant le nom de CFACT et possède un équivalent CPAL qui exploite les palindromes génétiques par ordre de longueur décroissante [Riv96, RDDD96, RDDD97].

Étant donné la séquence $S \in \mathcal{N}^n$, l'algorithme CFACT procède en trois phases: le **pré-traitement**, l'**analyse** et le **codage**.

Pré-traitement : L'arbre des suffixes de la séquence S est construit et les répétitions sont classées par ordre de longueurs décroissantes. En fait, une structure de liste ordonnée de nœuds internes de l'arbre des suffixes est construite; chaque nœud interne correspond à un facteur répété (voir partie I, section 1.3.3).

Analyse : Les répétitions sont parcourues de la plus longue à la plus courte grâce à la structure construite par la phase de pré-traitement. Pour chaque répétition, le test permettant de savoir si la répétition doit être exploitée, est effectué. Si elle doit être exploitée, alors chaque occurrence de la répétition (excepté la première qui sert de référence) est placée dans une *Liste de Zones à Coder* (LZC) en vue de la phase finale de codage. Pour chaque occurrence, les informations placées dans la liste sont celles qui identifient totalement la répétition:

- La position de l'occurrence: i.
- La position de la première occurrence: pos.
- La longueur du facteur répété: lg.

Le test permettant de savoir si les occurrences de la répétition doivent être introduites dans LZC tient compte des trois points suivants:

1. Une occurrence n'est placée dans LZC que si elle ne chevauche aucun facteur déjà présent dans LZC (et donc de longueur supérieure au sens large).
2. Le nombre d'occurrences doit être suffisant.
3. La longueur du facteur répété doit être suffisante.

Ces deux dernières conditions prennent en considération les longueurs des codages auto-délimités des entiers qui identifient l'occurrence. En pratique, CFACT utilise *Fibo* pour auto-délimiter les entiers.

Codage : Le code de la séquence comprimée est divisé en deux parties :

1. Le codage de tous les triplets (i, pos, lg) de LZC, préfixé du codage auto-délimité du nombre de triplets. Tous les entiers des triplets sont codés de manière auto-délimitée à l'aide du code *Fibo*.

   Ce sont tous les pointeurs qui référencent les premières occurrences des répétitions exploitées.

2. La séquence restante : tous les facteurs de la séquence initiale qui n'ont pas été codés dans la première partie sont concaténés et codés à l'aide de NUC^*.

   Il s'agit de toutes les zones ne correspondant à aucune répétition exploitée ainsi que les premières occurrences des répétitions exploitées.

   Remarque 3.2 *Il est envisageable de comprimer la suite de nucléotides de cette deuxième partie de codage à l'aide d'un autre algorithme de compression, dédié aux séquences génétiques ou non, pour améliorer le gain de compression comme cela a été fait dans* BIOCOMPRESS-2. *On peut par exemple utiliser un codage arithmétique ou le compresseur* CPAL *qui exploite les palindromes génétiques (voir ci-dessous).*

Pour accélérer la phase d'analyse, toutes les répétitions ne sont pas considérées. On sait en effet qu'une séquence d'ADN contient énormément de courtes répétitions qui ne sont pas de longueur suffisante pour apporter un gain. Dès lors, CFACT considère uniquement les répétitions de longueurs supérieures ou égales à un paramètre MLR (*Minimal Length of Repeats*) . La structure ordonnée, construite dans la phase de pré-traitement, est ainsi de plus petite taille, ce qui engendre une grande économie mémoire et améliore la complexité en temps de la phase d'analyse. De plus, la longueur lg d'une répétition peut maintenant être codée de manière relative : on code $(lg - MLR)$ plutôt que lg, ce qui coûte moins de bits. Bien entendu, il convient de choisir une valeur adéquate pour MLR. En pratique on sait que, dans une séquence aléatoire de longueur n, la longueur moyenne de la plus longue répétition est $\log n$. D'après la théorie de la complexité de Kolmogorov, les répétitions dues au hasard ne permettent pas de comprimer, $\log n$ constitue donc une bonne valeur à donner à MLR.

Un des points importants de CFACT est la garantie de compression qu'exprime le théorème 3.2.

Théorème 3.2 *Si le plus long facteur répété dans la séquence $S \in \mathcal{N}^n$ est de longueur lg et apparaît au moins deux fois aux positions i, j telles que $i + lg < j$, alors la condition suivante est suffisante pour produire un gain de compression positif :*

$$2lg - |Fibo(lg)| > 3|Fibo(n)| + 1$$

La preuve est omise. Elle se trouve dans [Riv96].

La complexité en temps de CFACT est en $O(n^2)$ et la complexité en espace est de $O(n \log n)$.

L'algorithme CPAL est dédié à la compression des séquences par exploitation des palindromes génétiques [Riv96, RDDD97]. Le principe est similaire à celui de CFACT, les palindromes génétiques sont trouvés à l'aide de l'arbre des suffixes en considérant la séquence complémentaire et renversée de S.

Dans le cadre de ses travaux de doctorat, E. Rivals a effectué de nombreuses expérimentations de CFACT sur des séquences de divers types et de diverses espèces [Riv96]. Il s'avère

que CFACT est capable de comprimer fortement les séquences qui contiennent des répétitions significativement longues. Les taux de compressions obtenus sont supérieurs, dans la majorité des cas, aux taux obtenus à l'aide des quatre compresseurs classiques (seul le codage arithmétique semble pouvoir comprimer de quelques pourcents certaines séquences que CFACT ne parvient pas à comprimer).

Malgré cela, il faut remarquer que la majorité des séquences restent difficilement compressibles par CFACT: si le taux maximal de compression est de 21% pour certaines séquences d'Arabidopsis thaliana, le taux moyen dépasse rarement quelques pourcents. Chez Escherichia coli, Bacilus subtilis, Saccharomyces cerevisiae et pour les mitochondries, le taux moyen reste inférieur à 1% mais il est supérieur chez toutes les autres espèces et dépasse même 2% chez l'homme, Arabidopsis thaliana et le crapaud. CFACT semble donc posséder un pouvoir discriminant sur les séquences de diverses espèces.

3.6 Classification de séquences: CBI (D. Loewenstern et al, 1995)

Les méthodes d'apprentissage telles que les réseaux de neurones ou les arbres de décision sont des approches courantes pour développer des outils de *classification de séquences d'ADN*. Dans le rapport technique [LHYN95], D. Loewenster, H. Hirsh, P. Yianilos et M. Noordewier s'intéressent au problème de classification de séquences d'ADN à l'aide de la compression. Dans ce papier, une nouvelle classe d'apprentissage appelée CBI (*Compression Based Induction*) est développée.

Commençons par présenter la problématique de la classification de séquences. On dispose de plusieurs classes de séquences et d'une séquence test. Il faut rattacher la séquence test à une des classes, c'est-à-dire conjecturer que la séquence test est du même type que les séquences d'une des classes.

Exemple 3.9 *Soient les deux classes C_I et C_E :*

- *C_I est la classe des séquences d'introns.*
- *C_E est la classe des séquences d'exons.*

Soit $S \in \mathcal{N}^+$ une séquence d'ADN. Il faut rechercher si elle doit être rattachée à C_I ou à C_E. Si on la rattache à C_I, on conjecture que c'est une séquence intronique, si on la rattache à C_E, on conjecture que c'est une séquence exonique.

L'idée promue dans [LHYN95] est de comparer, par compression, la similarité de la séquence test avec des échantillons extraits de chacune des classes. La séquence test est alors rattachée à la classe correspondant à l'échantillon pour lequel la similarité est la plus forte.

Soit $\mathcal{E} = \{S1, S2, \ldots SN\}$ un *échantillon* d'une classe de séquences avec $S1, S2, \ldots SN \in \mathcal{N}^+$. Soit $T \in \mathcal{N}^+$ la séquence test. On dispose d'une méthode de compression de séquences d'ADN, A. Notons $A(S) \in \mathcal{B}^+$ le résultat de la compression d'une séquence S par A.

Pour évaluer la similarité de T avec l'échantillon $\mathcal{E}$, on comprime, à l'aide de A, la concaténation de toutes les séquences de $\mathcal{E}$ et de T. On s'intéresse uniquement à la longueur de la séquence comprimée: $l = |A(S1\,S2 \ldots SN\,T)|$. De la même manière, on comprime la concaténation de toutes les séquences de $\mathcal{E}$ (sans la séquence T) à l'aide de A. On s'intéresse à la longueur de la séquence comprimée: $l' = |A(S1\,S2 \ldots SN)|$. La différence $l - l'$ évalue la

similarité entre T et $\mathcal{E}$. Plus petite est cette différence, meilleure est la similarité. En quelque sorte, $l - l'$ évalue la quantité d'informations que la séquence T véhicule et qui ne sont pas présentes dans $\mathcal{E}$. Plus petite est cette quantité d'information, plus proche est T de $\mathcal{E}$.

Pour la séquence test T, cette procédure est appliquée à un échantillon de chacune des classes, T est alors rattachée à la classe pour laquelle la compression a été la meilleure.

Les critères permettant de classifier les séquences influencent directement le choix du compresseur A à utiliser. Dans [LHYN95], plusieurs compresseurs sont considérés : ZDIFF (implémentation de LZ78), BASIC-CBI (compresseur similaire à celui décrit dans [MJ93]), TRYEACH (effectue la compression avec chacune des séquences de l'échantillon et prend ensuite la moyenne, plutôt qu'une compression globale avec tout l'échantillon),...

Cette méthode est appliquée à deux problèmes de classification :

1. Distinction entre les séquences promotrices de transcription et les séquences non promotrices de transcription chez Escherichia coli.

2. Reconnaissance des sites d'épissage chez les eucaryotes.

Les résultats expérimentaux semblent prometteurs, la majorité des séquences testées ont été classées correctement (voir [LHYN95] pour un tableau détaillé des résultats). Il semble y avoir un besoin de méthode simple et générale d'apprentissage telle que CBI. Toutefois, les raisons pour lesquelles ces deux types de problèmes de classification ont pu être résolus grâce à la méthode CBI ne sont pas totalement connues, les auteurs continuent leur recherche dans ce sens.

Chapitre 4

Localisation de répétitions en tandem approximatives

Grâce à un problème précis de biologie, nous montrons dans ce chapitre que la méthode d'optimisation de courbes par liftings peut être avantageusement utilisée pour localiser des régularités dans des séquences. Les régularités qui nous intéressent ici sont les *Répétitions en Tandem Approximatives (RTA)* . Une répétition est dite "en tandem" si plusieurs copies du même motif se suivent côte à côte. Nous proposons un algorithme exact qui produit, en un temps $O(np + n \log n)$, la meilleure façon de décomposer la séquence en zones régulières et zones non régulières de longueurs quelconques. L'entier n est la longueur de la séquence et p est la longueur du motif. On évalue également l'importance des régularités à l'aide du gain de compression.

Précédemment, les méthodes nous obligeaient souvent à décomposer une séquence en fenêtres de tailles fixées ou à déterminer des seuils pour arriver à savoir si la fenêtre était régulière ou non. Notre méthode n'est tributaire d'aucun seuil ni taille de fenêtres. Les zones régulières sont de longueurs quelconques.

Avant de présenter notre méthode, nous donnons les bases biologiques du problème de localisation de répétitions en tandem approximatives. Les principales motivations sont leurs liens avec des maladies génétiques humaines. Nous ne nous intéressons qu'au cas "périodique" des répétitions en tandem approximatives : le nombre de copies du motif est quelconque.

Ensuite, nous présentons la méthode de *programmation dynamique cyclique (Wraparound Dynamic Programming (WDP))*, introduite dans [FLSS92], qui aligne une séquence avec un mot périodique. La méthode heuristique de localisation de Benson-Waterman [BW94] est alors présentée, elle se base sur la WDP.

Nous introduisons ensuite la première méthode de détection de RTA par compression. Elle a été développée au sein de notre équipe par Eric Rivals [Riv96]. C'est une méthode heuristique qui procède par découpage de la séquence en fenêtres contiguës et qui recherche, dans chacune des fenêtres les RTA d'un motif inconnu.

Enfin, nous développons notre méthode qui possède trois phases bien distinctes : la première recherche le meilleur alignement avec une répétition périodique du motif. Nous développons une méthode similaire à la méthode WDP. Nous la modélisons à l'aide du formalisme des *automates finis* plus agréables et plus faciles à modifier que les équations classiques de récurrence de la programmation dynamique. La deuxième étape procède au codage de l'alignement et la troisième étape optimise le codage à l'aide de la méthode des liftings.

4.1 Bases biologiques et motivations

Dans cette section, nous présentons la problématique biologique des répétitions en tandem approximatives. Le fait qu'elles soient liées à des maladies génétiques chez l'homme leur donne une place de choix dans les recherches effectuées ces dernières années en génétique moléculaire. Les principales sources de documentation qui ont inspiré cette section sont les ouvrages [Riv96, WS93, Ben97].

La *duplication en tandem* est un processus de mutation dans lequel un fragment d'ADN est dupliqué pour produire une ou plusieurs nouvelles copies qui se suivent de façon continue [Ben97].

Exemple 4.1

$$acttacgACTGGACTTTTacatgagcatgggtc$$
$$\downarrow$$
$$acttacgACTGGACTTTTACTGGACTTTTACTGGACTTTTACTGGACTTTTacatgagcatgggtc$$

Avec le temps, une telle répétition est sujette à d'autres mutations, on parle alors de *répétition en tandem approximative (RTA)* : les copies du motif de base de la répétition sont imparfaites. On trouve rarement des répétitions en tandem parfaites, elles sont quasiment toujours approximatives.

Exemple 4.2 *La séquence*

$$\underline{CATGG}\,\underline{CATaG}\,\underline{CTGG}\,\underline{CAtTGG}\,\underline{ATGa}\,\underline{CATGG}\,\underline{CATGG}\,\underline{CAaTGG}\,\underline{CATcG}$$

est une RTA de 9 copies du motif de base CATGG.

Les répétitions en tandem sont très fréquentes, la quantité est évaluée à plus de 5% chez l'homme (c'est-à-dire plus que de gènes !) et jusqu'à 20% chez *Bovis domesticus* (le bœuf) [Riv96].

Elles font partie d'une classe plus large de fragments d'ADN qui contiennent des éléments de symétrie dans leur séquence : le *dosDNA (defined ordered sequence DNA)*. Ces éléments de symétrie donnent des conformations spatiales très éloignées de l'habituelle structure double hélicoïdale découverte par Watson et Crick. Le dosDNA est très étudié, la plupart de ces conformations alternatives de l'ADN sont importantes en biologie. Dans certains cas, le dosDNA est impliqué dans des événements mutationnels qui peuvent mener au cancer ou à des maladies génétiques chez l'homme. L'ouvrage [WS93] effectue un large parcours de la problématique biologique du dosDNA et des conformations spatiales possibles.

Le cas particulier des RTA est important. L'implication des RTA dans au moins huit maladies génétiques humaines a été mise en évidence récemment [Ben97] [1]. Dans ces maladies, des répétitions en tandem de tri-nucléotides sont anormalement amplifiées, c'est-à-dire que le nombre de copies du tri-nucléotide (le motif de base de la répétition) est fortement augmenté. Plus grand est le nombre de copies du motif, plus grave est la maladie. De plus, la gravité de la maladie augmente de génération en génération [CPF+92, RS92, SW92]. Dans certains cas, le

1. Parmi elles, citons le retard mental "fragile X" [VPS+91], la maladie d'Huntington [Hun93], l'atrophie musculaire de la maladie de Kennedy [LWL+91], la distrophie myotonique [FPF+92] et l'ataxie de Friedreich [CMM+96].

nombre de répétitions est multiplié par un facteur 10 ou plus lorsque la RTA est transmise du parent vers son enfant [SW92]. Ce lien étroit entre les maladies génétiques et les RTA motive de nombreuses recherches à leur égard. Notons également que les RTA interviennent dans la machinerie d'expression des gènes en protéines [LWGE93, HSHG84, PLRN87, RHYS93, YWvR91] et dans les mécanismes d'évolution [HSSP88].

Nous distinguons deux catégories de répétitions en tandem :

Deux copies en tandem d'un même long fragment : Dans les séquences d'ADN, il arrive que deux longs fragments, situés côte à côte, soient très ressemblants : Soit $S = ABB'C$ une telle séquence avec $A, B, B', C \in \mathcal{N}^*, |B| \approx |B'|$ et le fragment B très "ressemblant" au fragment B' (au sens de la distance de Levenshtein par exemple, voir section 2.5.2, remarque 2.2). Les fragments B et B' ne sont pas eux-mêmes des RTA d'un motif plus court.

C'est notamment le cas des gènes dupliqués en tandem : la création d'une nouvelle protéine se fait par duplication d'un gène existant et évolutions indépendantes des deux copies du gènes. Par exemple, chez l'homme et la souris, les gènes de l'alpha-fœtoprotéine et de l'albumine sont suffisamment proches pour montrer qu'ils proviennent de la duplication d'un même gène ancestral [Riv96]. Nous ne nous intéressons pas à ce cas que nous préférons appeler un *mot carré approximatif*.

L'ADN satellite : Il s'agit de classes de fragments d'ADN qui sont des RTA de motifs très diversifiés. Ici, le terme RTA prend un sens "périodique" : le nombre de copies du motif de base est quelconque, il peut varier de quelques unités à plusieurs milliers de copies. Le nombre de nucléotides du motif est également très variable : de quelques bases à plusieurs milliers de bases.

Ce sont ces classes de RTA qui donnent des conformations spatiales inhabituelles à l'ADN : ils provoquent des empilements cycliques sur un brin d'ADN qui imposent à la molécule des structures alternatives (nommées ADN-Z, triplex,...) [Riv96, WS93].

Les fragments d'ADN satellite correspondent souvent à des séquences de polymorphisme : elles peuvent varier fortement en longueur selon les individus d'une même espèce. De telles régions de polymorphisme sont utiles pour localiser les gènes dans les régions spécifiques d'un chromosome et également pour déterminer la probabilité de concordance d'empreintes génétiques [BW94, WM89, EHJ+92].

Il faut distinguer deux classes d'ADN satellites :

1. Les *micro-satellites* sont des RTA de motifs très courts (moins de 5 bases). Le nombre de copies du motif varie de quelques unités à plusieurs centaines. Ces micro-satellites semblent être le siège des maladies génétiques mentionnées plus haut dans lesquelles l'amplification des RTA de triplets joue un rôle particulier.

   Exemple 4.3 *La séquence d'ADN suivante est un fragment du chromosome 4 de la levure (position de début :* 149101, *longueur :* 92*). C'est une RTA de* 23 *copies du motif* TAAA.

   TAA TAAAc TAAA TAAAa TAAA TAAA TAAA TAAA TAAA TAAA TAAA TAAA TAAA TAAA TAAA TAAA TAA TAtA cgA TcA TAAAa TAAA TAAtA

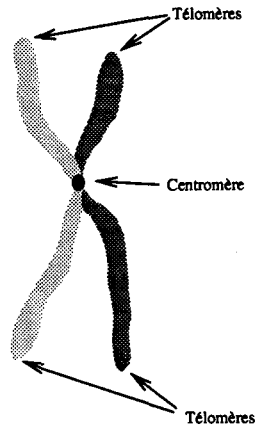


FIG. 4.1 - *Les deux chromatides sœurs d'un chromosome avant la division cellulaire*

2. Les *mini-satellites* sont des RTA de motifs plus longs dont le nombre peut atteindre plusieurs milliers. Les *centromères* et les *télomères* des chromosomes contiennent de nombreux mini-satellites (le centromère d'un chromosome est la partie compacte en son centre qui relie les deux chromatides sœurs, il contrôle le mouvement du chromosome pendant la division cellulaire; les télomères sont les extrémités naturelles des chromosomes, voir figure 4.1).

   Exemple 4.4 ([Riv96]) *Chez la drosophile, le motif AAGAGAG est répété 16000 fois dans le centromère du chromosome* 3.

   Exemple 4.5 ([Riv96]) *Le satellite-α humain de 170 bases est dupliqué par paquets de* 12 *sur* 2.1 *kb. Sa fonction est inconnue.*

   L'approche de localisation de RTA, à l'aide de l'optimisation par liftings, que nous proposons dans ce chapitre est aussi bien adaptée aux micro-satellites qu'aux mini-satellites.

Dans le cas des maladies génétiques mentionnées plus haut, les mécanismes biologiques qui provoquent l'amplification des RTA de tri-nucléotides ne sont pas connus. Dans [WS93], deux mécanismes hypothétiques sont proposés, le premier est basé sur la *recombinaison homologue inégale* entre chromosomes et le second sur le *dysfonctionnement de la réplication* de l'ADN. Décrivons-les brièvement.

Recombinaison homologue inégale : La recombinaison homologue, également appelée crossing-over, est le processus qui permet des échanges de fragments d'ADN entre des chromosomes homologues. Elle intervient dans la phase de méiose pour créer de nouvelles associations de gènes sur les chromosomes.

Le processus consiste en l'échange, entre les deux chromosomes, de deux fragments d'ADN très semblables (homologues) contenant deux versions d'un même gène. La figure 4.2 représente une recombinaison homologue entre les chromosomes A et B. Habituellement, les longueurs des deux fragments échangés sont identiques. Avant l'échange, ils sont alignés l'un au-dessus de l'autre de telle sorte que la ressemblance entre les fragments soit grande. Si les fragments à échanger sont des répétitions en tandem d'un même

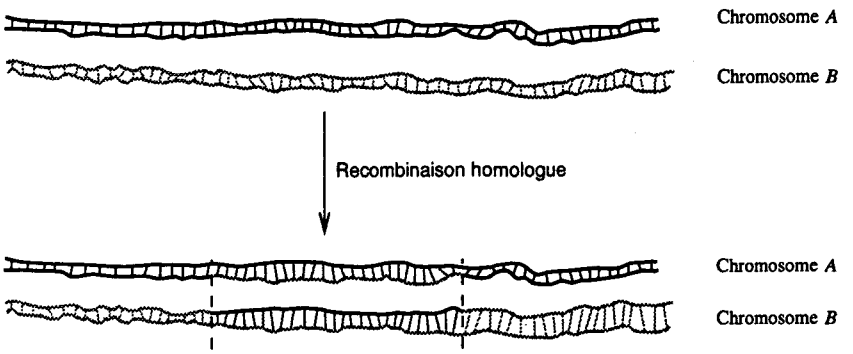


FIG. 4.2 - *Recombinaison homologue*

motif de base, alors l'alignement peut être imparfait: les deux morceaux homologues peuvent être décalés l'un par rapport à l'autre. La figure 4.3 représente un aligne-

```
...aggatcactactactACTACTACTACTACTACTtgaaaccaccatcggg...
...tgacagtctgacaggACTACTACTACTACTACTactactactaggagca...
```

FIG. 4.3 - *Alignement de deux répétitions en tandem du motif ACT avec décalage*

ment, avec décalage, de deux répétitions en tandem de ACT. Les caractères majuscules montrent les bases mises en correspondance.

Dans ce cas, il se peut que la recombinaison soit inégale: les longueurs des deux fragments échangés peuvent ne pas être identiques. Supposons, dans le cas de la figure 4.3, que la recombinaison s'opère sur les deux fragments soulignés. Dès lors, **il résulte une amplification de la répétition en tandem dans un des deux chromosomes** et une réduction de la répétition dans l'autre chromosome (les deux répétitions en tandem sont représentées en caractères majuscules sur la figure 4.4).

```
...aggatcACTACTACTACTACTACTACTACTACTACTACTtgaaaccaccatcggg...
          ...tgacagtctgacaggACTACTACTACTACTACTaggagca...
```

FIG. 4.4 - *Chromosomes de la figure 4.3 après recombinaison inégale*

Il a été montré, expérimentalement, que les répétitions en tandem sont très souvent sujettes à la recombinaison. L'hypothèse d'amplification des répétitions en tandem par recombinaison inégale semble donc très plausible [Riv96].

Dysfonctionnement de la réplication : Le processus de réplication de l'ADN est réalisé à l'aide d'un enzyme particulier appelé *ADN polymérase*. Il se fixe sur un brin d'ADN désolidarisé de l'autre (voir figure 2.3, page 131) et génère en se déplaçant, nucléotide par nucléotide, un nouveau brin complémentaire.

Les hypothèses d'amplification des répétitions en tandem par dysfonctionnement de la réplication supposent toutes que l'ADN polymérase soit entravée dans son déplacement. Pour illustrer les propos, imaginons qu'une molécule soit venue se fixer à un endroit

particulier de la molécule d'ADN mère, empêchant de la sorte la séparation des deux brins et la progression de l'ADN polymérase (voir figure 4.5).

Plusieurs phénomènes peuvent en découler, parmi ceux-ci citons :

- La polymérase "enjambe" la molécule qui entrave la réplication et continue à synthétiser l'ADN. Il en découle une absence de quelques nucléotides dans la nouvelle molécule synthétisée.
- La polymérase glisse en arrière sur le brin modèle avant de synthétiser à nouveau le même motif. L'itération de ce processus plusieurs fois de suite sur une répétition en tandem provoque **l'amplification de la répétition en tandem sur le nouveau brin synthétisé** qui se replie au fur et à mesure en une structure d'épingle à cheveux (voir figure 4.5). On parle de *bégaiement* de l'ADN polymérase. Ce processus n'est possible que si la répétition en tandem est un palindrome génétique approximatif : les deux brins de l'épingle à cheveux doivent pouvoir s'apparier. Cette contrainte est forte car elle impose une structure particulière au motif de la répétition en tandem : il doit pouvoir s'apparier avec sa séquence renversée, au moins partiellement. Par exemple, `TA` s'apparie avec son renversé `AT` : $\texttt{AT} = Comp(Rv(\texttt{AT}))$. Dès lors, le processus de bégaiement de l'ADN polymérase peut être à l'origine d'une amplification de RTA du motif `TA`.

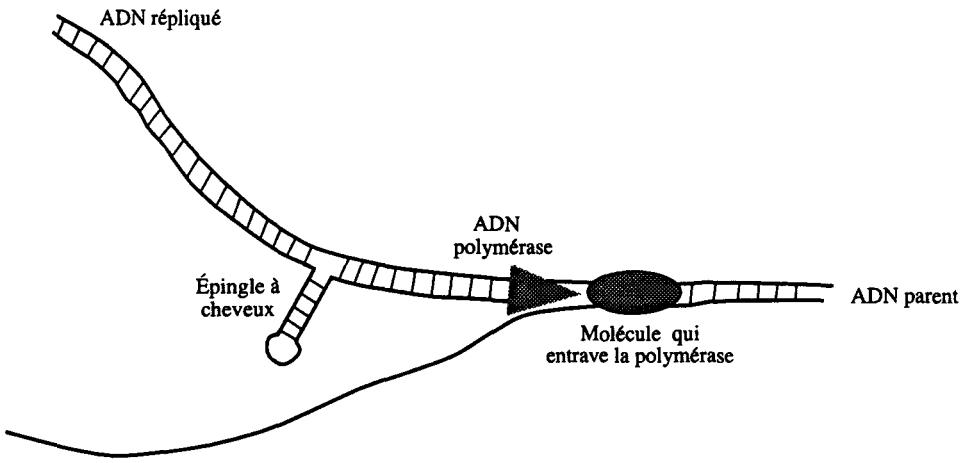


FIG. 4.5 - *Bégaiement l'ADN polymérase*

- La polymérase saute sur l'autre brin de la molécule modèle et synthétise un fragment qui est le complémentaire renversé du motif qui vient d'être synthétisé. S'il s'agissait d'une répétition en tandem, il y a synthèse de la répétition en tandem complémentaire et renversée. Ce mécanisme peut se produire plusieurs fois, ce qui **donne naissance à une plus longue répétition en tandem**.

Pour plus de détails concernant ces mécanismes hypothétiques d'amplification des RTA, se référer à [Riv96, WS93].

Les maladies génétiques associées à l'amplification de répétitions en tandem de triplets ont été découvertes chez l'homme mais aucun argument ne pousse à croire que l'amplification des RTA est confinée à l'être humain. Il convient donc d'étudier les RTA chez d'autres organismes que chez l'homme. La levure est l'organisme d'étude par excellence : c'est le premier

organisme eucaryote dont tous les chromosomes ont été complètement séquencés. Il est donc enfin possible de réaliser une étude exhaustive dans l'entièreté de l'ADN d'un organisme eucaryote. De plus, c'est un organisme très étudié en génétique. Dans ce chapitre, nous nous intéressons de près à la localisation des RTA dans les chromosomes de levure.

De la même façon, aucune raison n'exige que le phénomène d'amplification soit limité aux RTA dont le motif de base est un triplet. La méthode de localisation que nous proposons dans ce chapitre n'est liée à aucune longueur de motif déterminée, encore moins à une classe de motif déterminée.

4.2 Programmation dynamique cyclique

La programmation dynamique cyclique (*WDP* pour *Wraparound Dynamic Programming*), introduite par Fischetti, Landau, Schmidt et Sellers [FLSS92], aligne la répétition en tandem d'un motif donné M avec une séquence S. La méthode WDP recherche le nombre de copies de la répétition en tandem de telle sorte que l'alignement soit optimal. Une fonction de coût ω est donnée (par exemple les coûts utilisés pour rechercher la distance de Levenshtein), la méthode WDP recherche en fait le meilleur alignement entre S et un préfixe de longueur finie de M^∞. Elle considère donc que **toute** la séquence S est une RTA du motif M.

Exemple 4.6 *Soit* $M =$ `ATC` *le motif et*

$$S = \texttt{ATATCATCATCATGCATCATCAGGTTAGGGCTGGCAGAGGTATGGCCATCATCATCA}$$

la séquence. En utilisant la matrice de coûts de la distance de Levenshtein, la méthode WDP fournit l'alignement optimal :

$$\begin{array}{lcl} M^\infty & = & \texttt{ATcATCATCATCAT-CATCATCA--TCA----T--CATC---AT--C-ATCATCATCA...} \\ S & = & \texttt{AT-ATCATCATCATgCATCATCAggTtAgggcTggCAgaggtATggCcATCATCATCA} \end{array}$$

La méthode WDP est une adaptation de la méthode classique d'alignement par programmation dynamique. Elle trouve le meilleur alignement en temps $\theta(n.p)$ où n est la longueur de la séquence et p la longueur du motif. Tout comme la méthode classique d'alignement, il existe une version "locale" de la WDP qui trouve le facteur de la séquence qui s'aligne au mieux avec une répétition en tandem du motif.

Nous décrivons, dans la section 4.5.2, un algorithme d'alignement global similaire à la méthode WDP. La présentation que nous en faisons n'est pas formalisée à l'aide de formules de récurrence comme c'est le cas pour la majorité des méthodes de programmation dynamique, mais plutôt en termes d'*automates finis* (de *transducteurs*).

Remarque 4.1 *La méthode décrite dans [FLSS92] ne se base pas sur un alignement par "distance" comme nous avons eu l'habitude de le faire jusqu'à présent (toute mutation possède un coût positif, la coïncidence deux bases possède un coût nul) mais plutôt sur un alignement par "similarité" (toute mutation a un poids négatif, toute coïncidence a un poids positif).*

Un alignement local avec une répétition en tandem n'est pas possible à l'aide de notre approche par "distance" car le préfixe de M^∞ qui offre le meilleur alignement avec un facteur de la séquence S est toujours le mot vide ϵ : son coût est nul.

Remarque 4.2 *De la même façon que pour l'alignement classique, la programmation dynamique cyclique locale est uniquement capable de trouver le facteur de la séquence qui s'aligne au mieux avec une répétition en tandem du motif (voir remarque 2.4, page 147). En quelque sorte, étant donné le motif de base M, la WDP locale permet de trouver uniquement la RTA du motif M la plus significative dans la séquence S. Elle ne permet pas de rechercher toutes les RTA du motif. Pour trouver plusieurs RTA il convient donc de définir un seuil de similarité et d'accepter toutes les RTA qui vérifient le seuil. Malheureusement, la définition du seuil est très délicate, d'une part parce qu'elle introduit de l'arbitraire dans la sélection des RTA et d'autre part parce que le score de l'alignement est dépendant du nombre de copies de la RTA. Il faut donc également tenir compte du nombre de copies du motif pour sélectionner une RTA.*

Dans les sections 4.4 et 4.5, nous présentons respectivement une méthode heuristique et une méthode exacte de localisation de RTA basées sur l'utilisation de la compression. Une zone est qualifiée de RTA uniquement si elle peut être codée de manière concise en utilisant la propriété de répétition en tandem. Cela évite la définition arbitraire d'un seuil.

4.3 Méthode heuristique de G. Benson et M.S. Waterman

La méthode de localisation de RTA de Benson et Waterman [BW94] était la seule méthode existante avant nos approches par compression. C'est une méthode heuristique rapide qui considère que dans chaque copie du motif d'une RTA, un certain nombre de caractères contigus du motif sont conservés.

Sont donnés à l'algorithme:

1. Une séquence d'ADN S dans laquelle doivent être localisées les RTA.
2. La *longueur p du motif* de base des RTA recherchées.
3. Le *paramètre de détection de motif*, c'est-à-dire la longueur du facteur conservé dans les copies approximatives du motif: c.
4. La *matrice des similarités* $\mathcal{M}_{sim}$.
5. Un *seuil de similarité* d'une RTA trouvée par rapport à la *Répétition en Tandem Exacte (RTE)* : t.

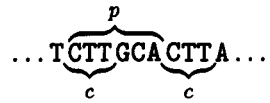
La méthode doit donc également rechercher les motifs de base pour les RTA, seule leur longueur p est donnée.

La première étape consiste en la recherche, dans S, de tous les motifs hypothétiques des RTA. Pour chacun des motifs hypothétiques u, la deuxième étape détermine les limites de la RTA de u en utilisant la WDP à gauche et à droite de l'endroit où u a été détecté à l'étape 1. Les limites sont déterminées en observant le fléchissement du score de l'alignement. Si le score d'alignement dépasse le seuil t alors l'étape 3 est exécutée, autrement le motif u est abandonné. Grâce à l'alignement de la deuxième étape, la troisième étape construit un motif consensus u' en remplaçant chaque base de u par la base majoritaire à cette position dans l'alignement. L'alignement de S avec la répétition en tandem de u' est alors calculé par WDP, ce qui fournit, la zone et son alignement.

Détaillons les trois étapes de l'algorithme.

Étape 1 : Dans une région de RTA, bien que des insertions, délétions et substitutions aient modifié les copies du motif, on s'attend à trouver de petites régions contiguës conservées.

Le paramètre de détection de motif, c, est la longueur de ces régions contiguës. Si p est la longueur du motif alors on s'attend à trouver deux occurrences d'un mot de longueur c à p symboles d'écart :



La première étape parcourt la séquence S et recherche toutes les répétitions de mots de longueur c dont les occurrences sont situées à une distance de p bases. Tous les mots séparant les deux positions d'occurrences sont des motifs hypothétiques de RTA et sont fournies à l'étape 2. Dans le cas de l'exemple ci-dessus, $u =$ `CTTGCA` est un motif hypothétique.

Remarque 4.3 *En réalité, le motif hypothétique fournit à l'étape 2 est la plus petite permutation cyclique (par ordre lexicographique) du motif trouvé. Cela permet de ne pas rechercher les RTA d'un motif hypothétique si les RTA d'une de ses permutations cycliques ont déjà été localisées. Dans l'exemple, la plus petite permutation cyclique de* u *est* `ACTTGC`.

Étape 2 : Soit u un motif hypothétique. Un score de similarité, de la région de S où u a été détecté, avec la répétition en tandem exacte (RTE) de u, est calculé par la méthode WDP utilisant la matrice de similarité $\mathcal{M}_{sim}$. Les limites de la région sont déterminées par le fléchissement du score en alignant de gauche à droite (resp. de droite à gauche) : lorsque le score devient nul, la limite droite (resp. gauche) est déterminée. Le score de la région est donné par le score maximal rencontré durant l'alignement. Si le score dépasse le seuil t alors la troisième étape est exécutée.

Étape 3 : Le motif u n'est peut-être pas le motif dont la RTE ressemble le plus à la région alignée à l'étape 2. Une région peut en effet contenir de nombreux motifs hypothétiques. Pour choisir le meilleur, le motif consensus u' est construit à partir de l'alignement réalisé de la façon suivante :

- $|u'| = |u|$.
- $u'_i = a$, avec $a \in \mathcal{N}$ si a est la base qui a été le plus souvent alignée avec la $i^{\text{ème}}$ base du motif u dans l'alignement de l'étape 2.

L'alignement de la région avec le motif consensus u' est alors calculé par WDP de la même manière que dans l'étape 2. Une RTA a donc été trouvée.

Exemple 4.7 ([BW94]) *Supposons que* $u =$ `ACGTT` *soit le motif hypothétique et que l'alignement fourni à l'étape 2 soit :*

u^∞	=	`ACGtt ACG-Tt aCGTT ACGTt AcGTt A...`
Région de S	:	`ACGaa ACGgTa -CGTT ACGT- AgGTa A`

Alors le motif consensus est $u' =$ `ACGTA`. *La RTE de* u' *possède une similarité plus forte que celle de* u *avec la région de* S *:*

u'^∞	=	`ACGtA ACG-TA aCGTa ACGTa AcGTA A...`
Région de S	:	`ACGaA ACGgTA -CGTt ACGT- AgGTA A`

La méthode de Benson et Waterman est très rapide mais repose entièrement sur un ensemble de paramètres parfois difficiles à choisir. De plus, certaines classes de RTA n'ont pas toujours deux occurrences d'un facteur conservé de longueur c à une distance **exacte** de p bases où p est la longueur du motif (sauf dans le cas trivial $c = 0$). En pratique, pour déterminer la valeur du paramètre c, il faut faire un compromis entre la rapidité d'exécution et la quantité de RTA trouvées. L'expérience montre qu'une valeur de 3 à 8 donne de bons résultats[2] [BW94].

L'article propose une méthode statistique pour fixer le seuil de similarité mais le calcul n'est pas inclus dans le programme [Riv96].

4.4 Méthode heuristique par compression

Nous décrivons ici notre première approche par compression de la localisation de RTA dans des séquences d'ADN. Il s'agit d'une méthode heuristique linéaire en temps. À l'aide de la compression, elle localise, dans une séquence de taille modérée, toutes les RTA d'un même motif. La méthode se limite aux motifs de longueurs $1, 2, 3$. Elle repose sur une définition particulière d'une RTA. La compression permet de déterminer objectivement (sans définir de seuil arbitraire) si une zone est une RTA ou non : si la connaissance du motif présumé de la répétition en tandem permet de coder la zone de manière concise, alors c'est une RTA.

Cette approche a été avant tout développée par Eric Rivals dans le cadre de sa thèse de doctorat [Riv96]. Beaucoup d'exemples de cette section sont issus de ce document. Les deux publications [RDDD95, RDD+97], jointes respectivement dans les annexes B et C, évoquent notre collaboration sur ce sujet.

Nous commençons par présenter la définition précise d'une RTA, nous décrivons alors le schéma de codage permettant de comprimer une séquence contenant des RTA. Nous poursuivons par l'algorithme heuristique de localisation des RTA d'une séquence connaissant le motif de base. Nous terminons en présentant brièvement l'application de la méthode à la recherche de RTA dans des chromosomes de levures.

4.4.1 Définition d'une RTA

L'algorithme utilise une définition particulière des RTA que nous donnons ici. Une zone RTA d'une séquence est un facteur de la séquence *similaire* à une Répétition en Tandem Exacte (RTE) d'un motif.

Précisons tout d'abord, de manière informelle, le problème de localisation que l'on se pose.

Problème 4.1 *Soit $S = s_1 s_2 \dots s_n \in \mathcal{N}^n$ une séquence et $M = m_1 \dots m_p \in \mathcal{N}^p$ un motif, avec $p < n$ et p "petit" (en pratique : $p \in \{1, 2, 3\}$). On désire trouver toutes les RTA significatives du motif M dans la séquence S.*

L'utilisation du terme "significatives" dans l'énoncé du problème évoque en fait la similarité de la zone avec une RTE du motif M.

Définition 4.1 *Le mot $t \in \mathcal{N}^+$ est une* RTE *de M si il existe un entier $l > 1$ tel que $t = M^l$.*

2. Les auteurs ne donnent pas les longueurs p des motifs pour ces "bonnes" valeurs de c.

Une *RTE maximale* d'un motif M dans une séquence est un facteur de la séquence, qui est une RTE de M mais dont aucune extension dans la séquence n'est une RTE de M.

Définition 4.2 *Un facteur $t = s_{i..j}$ est une* RTE maximale *de M dans S si:*

- *t est une RTE de M.*
- *Il n'existe aucun facteur u de S tel que:*
  - *u soit une RTE de M.*
  - *t soit facteur de u et $|u| \neq |t|$.*

Exemple 4.8 *Soit $M = CGT$ le motif. Soit $S =$ actaaggCGTCGTCGTCGTCGTggact la séquence. Le facteur $t = S_{14..22} = CGTCGTCGT$ (le facteur souligné de S) est une RTE de M : $t = M^3$. Ce n'est pas une RTE maximale de M dans S car le facteur $u = S_{8..22}$ (le facteur contenant toutes les lettres majuscules) est une RTE de M et contient t comme facteur. Le mot u est une RTE maximale de M dans S.*

Définissons maintenant ce qu'est une RTA. Intuitivement, un mot t est une RTA du motif M s'il peut être factorisé en une concaténation de mots "similaires" à M. La définition de *similarité au motif M* suppose que le motif soit très court.

Définition 4.3 *Soit $M \in \mathcal{N}^p$ le motif de longueur p. L'*ensemble de tous les mots similaires *à M, noté $Sim(M)$, est donné par la réunion:*

$$Sim(M) = \{M\} \cup sub(M) \cup del(M) \cup ins(u)$$

Où:

- *$sub(M)$ est l'ensemble de tous les mots obtenus par substitution d'une et d'une seule base de M :*

$$sub(M) = \{u \in \mathcal{N}^p | \exists j \in [1,p] : u_j \neq m_j \text{ et } \forall i \neq j, u_i = m_i\}$$

- *$del(M)$ est l'ensemble de tous les mots obtenus par délétion d'une et d'une seule base de M :*

$$del(M) = \{u \in \mathcal{N}^{p-1} | \exists j \in [1,p] : \forall i \in [1, j-1], u_i = m_i \text{ et } \forall i \in [j, p-1], u_i = m_{i-1}\}$$

- *$ins(M)$ est l'ensemble de tous les mots obtenus par insertion d'une base à la fin d'un mot de $\big(\{M\} \cup sub(M) \cup del(M)\big)$:*

$$ins(M) = \{u = v\,a \in \mathcal{N}^+ | v \in \big(\{M\} \cup sub(M) \cup del(M)\big) \text{ et } a \in \mathcal{N}\}$$

Cette définition de similarité exprime que si $u \in Sim(M)$, alors u est obtenu par 0, 1 où 2 mutations à partir de M. C'est un choix arbitraire, mais il est raisonnable de supposer que

le motif est relativement bien conservé dans une RTA. Pour un motif de longueur 3, au moins deux des bases du motif sont conservées.

Exemple 4.9 *Soit $M = ACT$ le motif. Alors l'ensemble des mots similaires à M est:*

$$Sim(ACT) = \left\{ \begin{array}{l} ACT, \\ cCT, gCT, tCT, AaT, AgT, AtT, ACa, ACc, ACg, \\ CT, AT, AC, \\ ACTa, cCTa, gCTa, tCTa, AaTa, AgTa, AtTa, ACaa, ACca, ACga, CTa, ATa, \\ ACTc, cCTc, gCTc, tCTc, AaTc, AgTc, AtTc, ACac, ACcc, ACgc, CTc, ATc, \\ ACTg, cCTg, gCTg, tCTg, AaTg, AgTg, AtTg, ACag, ACcg, ACgg, CTg, ATg, \\ ACTt, cCTt, gCTt, tCTt, AaTt, AgTt, AtTt, ACat, ACct, ACgt, CTt \end{array} \right\}$$

La définition formelle d'une RTA est la suivante.

Définition 4.4 *Un mot $t \in \mathcal{N}^+$ est une RTA de $M \in \mathcal{N}^p$ si et seulement si:*

$$\exists l > 0 : \exists u_1, u_2, \ldots u_l \in Sim(M) : t = u_1 u_2 \ldots u_l$$

Exemple 4.10 *Le mot $t = ACTAGTAGCTCTCTACTACTACTTATCTCTGACACT$ est une RTA de ACT car il se factorise en une concaténation de mots de $Sim(ACT)$:*

$$t = \underline{ACT}\,\underline{AgTa}\,\underline{gCTc}\,\underline{tCT}\,\underline{ACT}\,\underline{ACT}\,\underline{ACTt}\,\underline{ATc}\,\underline{tCTg}\,\underline{AC}\,\underline{ACT}$$

4.4.2 Schéma de codage

Supposons que la séquence S contienne des RTA du motif M. Nous présentons un schéma de codage qui comprime S en exploitant le fait que les RTA sont des "RTE mutées". Nous admettons pour l'instant que les RTA sont non-chevauchantes, nous verrons dans la section suivante comment elles sont localisées.

Soit t une RTA de M. Elle se factorise en mots de $Sim(M)$: $t = u_1 u_2 \ldots u_l$ avec $u_i \in Sim(M)$. Une façon naturelle de coder t est de lister la suite des mutations à appliquer aux copies du motif M pour obtenir les mots u_i. Connaissant le motif M, le **codage d'une RTA** est donc composé des informations suivantes:

- Le *nombre l de copies* du motif.
- Le nombre n_μ de mutations (il n'est pas forcémment égal à l car chaque copie approximative u_i du motif est obtenue par 0, 1 ou 2 mutations à partir de M).
- La *liste de ces mutations.* Le codage de chaque mutation est composée de:
  - L'*identification du motif* u_i sur lequel porte la mutation. C'est l'indice relatif $(i-j)$ du motif par rapport au précédent motif muté u_j.
  - Le *type de mutation* : un mot code de longueur c_μ fixée.

Tous ces renseignements sont suffisants pour reconstruire t lorsque M est connu.

Le **codage de la séquence** S est divisé en trois parties (on suppose pour l'instant que la longueur de la séquence est connue, nous verrons dans la section 4.4.4 que c'est le cas en pratique):

1. Les *informations générales* :
   - La longueur p du motif.

- Le motif M.
- Le *nombre de zones* de RTA.

2. Le codage des zones de RTA. Il contient, pour chacune des zones :
   - La *position du premier nucléotide* de la zone (position relative par rapport à la position de fin de la RTA précédente).
   - Le codage de la RTA, à proprement parler, comme décrit ci-dessus.
3. Le codage de tous les facteurs qui ne contiennent pas de RTA : ils sont concaténés les uns aux autres et codés à l'aide de NUC^*.

Exemple 4.11 *Soit M = TCG le motif, considérons le morceau de séquence suivant :*
...tgctTCG TCc TCG TCc TCG TCa TCa TCG TCG TCG TCG TCG TCG TCG TCG TCG TCG TCG TCG TCG TCG TCa TCG TCG TCa TCG TCa TCa TCa TCG ttgttgttttcgttcttatcttctgtatcttcttct *TCG TCG TCG* tct...*

Il contient les deux RTA soulignées. Le morceau de codage qui le remplace est (les nombres ont été écrits sous forme lisible, les ";" délimitent les mots code et les lignes commençant par % sont des commentaires) :

```
%-----------------------------------------------------------
3; TCG; 2;
% Longueur p du motif; motif; nombre de RTA;

112; 29; 9;
% RTA num 1 : position relative; nombre de copies du motif; nombre de mutations;

(2,12); (2,12); (2,11); (1,11); (14,11); (3,11); (2,11); (1,11); (1,11);
% sa liste de mutations : (numéro relatif du motif, type de mutation),...;

36; 3; 0;
% RTA num 2 : position relative (36 = 235 - 199); nombre de copies du motif; nombre de mutations;
% Pas de liste de mutations

NUC*(...TGCT TTGTTGTTTTCGTTCTTATCTTCTGTATCTTCTTCTTCT...)
% Les facteurs qui n'appartiennent à aucune RTA.
%-----------------------------------------------------------
```

Si la longueur de la séquence est connue et relativement petite, tous les entiers du schéma de codage sont codés sur un nombre fixé de bits.

Il est alors aisé de calculer le gain en compression (le nombre de bits économisés) par un facteur t codé comme RTA :

$$GainZone(t) = 2|t| - c_\mu n_\mu - C$$

Où :

- $2|t|$ représente le nombre de bits gagnés en ne codant pas t à l'aide de NUC^*.
- c_μ est le nombre fixé de bits du codage de chaque mutation : lié à la longueur du motif.

- n_μ est le nombre de mutations.

- C est le coût fixe du codage d'une zone (codage de la position relative de la RTA, du nombre de copies du motif et du nombre de mutations) : lié à la longueur de la séquence et à la longueur du motif.

Exemple 4.12 *Dans le cas de séquences de longeur* 500 *et de motifs de longueur* 3*, on calcule (nous n'incluons pas le détail) :*

$$GainZone(t) = 2|t| - 7n_\mu - 22$$

Moins la factorisation de la RTA possède de mutations, meilleur est le gain de compression apporté par la zone. Si $GainZone(t) \geq 0$, la théorie de la complexité de Kolmogorov nous pousse à conclure que t est une **réelle zone de RTA** : la régularité de t nous permet de réduire la longueur de son codage (ou du moins de ne pas la rallonger). Le test de significativité algorithmique [MJ93], présenté à la section 3.3.1 du chapitre précédent, peut être utilisé pour connaître la probabilité que cette conclusion soit erronée, c'est-à-dire que cette zone RTA soit due au hasard. Dans le cas où $GainZone(t) < 0$, la zone n'est pas qualifiée de zone de RTA.

4.4.3 Algorithme de localisation

Nous décrivons l'algorithme RECH_RTA qui recherche les zones de RTA d'un motif M donné dans une séquence S. Il est applicable à tous les motifs M de longueur $p \in \{1, 2, 3\}$ mais pour simplifier la présentation, nous fixons p à 3 (les deux autres cas sont en fait des cas plus simples à développer). Il s'agit d'un algorithme glouton qui s'exécute en un temps proportionnel à la longueur n de la séquence S. Il ne recherche que les zones de RTA qui ressemblent significativement à une RTE du motif dans le sens où elles sont compressibles. Nous nous permettons donc de restreindre la définition des RTA. Des heuristiques sont appliquées pour localiser et factoriser les zones de RTA dans le but de rendre le gain en compression le plus élevé possible. Par définition d'un algorithme glouton, les choix effectués ne sont jamais remis en question.

L'algorithme RECH_RTA fournit en sortie un ensemble de facteurs $s_{i..j}$ de S qui possèdent les trois propriétés suivantes :

1. $s_{i..j}$ est une RTA de M selon la définition 4.4.

2. $GainZone(s_{i..j}) \geq 0$.

3. Le facteur $s_{i..j}$ ne peut être ni rallongé, ni raccourci, sans provoquer une baisse du taux de compression. C'est-à-dire :

   - Il n'existe aucun facteur $s_{g..h}$ qui contienne $s_{i..j}$ comme facteur, qui soit une RTA de M et tel que $GainZone(s_{g..h}) > GainZone(s_{i..j})$.

   - Il n'existe aucun facteur $s_{g'..h'}$ inclus dans $s_{i..j}$, qui soit une RTA de M et tel que $GainZone(s_{g'..h'}) > GainZone(s_{i..j})$.

Cette dernière propriété exprime la maximalité, au niveau gain produit par la compression de la zone.

Exemple 4.13 *Soit $M = GTA$ le motif, considérons le morceau de séquence suivant :*

$$\underbrace{\ldots acccGTg\overbrace{GTA\underbrace{GTA}GTA}^{s_{g'..h'}}GTA\,cTA\,gttac\ldots}_{s_{a..b}}$$
$$s_{i..j}$$

On a :

$$\begin{array}{lcrcr} GainZone(s_{a..b}) & = & 6\times 2-0\times 7-22 & = & -10 \\ GainZone(s_{g'..h'}) & = & 12\times 2-0\times 7-22 & = & 2 \\ GainZone(s_{i..j}) & = & 18\times 2-2\times 7-22 & = & 0 \end{array}$$

Le facteur $s_{i..j}$ n'est donc pas une RTA, le facteur $s_{a..b}$ non plus. La seule RTA est le facteur $s_{g'..h'}$.

Suite à ces propriétés, chaque zone de RTA commence et se termine par une copie exacte du motif (pour s'en convaincre, il suffit de remarquer que tous les mots de $Sim(M)$, à l'exception de M lui-même, induisent une perte en compression. Le seul gain en compression est produit par une copie exacte du motif). En poussant le raisonnement plus loin, **chaque zone de RTA est délimitée par deux RTE maximales** comme le montre la figure 4.6.

$$\ldots\underbrace{\overbrace{MM\ldots MM}^{\text{RTE maximale}}\ldots\ldots\ldots\overbrace{MM\ldots MM}^{\text{RTE maximale}}}_{\text{RTA}}$$

FIG. 4.6 - *Toute RTA est délimitée par deux RTE maximales*

Cette propriété essentielle permet de séparer le travail de RECH_RTA en deux étapes :

Étape 1 : Localisation de toutes les zones de RTE maximales. L'algorithme effectue un passage gauche-droite sur la séquence et repère toutes les occurrences du motif. Puisque la taille p du motif est de 3 au maximum, il n'est pas utile d'utiliser un algorithme évolué de "string matching", on considère que la recherche s'effectue en temps linéaire.

Soient $r_1, r_2, \ldots r_k$ les zones de RTE maximales de la séquence S.

Étape 2 : L'algorithme tente de joindre les zones de RTE maximales en factorisant les mots qui les séparent par des mots de $Sim(M)$.

L'algorithme construit une zone RTA à la fois. Soit w la zone courante. On l'initialise en lui donnant la valeur de la zone RTE maximale courante (les RTE maximales sont parcourues de gauche à droite) : $w \leftarrow r_i$. Elle est ensuite étendue vers la droite tant que c'est possible. Soit f le facteur qui sépare r_i de la RTE maximale suivante r_{i+1} :

$$\ldots\underbrace{=======\overbrace{----}^{r_i}}_{w}\underbrace{\ldots\ldots\ldots}_{f}\overbrace{--}^{r_{i+1}}\ldots$$

S'il est factorisable en mots de $Sim(M)$, alors l'extension de w vers la droite est possible. Soit $w' = w.f.r_{i+1}$ la nouvelle zone.

Si $GainZone(w') > GainZone(w) + GainZone(r_{i+1})$, alors la nouvelle zone w peut être étendue jusqu'à r_{i+1}, le gain en compression est meilleur si les deux zones w et r_{i+1} sont jointes. Soit w' la nouvelle zone courante et r_{i+1} la nouvelle zone RTE maximale courante. L'extension de la zone courante est poursuivie de la même manière avec w' et r_{i+1}.

Lorsque ce processus ne peut être poursuivi (parce qu'il n'est plus possible de factoriser f en mots de $Sim(M)$ ou parce que la nouvelle zone courante ne possède pas un gain suffisant), alors la zone courante w ne sera plus modifiée. Elle est acceptée comme RTA si $GainZone(w) \geq 0$.

L'algorithme construit alors la RTA suivante à partir de la RTE maximale courante et ainsi de suite. Toutes les RTE maximales sont ainsi parcourues de la gauche vers la droite.

La factorisation de f en mots de $Sim(M)$ est réalisée par une procédure heuristique. Elle utilise un algorithme glouton de factorisation de gauche à droite qui ne remet jamais en cause le mot de $Sim(M)$ choisi. Elle favorise les mots de $Sim(M)$ les moins coûteux et ceux qui n'apportent pas de décalage dans la répétition (c'est-à-dire qu'elle défavorise les insertions ou les délétions au profit des substitutions). Elle peut faire un mauvais choix de mot qui bloque la factorisation mais ceci est rare [Riv96]. La procédure de factorisation fonctionne en temps $O(|f|)$, l'étape 2 est donc linéaire en temps.

Puisque l'étape 1 et l'étape 2 fonctionnent toutes deux en temps linéaire, RECH_RTA a une complexité en temps de $O(n)$ où n est la longueur de la séquence.

Soient $w_1, w_2, \ldots w_z$ les z zones de RTA trouvées à l'aide de RECH_RTA, le gain en compression de la séquence S est égal à la somme des gains des zones de RTA moins le coût fixe C_{RTA} du codage de la séquence (coût du codage de la longueur du motif, du motif lui-même et du nombre de zones, voir section 4.4.2) :

$$Gain_{RTA}(S) = \sum_{i=1}^{z} GainZone(w_i) - C_{RTA}$$

4.4.4 Localisation de RTA dans des chromosomes de la levure

La méthode heuristique de localisation de RTA de motifs de longueurs $1, 2, 3$ a été expérimentée sur un certain nombre de séquences dont les quatre premiers chromosomes séquencés de la levure, des séquences de procaryotes (Escherichia coli et Bacilus subtilis) et des séquences générées aléatoirement [Riv96]. Ces expérimentations ont permis de montrer notamment que les RTA ne sont pas seulement présentes chez l'homme et qu'elles ne sont pas limitées aux tri-nucléotides. Les RTA sont uniformément réparties le long des chromosomes. Les motifs choisis préférentiellement comme motifs de base des RTA chez la levure semblent invalider l'hypothèse d'amplification des RTA par bégaiement de l'ADN polymérase dont nous avons parlé à la section 4.1.

Nous décrivons rapidement le protocole expérimental utilisé, nous montrons quelques résultats de la méthode appliquée aux chromosomes de levure et aux autres séquences, et donnons quelques conclusions biologiques qui peuvent être tirées des résultats. La principale source d'information utilisée dans cette section est [Riv96].

4.4.4.1 Protocole expérimental

La méthode de localisation de RTA développée est limitée aux séquences de petites tailles. En effet :

- Les entiers sont codés sur un nombre fixé de bits. Ce nombre de bits est directement lié aux nombres maximaux qui peuvent être codés et donc à la longueur de la séquence. Le fait de consacrer un nombre fixé de bits à chaque entier permet de calculer facilement le gain d'une zone RTA, ce qui rend l'algorithme linéaire.
- Toutes les RTA d'une séquence ont le même motif de base, ce qui est peu probable dans une longue séquence.

Dès lors, pour rechercher des RTA dans de longues séquences, telles que les séquences des chromosomes de levure, la méthode doit être adaptée. Puisque les répétitions en tandem sont des structures locales, une longue séquence est découpée en fenêtres consécutives de longueur constante. La méthode est alors appliquée sur chacune des fenêtres avec un motif (éventuellement) différent.

La longueur choisie pour les fenêtres est de 500 nucléotides (il s'agissait, dans un premier temps, de pouvoir comparer les résultats avec ceux d'une étude statistique menée sur les chromosomes de levure [ODH95]). Chaque fenêtre est soumise à trois expérimentations : une pour chaque longueur p de motifs (1, 2 ou 3). Pour une fenêtre et une longueur de motif fixés, le motif M qui apparaît le plus dans la fenêtre (on compte les motifs chevauchant) est choisi.

L'algorithme RECH_RTA localise alors les RTA de M dans la fenêtre. Les résultats de l'expérimentation sur une fenêtre sont donc :

- Le motif choisi.
- Le gain en compression pour la fenêtre.
- Le nombre de zones, et pour chacune d'elles :
  - Le gain en compression de la zone.
  - Le nombre d'occurrences du motif.
  - La liste des mutations codées.

Exemple 4.14 *La figure 4.7 représente le résultat de l'application de* RECH_RTA *à la fenêtre de 500 nucléotides, débutant à la position* 66000 *dans le chromosome* 2 *de la levure. La longueur de motif est* 3. *Le motif le plus présent détecté par l'algorithme est* $M = CAA$, *deux zones RTA ont été détectées, le gain total est de* 84 *bits. La première zone contient* 4 *répétitions du motif, elle ne possède aucune mutation, elle produit un gain de* 6 *bits. La deuxième zone possède* 21 *répétitions du motif dont deux sont mutées : deux occurrences de* CAg *font partie de la factorisation en mots de* $Sim(M)$. *Elle produit un gain de* 89 *bits.*

4.4.4.2 Expérimentations

Les expérimentations ont été réalisées essentiellement sur les quatre chromosomes de la levure disponibles à ce moment-là : le chromosome II [FAA+94] (807188 pb), III [OdAAC+92] (315357 pb) , VIII [Ja94] (562638 pb) et XI [DAAa94] (666448 pb)). La levure a été choisie car c'était le premier organisme eucaryote pour lequel on disposait de séquences complètes de

```
Fichier: c2.66000  Algo: 3 Motif: CAA NbOccur: 44 Gain:  84

      0         1         2         3         4           Zone
      01234567890123456789012345678901234567890123456789
-------------------------------------------------------------
  0   AGTAATTCCTTCCAGTCTCACAATGCGCCCTCCCACCAGTCGAACTACCA

 50   CCCCCATTACAATCACATGAAATACAACAACACTGGTAGCTATTACTATT

100   ACAACAACAACAATAACAGCAGTGTAAACCCACATAACCAAGCTGGTCTA
       ------------                                      1
150   CAATCCATTAACAGATCTATTCCATCGGCCCCGTACGGGGCTTACAACCA

200   GAACAGAGGCTAATGACGTACCATATATGAATACCCAAAAGAAACACCACA

250   GATTTAGCGCTAACAATAATTTGAACCAGCAAAAATACAAGCAATATCCC

300   CAGTATACGTCCAATCCAATGGTTACTGCACATCTGAAGCAAACGTACCC

350   TCAACTGTACTACAATAGCAACGTCAATGCTCACAACAACAACAACAACA
                                  ----------------        2
400   GCAACAACAACAACAACAACAACAACAACAGCAACAACAACAACAATCTT
      ---------------------------------------------       2
450   TACAACCAGACGCAGTTCTCCACGAGGTACTTCAACTCGAACTCCTCTCC


Zone  1 : 101-113 :   4 repet,   0 erreurs, Gain :   6
=======

Zone  2 : 383-446 :  21 repet,   2 erreurs, Gain :  89
=======
     Position      Type d'Erreur      Lettre       Num Erreur
        6             SUBS_3           G            15
       16             SUBS_3           G            15
```

FIG. 4.7 - *Résultat de* RECH_RTA *appliqué au segment [66000-66500] du chromosome* 2 *de la levure pour les motifs de longueur* 3

chromosomes. Puisque les RTA ne sont pas, a priori, limitée à l'homme, il était intéressant de savoir si elles pouvaient caractériser certaines zones des chromosomes. Le chromosome est une entité moléculaire complète alors qu'une séquence d'ADN quelconque n'en est qu'un morceau.

La méthode a été également appliquée sur des séquences des deux bactéries Escherichia coli et Bacilus subtilis (procaryotes) pour permettre de comparer les résultats.

Enfin, pour tester si la méthode ne détectait pas les zones qui avaient toutes les chances d'apparaître dans des séquences aléatoires, elle a été testée sur de telles séquences.

Le tableau 4.1 effectue un récapitulatif du nombre de fenêtres examinées au total dans chacun des quatre chromosomes de levures, du nombre de fenêtres comprimées et du pourcentage de fenêtres comprimées. Le tableau 4.2 compare les nombres de fenêtres comprimées dans les chromosomes pour chacune des tailles de motifs.

La colonne "Comp" donne le nombre de fenêtres comprimées, "% comp" le pourcentage du nombre de fenêtres comprimées pour ce motif par rapport au nombre total de fenêtres comprimées et "% total" le pourcentage par rapport au nombre de fenêtres examinées. Le tableau 4.3 montre les gains moyens et maximaux par chromosome et par longueur de motif.

Ces tableaux montrent que les RTA sont présentes dans les mêmes proportions dans chacun des quatre chromosomes (tableau 4.1) et que les proportions de fenêtres comprimées à l'aide d'une longueur de motif déterminées sont également pareilles (tableau 4.2). Le nombre de fenêtres comprimées pour une longueur de motif 1 est beaucoup plus important que pour les longueurs de motifs 2 et 3 mais les gains moyens de compression sont moins forts (tableaux 4.2 et 4.3). Les gains maximaux de compression sont toujours obtenus à l'aide d'une longueur 2 ou 3.

Chromosome	Total	Comprimées	Pourcentage
2	1614	140	8.67
3	630	62	9.84
8	1124	109	9.69
11	1332	120	9.00

TAB. 4.1 - *Récapitulatif du nombre de fenêtres comprimées dans les quatre chromosomes*

Chr	Motifs de longueur 1			Motifs de longueur 2			Motifs de longueur 3		
	Comp	% comp	& total	Comp	% comp	& total	Comp	% comp	& total
2	107	76.4	6.29	11	7.8	0.68	22	15.6	1.36
3	42	67.7	6.66	11	16.6	1.74	9	14.5	1.42
8	76	69.7	6.76	16	14.6	1.42	17	15.6	1.51
11	84	70.0	6.30	14	11.6	1.05	22	18.3	1.65

TAB. 4.2 - *Nombre de fenêtres comprimées dans les quatre chromosomes pour les trois longueurs de motifs*

Récapitulatif			Motifs de lg 1		Motifs de lg 2		Motifs de lg 3	
Chr	Moy	Max	Moy	Max	Moy	Max	Moy	Max
2	11.5	121	7	37	15	50	32	121
3	14	298	9	65	37	298	9	30
8	12.2	144	8	46	26	144	18	65
11	14	292	7	46	19	89	38	292

TAB. 4.3 - *Gain moyen et maximal par chromosome pour chacune des longueurs de motifs*

La répartition des fenêtres comprimées le long d'un chromosome semble uniforme, quel que soit le chromosome et quelle que soit la longueur du motif. A priori, la localisation de RTA de motifs de longueurs 1, 2 et 3 ne permet pas de déduire une quelconque structure des chromosomes (du moins chez la levure). La figure 4.8 représente les gains en compression obtenus dans les fenêtres du chromosome 11 à l'aide d'une longueur de motif 1. Les fenêtres non comprimées ne sont pas représentées. Chaque trait vertical représente la fenêtre comprimée dont l'abscisse est la position de début de la fenêtre. La hauteur du trait est le gain de compression obtenu pour cette fenêtre. Bien entendu, les graphiques pour les longueurs 2 et 3 sont moins "denses" vu le nombre moins élevé de fenêtres comprimées mais la répartition semble tout aussi uniforme.

Les mêmes expérimentations ont été réalisées sur de longues séquences de Escherichia coli et Bacilus subtilis, le récapitulatif des résultats est donné dans le tableau 4.4. Très peu de fenêtres sont comprimées chez ces deux organismes procaryotes, de plus le gain en compression moyen est très faible. Pour Bacilus subtilis, le nombre de fenêtre comprimées semble plus grand mais il est difficile d'effectuer des comparaisons lorsque si peu de fenêtres sont comprimées.

L'expérimentation de la méthode sur des séquences générées aléatoirement[3] donne des

3. Les séquences sont générées à l'aide de modèles de Markov d'ordre 1 déterminés par les fréquences observées des di-nucléotides dans les quatre chromosomes.

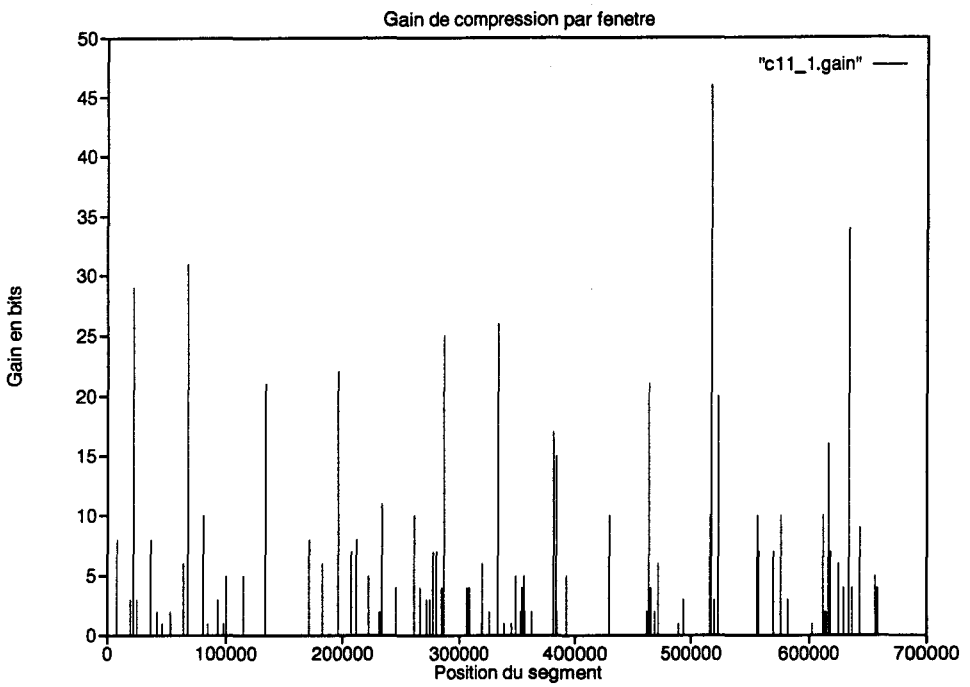


FIG. 4.8 - *Répartition des fenêtres comprimées le long du chromosome 11 pour les motifs de longueur* 1

Organisme	Nombre de fenêtres			Gain	
	Total	Comprimées	Pourcentage	Moyen	Max
Escherichia coli	1025	2	0.19	2	2
Bacilus subtilis	660	10	1.51	1.5	11

TAB. 4.4 - *Récapitulatif des résultats chez Escherichia coli et Bacilus subtilis*

résultats semblables à l'expérimentation sur Escherichia coli. Ce constat est très important car il montre que les fenêtres comprimées dans les chromosomes de levure ne sont pas dues au hasard.

4.4.4.3 Invalidation de la contrainte d'auto-appariement dans l'hypothèse de bégaiement de l'ADN polymérase

Les expérimentations ont été effectuées en collaboration avec M-O. Delorme, A. Hénaut et E. Ollivier (voir la référence [RDD+97] en annexe C), statisticiens-biologistes à l'université de Versailles-Saint-Quentin. Grâce à eux, quelques interprétations biologiques ont pu être avancées. Nous en mentionnons une importante sans la détailler.

Dans la section 4.1, nous avions émis l'hypothèse d'amplification des RTA par bégaiement de l'ADN polymérase lors de la réplication. Cette hypothèse ne semble possible que si le motif de la RTA a la faculté de s'auto-apparier pour que la RTA puisse adopter une structure en épingle à cheveux (voir figure 4.5). Pour la longueur de motifs égale à 3 (il n'y a pas d'auto-appariement possible pour des motifs plus courts), les expérimentations ont mis en évidence

des classes sur-représentées de motifs répétés en tandem. Les motifs de ces classes n'ont pas la faculté de pouvoir s'auto-apparier. L'hypothèse d'amplification de RTA par bégaiement de l'ADN polymérase est donc contredite, au moins chez la levure.

4.5 Méthode exacte : utilisation de l'optimisation par liftings

L'approche que nous développons ici est fondamentalement différente de l'approche introduite dans la section précédente. Elle s'intéresse à la localisation garantie de RTA d'un motif M dans une séquence S. La méthode ne souffre d'aucun paramètre :

- La séquence est de taille quelconque (elle n'est pas découpée en fenêtres).
- Le motif est de taille quelconque.
- Toutes les mutations sont autorisées.
- Seule la compression guide la sélection précise des zones.
- Tout est codé en longueur variable, aucune taille maximale ne vient limiter les possibilités de l'algorithme (si ce n'est bien entendu, les limitations de la machine sur laquelle il est implémenté).

La méthode d'optimisation de courbes par liftings est utilisée pour localiser les zones "régulières" (les RTA) et les zones "non régulières" (le reste). Cette utilisation est directement inspirée du chapitre 1.

Le but est également d'illustrer l'utilité de l'optimisation par liftings à l'aide d'un exemple concret.

4.5.1 Présentation générale

Nous présentons l'objectif de notre méthode et la séparation de son travail en trois étapes distinctes : l'analyse, le codage et l'optimisation de codage à l'aide de TURBOOPTLIFT.

La méthode EXACTEMENTRTA que nous développons ici localise toutes les zones RTA, d'un motif de longueur quelconque, dans une séquence de longueur quelconque également. Le terme RTA est à considérer dans un sens moins restrictif que celui utilisé dans la section 4.4. Une RTA d'un motif M est un mot de $\mathcal{N}^+$, obtenu par application d'une suite finie de *transformations élémentaires* (substitutions, insertions et délétions d'une base) à partir d'un préfixe d'une Répétition en Tandem Exacte (RTE) de M. Toute suite finie de transformations élémentaires est appelée une *transformation*. Les concepts de transformation et de transformations élémentaires sont définis formellement dans la section suivante.

Exemple 4.15 *Soit* $M =$ `ACTGGC` *le motif et* $t =$ `AGTGAGTAACAACGACTATCGTCACTGACTTCA`. *Le mot* t *est une RTA de* M. *En effet, soit* $r =$ `ACTGGCACTGGCACTGGCACTGGCACT` *préfixe de* M^∞. *Il s'aligne avec* t *de la façon suivante :*

```
t  =  AgTGaGt aACaacGaC tA-TcGtC A-ctGaCt tCa
r  =  AcTG-Gc -ACt--GgC -AcTgG-C ActgG-C- aCt
```

L'alignement détermine la transformation à appliquer à r *pour obtenir* t :

- *Substituer la* $2^{ème}$ *base par* `G`.

- *Insérer un A après la* $4^{ème}$ *base.*
- *Substituer la* $6^{ème}$ *base par T.*
- ...

Bien entendu, grâce à cette définition, n'importe quel mot t est RTA de n'importe quel motif M. Il suffit de choisir un préfixe d'une RTE de M et d'appliquer la transformation adéquate pour construire t.

La différenciation entre les "vraies" zones de RTA et les "fausses" zones de RTA est réalisée à l'aide de la compression. Si un facteur de la séquence peut être comprimé par utilisation d'un schéma de codage adapté aux RTA, alors il est qualifié de RTA, autrement il ne l'est pas. Dans la suite, les "vraies" zones de RTA sont appelées VRTA et les "fausses" zones FRTA. Dorénavant, le terme général RTA désigne un mot dont on ne sait pas s'il est VRTA ou FRTA mais dont on tente d'exploiter la ressemblance avec une RTE.

Remarque 4.4 *Nous ne donnons pas de définition formelle des VRTA (resp. FRTA). Ce sont simplement des facteurs de la séquence qui possèdent la propriété d'être compressibles (resp. de ne pas être compressibles) dans un certain sens qui est précisé clairement à la section 4.5.5.*

L'originalité de la méthode EXACTEMENTRTA réside dans sa faculté de localiser le début et la fin de toutes les zones VRTA de manière optimale, c'est-à-dire de choisir précisément quels sont les facteurs à considérer comme zones VRTA de manière telle que le gain de compression global de la séquence soit optimal.

La localisation optimale des zones VRTA est effectuée par l'algorithme TURBOOPTLIFT d'optimisation de courbes par liftings. Nous savons que TURBOOPTLIFT peut être utilisé efficacement pour décomposer une séquence, de manière optimale, en facteurs "réguliers" et facteurs "non réguliers" (voir chapitre 1, section 1.2). Dans le cas présent, un facteur "régulier" est une zone VRTA et un facteur "non régulier" est une zone FRTA.

Soit $M \in \mathcal{N}^p$ un motif et $S \in \mathcal{N}^n$ la séquence dans laquelle nous devons localiser les VRTA. Les longueurs p et n sont quelconques[4].

Puisque tout mot est une RTA de M, c'est le cas de la séquence S : elle peut être construite par application d'une transformation à un préfixe fini de M^∞.

La méthode EXACTEMENTRTA localise les VRTA en trois étapes **indépendantes** : la première recherche la transformation de coût minimal qui permet de construire S à partir d'un préfixe fini de M^∞. La seconde utilise cette transformation pour comprimer la séquence. La troisième étape optimise, à l'aide de TURBOOPTLIFT, le codage effectué dans la deuxième étape. Elle décompose la séquence en une succession de zones VRTA et FRTA. Les zones VRTA sont les facteurs de la séquence pour lesquels la compression effectuée à l'étape 2 est bien adaptée. Les zones FRTA sont les facteurs pour lesquels la compression n'est pas adaptée, il est préférable de les coder tels quel à l'aide du codage NUC^*.

Étape 1 : À chaque transformation permettant de construire S, on associe un *coût* égal à la somme des coûts de ses transformations élémentaires. Les coûts des transformations élémentaires sont déterminés par une matrice de coûts telle qu'elle a été définie dans le chapitre 2, page 140. En pratique, nous choisissons la matrice des coûts de la distance

4. Le problème de localisation avec $n < p$ n'a pas vraiment de sens mais il est autorisé.

de Levenshtein. Parmi toutes les transformations permettant de construire S à partir d'un préfixe fini de M^∞, il en existe une (ou plusieurs) qui est (sont) optimale(s): elle(s) possède(nt) un coût minimal. **Cette étape recherche, en temps $\theta(np)$, une transformation optimale.** La méthode utilisée est similaire à la méthode d'alignement WDP [FLSS92], nous la développons dans la section 4.5.2

Étape 2: La transformation optimale calculée à l'étape 1 est utilisée pour comprimer la séquence S. Puisque S est une RTA du motif M, elle est complètement déterminée par la suite des transformations élémentaires de la transformation optimale et la longueur du préfixe de M^∞ associé à cette transformation. La deuxième étape effectue un codage, de la gauche vers la droite, des transformations élémentaires de la transformation optimale. Elle est décrite dans la section 4.5.3.

Étape 3: Si la transformation optimale contient très peu de transformations élémentaires, alors le codage effectué à l'étape 2 est efficace: le gain de compression est positif. Il se peut que certains facteurs de la séquence défavorisent le codage, c'est-à-dire que la longueur du codage des transformations élémentaires qui permettent de construire un tel facteur soit plus grande que la longueur du codage du facteur tel quel (à l'aide de NUC^*) précédé de sa longueur auto-délimitée. Un tel facteur doit être qualifié de FRTA. **Le but de la troisième étape est de localiser les facteurs qui doivent être qualifiés de VRTA ou de FRTA.** Elle utilise l'algorithme TURBOOPTLIFT appliqué à la courbe de compression résultant du codage de l'étape 2. Les portions de ruptures détectées par TURBOOPTLIFT sont qualifiées de zones FRTA, les portions d'applications de zones VRTA. La localisation est optimale vis-à-vis du gain de compression global de la séquence. Cette utilisation de TURBOOPTLIFT s'inspire directement de ce qui a été exposé dans la section 1.2 du chapitre 1. Elle est décrite à la section 4.5.5.

4.5.2 Alignement optimal: approche par transducteurs

Étant donné la séquence $S \in \mathcal{N}^n$ et $M \in \mathcal{N}^p$, nous présentons une méthode permettant de calculer la transformation optimale qui construit S à partir d'un préfixe de M^∞ en temps $\theta(np)$. Nous commençons par présenter formellement le concept de *transformation* et de *transformations élémentaires* et définissons le coût associé à une transformation. Nous démontrons alors que la transformation optimale est calculée en temps $\theta(np)$. Nous terminons par la modélisation, à l'aide d'un transducteur, du calcul de la transformation optimale. La méthode est similaire à la méthode d'alignement WDP de Fischetti, Landau, Schmidt et Sellers [FLSS92].

4.5.2.1 Transformation, transformations élémentaires, coût et phase

Étant donné un motif $M \in \mathcal{N}^p$, une *transformation* permet de construire une séquence finie à partir d'un préfixe fini de M^∞.

Définition 4.5 *Soit $\mathcal{T} = \{S_A, S_C, S_G, S_T, D, I_A, I_C, I_G, I_T, M\}$ l'alphabet des* transformations élémentaires. *Tout mot fini $T \in \mathcal{T}^*$ est appelé une* transformation.

La construction d'une séquence $x \in \mathcal{N}^+$ à partir d'une transformation $T = t_1 t_2 \dots t_l \in \mathcal{T}^*$ et du mot infini M^∞ suit le procédé suivant: les symboles de T et de M^∞ sont parcourus

un à un de la gauche vers la droite. Soit $t_i \in \mathcal{T}$ le symbole courant de T et m_j le symbole courant de M^∞. La séquence x est construite, symbole par symbole de la gauche vers la droite en fonction des caractères courants t_i et m_j. Les symboles $\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}$ représentent la substitution du caractère courant m_j, les caractères $\mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}$ l'insertion d'un nouveau caractère, D la délétion du caractère m_j et M, la conservation du caractère m_j (dans la suite, nous dirons que cette transformation élémentaire est un *match*).

Le symbole courant t_i est considéré, en fonction de sa valeur, les actions entreprises sur x sont représentées dans le tableau 4.5.

Type	t_i	Influence sur x	Déplacement dans M^∞
Substitutions	$\mathtt{S_A}$	Le symbole A est concaténé à x	$j \leftarrow j+1$
	$\mathtt{S_C}$	Le symbole C est concaténé à x	$j \leftarrow j+1$
	$\mathtt{S_G}$	Le symbole G est concaténé à x	$j \leftarrow j+1$
	$\mathtt{S_T}$	Le symbole T est concaténé à x	$j \leftarrow j+1$
Insertions	$\mathtt{I_A}$	Le symbole A est concaténé à x	
	$\mathtt{I_C}$	Le symbole C est concaténé à x	
	$\mathtt{I_G}$	Le symbole G est concaténé à x	
	$\mathtt{I_T}$	Le symbole T est concaténé à x	
Délétion	D		$j \leftarrow j+1$
Match	M	Le symbole courant m_j est concaténé à x	$j \leftarrow j+1$

TAB. 4.5 - *Actions entreprises sur x en fonction de la transformation élémentaire t_i.*

Après le traitement du symbole t_i, le symbole suivant de T est considéré : $i \leftarrow i+1$. La construction de x se termine lorsque tous les symboles de T ont été parcourus. Puisque T est un mot fini, la séquence x construite l'est également.

Puisque M^∞ est un mot périodique, les déplacements $j \leftarrow j+1$ sont effectués de manière cyclique dans M : lorsque j vaut $p+1$, il est réinitialisé à 1. La valeur de j est donc toujours comprise entre 1 et p.

Une notion importante, qui sera souvent considérée dans la suite, est la *phase* d'une transformation.

Définition 4.6 *La* phase *k de la transformation T est la valeur de l'indice j après construction de la séquence x par le processus présenté ci-dessus.*

Exemple 4.16 *Soit $M = ATC$ le motif. La séquence $x = ATTTCAGTGATCATCA$ est obtenue par application de la transformation $T = MMS_TDMMMS_GS_TI_GMMMMMM$ à M^∞.*

Exprimé en terme d'alignement, on a :

$$\begin{array}{rcl} x & = & ATt\text{-}TCAgtgATCATCA \\ M^\infty & = & ATc\,aTCAtc\text{-}ATCATCA\ldots \end{array}$$

La phase de T est 2 car le caractère courant de M après construction de x est $m_2 = T$.

Un *coût* est attribué à chacune des transformations élémentaires. Cette notion de coût est exactement la même que la notion de coût des paires alignées dans le contexte de l'alignement (voir chapitre 2, section 2.5.1). Notons $c(t_i)$ le coût de la transformation élémentaire $t_i \in \mathcal{T}$. Nous choisissons ici les coûts de la distance de Levenshtein :

- $c(\mathtt{M}) = 0$: un match ne coûte rien.

- $c(\mathtt{S_A}) = c(\mathtt{S_C}) = c(\mathtt{S_G}) = c(\mathtt{S_T}) = 1$: les substitutions ont un coût de 1.
- $c(\mathtt{I_A}) = c(\mathtt{I_C}) = c(\mathtt{I_G}) = c(\mathtt{I_T}) = 1$: les insertions ont un coût de 1.
- $c(\mathtt{D}) = 1$: la délétion a un coût de 1.

Remarque 4.5 *Le choix de cette fonction de coût est arbitraire. A priori, nous ne disposons d'aucune information objective capable de guider ce choix.*

Cette fonction de coût est très connue, elle est intéressante d'un point de vue combinatoire. C'est un point de départ naturel avant de considérer des fonctions de coût plus complexes [CL92, GBN94].

Dans la section 4.5.3, nous comprimons une séquence en codant la suite de transformations élémentaires de la transformation optimale. Il nous semble donc naturel, pour le moment, de donner un coût nul au match (M) et un coût non nul aux autres transformations élémentaires pour que le gain de compression exprime si une séquence "ressemble" à un préfixe fini de M^∞.

La méthode de recherche de la transformation optimale que nous développons ici peut être adaptée à d'autres fonctions de coûts sans trop de difficultés. C'est d'ailleurs ce qui est réalisé dans [FLSS92].

Le coût d'une transformation $T = t_1 \dots t_l$ est égal à la somme des coûts de ses transformations élémentaires :

$$c(T) = \sum_{i=1}^{l} c(t_i)$$

Exemple 4.17 *Le coût de la transformation présentée dans l'exemple 4.16 est* 5.

La proposition 4.1 montre que le préfixe d'une transformation engendre un préfixe de la séquence.

Proposition 4.1 *Soit $T \in \mathcal{T}^*$ la transformation qui engendre $x \in \mathcal{N}^n$ à partir du motif $M \in \mathcal{N}^p$. Le préfixe de transformation $T' = T_{..i}$ engendre un préfixe $x' = x_{..a}$ de x.*

Preuve. Considérer le préfixe T' au lieu de T revient à interrompre le processus de construction de x à un moment précis. Puisque x est construit de gauche à droite par concaténations successives de caractères, lorsque le processus est interrompu, un préfixe $x_{..a}$ a déjà été construit. □

Il est important de constater que le préfixe d'une transformation n'est pas forcément de même phase.

Exemple 4.18 *Le préfixe $T' = \mathtt{MMS_TDMM}$ de la transformation de l'exemple 4.16 engendre le préfixe $x' = \mathtt{ATTTC}$ de x. La phase de T' est* 1.

4.5.2.2 Calcul de la transformation optimale

Soit $S = s_1 s_2 \dots s_n \in \mathcal{N}^*$ une séquence et $M = m_1 m_2 \dots m_p \in \mathcal{N}^+$ un motif. Nous démontrons que le calcul d'une transformation optimale $T = t_1 t_2 \dots t_l \in \mathcal{T}^*$ qui construit S à partir du motif M se calcule en temps $\theta(np)$.

Dès à présent une transformation sous-entend toujours une transformation de M^∞.

Notation 4.1 *Soit $U \in \mathcal{N}^*$ une séquence et $k : 1 \leq k \leq p$ un indice dans M. On note $T_k(U)$ une transformation optimale de phase k qui construit U. Son coût est noté $c_k(U)$: $c_k(U) = c(T_k(U))$.*

Dès lors le coût de la transformation optimale T est le minimum entre les coûts des transformations optimales de chaque phase qui construisent S :

$$c(T) = \min\{c_k(S) | 1 \leq k \leq p\}$$

La proposition suivante montre que le préfixe d'une transformation optimale est optimale pour sa phase et pour le préfixe de S qu'elle construit.

Proposition 4.2 (Propriété des préfixes) *Soit $T' = T_{..i}$ un préfixe de la transformation optimale T. Soit k la phase de T' et $U = S_{..j}$ le préfixe de S construit par T'. Alors T' est une transformation optimale de phase k qui construit U. De plus, T' peut être remplacée par n'importe quelle autre transformation optimale de phase k qui construit U.*

Preuve. Supposons que R soit une transformation optimale de phase k qui construit U, avec $c(R) \leq c(T')$. Soit T'' le suffixe de T tel que $T = T'.T''$. Puisque la phase de R est égale à la phase de T' et puisque R et T' construisent tous les deux le préfixe U, alors $R.T''$ est une autre transformation qui construit S. Son coût est $c(R.T'') = c(R) + c(T'') \leq c(T)$, par définition de la fonction de coût c. Puisque la transformation T est optimale, on a $c(R) + c(T'') = c(T) = c(T') + c(T'')$. Dès lors, $c(T') = c(R)$ donc R est également une transformation optimale de phase k. □

Pour rechercher la transformation optimale T, l'algorithme construit les transformations $T_k(U)$ pour chaque phase k et tout préfixe U de S. La seule façon de construire le préfixe ϵ de S à partir d'un préfixe de longueur $k-1$ de M^∞ est de supprimer $k-1$ symboles. Dès lors, au départ $T_k(\epsilon) = \mathtt{D}^{k-1}$ et $c_k(\epsilon) = k-1$ pour toute phase k. L'algorithme construit alors toutes les transformations $T_k(S_{..i})$ en débutant avec le préfixe $S_{..1}$ et en terminant par le préfixe $S_{..n}$. Les étapes sont numérotées de 1 à n.

À l'étape i, il faut calculer $T_k(Ux)$ avec $|U| = i-1$ et $x \in \mathcal{N}$, pour tout $k : 1 \leq k \leq p$. Grâce à la propriété des préfixes, nous savons que $T_k(Ux) = T_a(U).t'_1 t'_2 \dots t'_j$ pour une phase a et $t'_b \in \mathcal{T}, 1 \leq b \leq j \leq p$. D'après la définition des transformations élémentaires et, puisque $|Ux| = |U| + 1$, tous les t'_b sont des délétions (D) excepté un t'_h qui est soit une substitution, une insertion ou un match. Donc $T_k(Ux) = T_a(U)\mathtt{D}^e t'_h \mathtt{D}^f$ avec $0 \leq e+f = j-1$, $t'_h \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}, \mathtt{M}\}$ et $c_k(Ux) = c_a(U) + e + f + c(t'_h)$.

La proposition 4.3 montre que si t'_h est une insertion, alors il n'y a aucune délétion : $e + f = 0$. Un match ou une substitution peut toujours remplacer un couple "délétion-insertion" ou "insertion-délétion".

Proposition 4.3 *Si $t'_h \in \{I_A, I_C, I_G, I_T\}$ alors $j = 1, a = k$ et $T_k(Ux) = T_k(U).t'_h$.*

Preuve. Supposons $e \geq 1$ (la preuve est similaire pour $f \geq 1$). Alors il existe une transformation élémentaire $t_g \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{M}\}$ qui peut remplacer la paire $\mathtt{D}t'_h$ dans $T_k(Ux)$ (la phase est identique et le facteur concerné de S est identique). Puisque $c(t_g) \leq 1$ et $c(\mathtt{D}t'_h) = 2$, $T_k(Ux)$ n'est pas optimal. Il y a contradiction. □

La proposition 4.4 montre que le nombre de délétions de $t'_1 t'_2 \ldots t'_j$ vaut au maximum $p-2$.

Proposition 4.4 *Si $t'_h \in \{S_A, S_C, S_G, S_T, M\}$, alors $e + f < p - 1$.*

Preuve. Supposons que $e + f = p - 1$. Dès lors, $\mathtt{D}^e t'_h \mathtt{D}^f$ parcourt exactement p symboles de M^∞ et donc la phase de $T_a(U)$ est identique à la phase de $T_k(U)$: $k = a$. Remarquons que $p - 1 \leq c(\mathtt{D}^e t'_h \mathtt{D}^f)$. Donc la transformation $T_a(U)t_g$, avec t_g l'insertion de x, est également de phase k, construit également Ux et a un coût $c(T_a(U)t_g) = c_a(U) + 1 \leq c_k(Ux)$ (on ne considère pas le cas trivial $p = 1$ puisque toutes les phases sont égales). Donc, dans ce cas, il est toujours préférable de choisir $T_k(Ux) = T_a(U)t_g$.

Supposons maintenant que $e + j \geq p$. Considérons la transformation $T_a(U)t_g\mathtt{D}^{e+f-p}$, avec t_g la substitution qui produit le symbole x. C'est aussi une transformation de phase k qui construit Ux et son coût est $c(T_a(U)t_g\mathtt{D}^{e+f-p}) = c_a(U) + 1 + e + f - p < c_k(Ux)$ (pour $p \geq 1$), ce qui contredit $T_k(Ux)$ optimal. □

Remarque 4.6 *Dans la démonstration, lorsque $e + f \geq p$, la transformation $T_a(U)t_g\mathtt{D}^{e+f-p}$ n'est pas forcément optimale. En effet, si x est un des $e + f - p + 1$ caractères suivants de M^∞, alors il existe e' et f', avec $e' + f' = e + f - p$ tels que la transformation $T_a(U)\mathtt{D}^{e'}M\mathtt{D}^{f'}$ coûte une substitution de moins que $T_a(U)t_g\mathtt{D}^{e+f-p}$.*

Les deux lemmes qui suivent établissent comment calculer $T_k(Ux)$.

Lemme 4.5 *Le temps nécessaire au calcul de $T_k(Ux)$, avec $k > 1$, lorsque $T_a(U)$ et $T_j(Ux)$ sont connus, pour $1 \leq a \leq p$ en $1 \leq j \leq k$ est $O(1)$.*

Preuve. D'après la propriété des préfixes et la proposition 4.3, la seule façon de construire $T_k(Ux)$ à partir des $T_a(U)$ sont $T_k(Ux) = T_k(U)t'$, avec $t' \in \{\mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}\}$, ou $T_k(Ux) = T_a(U)\mathtt{D}^e t'\mathtt{D}^f$, avec $e, f \geq 0$ et $t' \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{M}\}$. Donc, $T_k(Ux)$ est construite en considérant, parmi ces transformations, celle de coût minimal.

Si $T_k(Ux) = T_a(U)\mathtt{D}^e t'\mathtt{D}^f$ pour une phase a et $f > 0$, alors $T_a(U)\mathtt{D}^e t'\mathtt{D}^{f-1}$ est une transformation optimale de phase $k - 1$ qui construit Ux. Ce cas est donc couvert par le cas $T_k(Ux) = T_{k-1}(Ux)\mathtt{D}$. Grâce à un argument similaire, le cas $T_k(Ux) = T_a(U)\mathtt{D}^e t'$ est couvert par le cas $T_k(Ux) = T_{k-1}(U)t'$.

Dès lors, $T_k(Ux)$ est la transformation minimale parmi $T_k(U)t'_1$, $T_{k-1}(Ux)\mathtt{D}$, $T_{k-1}(U)t'_2$ avec $t'_1 \in \{\mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}\}$ et $t'_2 \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{M}\}$ selon que x est égal au caractère courant de M^∞ ou non. □

Lemme 4.6 *Le temps nécessaire au calcul de $T_1(Ux)$ lorsque $T_k(U)$ est connu, pour tout $k : 1 \leq k \leq p$, est en $\theta(p)$.*

Preuve. D'après la propriété des préfixes et, d'après les propositions 4.3 et 4.4, $T_1(Ux)$ est une des p transformations :

- $T_1(U)t'$ avec $t' \in \{\mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}\}$.

- $T_k(U)t'\mathtt{D}^{p-k}$ avec $1 < k \leq p$ et $t' \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{M}\}$, selon que x est égal au caractère courant de M^∞ ou non.

Le temps est en $\theta(p)$ car parmi ces p transformations, celle de coût minimal doit être choisie. □

Nous pouvons maintenant préciser le temps global du processus de calcul de la transformation optimale.

Théorème 4.7 *Le temps complet de calcul de la (d'une) transformation optimale pour S est en $\theta(np)$.*

Preuve. L'algorithme calcule tous les $T_k(U)$ avec U préfixe de S. Le calcul des p transformations initiales $T_k(\epsilon)$ est effectué en temps $\theta(p)$. Après cela, tous les préfixes sont considérés par ordre de longueur croissante. Pour un préfixe U le calcul de tous les $T_k(U)$ est réalisé en temps $\theta(p)$ ($\theta(p)$ pour le calcul de $T_1(U)$, voir lemme 4.6 et $O(1)$ pour chacune des p transformations $T_k(U)$, voir lemme 4.5). Le calcul complet est terminé après le parcours des n préfixes de S, soit en temps $\theta(np)$. □

4.5.2.3 Modélisation par transducteurs

Le formalisme des automates finis peut être utilisé pour modéliser des mécanismes biologiques simples de mutation [SM95, Lef96]. Cette approche est plus riche et plus élégante que les habituelles équations de récurrence dans lesquelles les méthodes classiques d'alignement par programmation dynamique sont exprimées. En pratique, des outils ont été créés qui permettent d'aligner automatiquement des séquences étant donné la modélisation du problème sous la forme d'automates. De telles modélisations sont beaucoup plus faciles à modifier que les équations classiques de récurrence. L'article [Lef96] effectue notamment une présentation d'un nouveau formaliste appelé *MTSAG (Multi-Tape S-Attribute Grammars)* permettant de décrire la majorité des modèles d'alignement et de repliement d'ADN, d'ARN et de protéines.

Nous nous restreignons ici à la modélisation du problème simple de calcul de la transformation optimale décrit dans la section précédente. Pour cela, nous définissons un transducteur fini non déterministe [HU79] particulier que l'on nomme *transformeur*. Le transformeur $\mathcal{F}_M$ est lié au motif M des RTA, il lit en entrée les symboles de la séquences S et produit, en sortie des transformations.

Définition 4.7 *Un* transformeur, *sur l'alphabet des nucléotides $\mathcal{N}$, avec sortie dans l'alphabet des transformations élémentaires $\mathcal{T}$, est la structure $\mathcal{F} = (Q, \mathcal{N}, \mathcal{T}, \delta, s, F)$ où :*

- *Q est un ensemble non vide d'*états.
- *$s \in Q$ est un état particulier appelé* état initial.
- *$F \subseteq Q$ est un sous-ensemble d'*états finaux.
- *$\delta \subset (Q \times (\mathcal{N} \cup \{\epsilon\}) \times \mathcal{T}^* \times Q)$ est un ensemble fini de* transitions.

  Pour chaque transition $d = \langle q, a, W, q' \rangle \in \delta$:

  - *q est son* état de départ.
  - *a est son* étiquette.

- *W est sa* sortie.
- *q' est son* état d'arrivée.

Si $a = \epsilon$, la transition d est appelée une ϵ-transition.

Un chemin *P de $\mathcal{T}$ est une suite consécutive de transitions: $P = (d_1, d_2, \ldots d_v)$, avec $d_i \in \delta$ pour tout $i : 1 \leq i \leq v$. L'état d'arrivée de chaque transition d_i de P (avec $i < n$) est l'état de départ de la transition suivante d_{i+1}.*
Pour le chemin $P = (d_1, d_2, \ldots d_v)$:

- *Son* étiquette *est le mot formé par concaténation des étiquettes de ses transitions.*
- *Sa* sortie *est le mot formé par concaténation des sorties de ses transitions.*
- *Son* premier état *est l'état de départ de sa première transition d_1.*
- *Son* dernier état *est l'état d'arrivée de sa dernière transition d_v.*
- *Son* coût *est égal à la somme des coûts des transformations élémentaires composant sa sortie.*

Le chemin P est réussi *si son premier état est l'état initial s et son dernier état appartient à F. Un mot $S \in \mathcal{N}^*$ est* accepté *par $\mathcal{T}$ s'il est l'étiquette d'au moins un chemin réussi de $\mathcal{T}$. Étant donné un mot accepté $S \in \mathcal{N}^*$, on note $\mathcal{T}(S)$ l'ensemble de tous les mots qui sont les sorties des chemins réussis d'étiquettes S. Si S n'est pas accepté par $\mathcal{T}$, alors $\mathcal{T}(S)$ est l'ensemble vide.*

Un transformeur (ou un transducteur d'une façon générale) est habituellement représenté par un graphe orienté dont chaque sommet est un état du transformeur. L'état initial est représenté par une flèche entrante, le cercle entourant un état final est dédoublé. Chaque transition $d = \langle q, a, W, q' \rangle \in \delta$ est représentée par un arc orienté (une *flèche*) qui mène du sommet q au sommet q'. La *légende* de l'arc est $a|W$ si a est l'étiquette de la transition et W sa sortie. Lorsque plusieurs transitions se partagent le même état de départ et le même état d'arrivée, elles peuvent être représentées par un seul arc possédant une légende pour chacune des transitions.

Exemple 4.19 *La figure 4.9 représente un transformeur $\mathcal{T}$ à 3 états. L'état initial est s,*

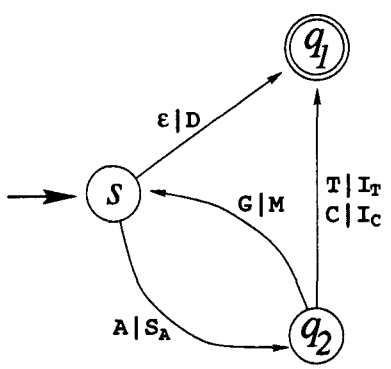


FIG. 4.9 - *Exemple de transformeur $\mathcal{T}$ à 3 états*

le seul état final est q_1. Soit S = AGAC. C'est un mot accepté par le transformeur, $\mathcal{T}(S)$ = $S_A MS_A I_C$.

Étant donné un motif $M \in \mathcal{N}^p$, considérons le transformeur $\mathcal{T}_M$ qui modélise la recherche de la transformation optimale pour toute séquence $S \in \mathcal{N}^n$. Il ne dépend que de $M = m_1 m_2 \ldots m_p$. Étant donné $S \in \mathcal{N}^n$, l'objectif de $\mathcal{T}_M$ est de rechercher des transformations qui construisent S. Toutes les transformations de l'ensemble $\mathcal{T}_M(S)$ ne sont pas optimales mais $\mathcal{T}_M(S)$ en contient au moins une qui est optimale.

Le transformeur $\mathcal{T}_M = (Q, \mathcal{N}, \mathcal{T}, \delta, s, F)$ a la structure suivante :

- Il contient $p+1$ états : $Q = \{s, q_1, q_2, \ldots q_p\}$.
- L'état initial est s.
- L'ensemble d'états finaux est $F = \{q_1, q_2, \ldots q_p\}$. Chaque état q_k correspond à une phase de transformation.
- Si $T \in \mathcal{T}_M(S)$ alors, cela signifie que T est une transformation qui construit S à partir de M^∞. Soit k la phase de T, alors q_k est le dernier état du chemin réussi dont S est l'étiquette et dont T est la sortie.
- Pour chaque état q_k, il existe une $\epsilon-$transition de la forme $\langle s, \epsilon, \mathrm{D}^{k-1}, q_k \rangle$ correspondant aux p transformations initiales $T_k(\epsilon)$.
- Les autres transitions sont directement déduites des preuves des lemmes 4.5 et 4.6 dans le sens suivant : soit q_k un état spécifique correspondant à la phase k. Si la transition $\langle q_a, x, T', q_k \rangle$ existe, avec $x \in (\mathcal{N} \cup \{\epsilon\})$, alors cela signifie qu'une transformation optimale de phase k, qui construit un mot Ux, peut être obtenue par concaténation de T' à une transformation optimale de phase a qui construit le mot U.

  De cette façon, $\mathcal{T}_M(S)$ contient une transformation optimale qui construit S pour chacune des p phases.

Exemple 4.20 *La figure 4.10 représente le transformeur $\mathcal{T}_M$ pour M = ATCA.*

La principale qualité du transducteur $\mathcal{T}_M$ est que pour toute séquence finie S, l'ensemble $\mathcal{T}_M(S)$ contient un nombre fini de transformations. En effet, entre deux transitions étiquetées par deux symboles consécutifs de S, il y a au plus $(p-1)$ ϵ-transitions.

Pour calculer efficacement la transformation optimale pour la séquence $S \in \mathcal{N}^n$, le comportement du transformeur doit être légèrement modifié. Après la lecture du préfixe $U = S_{..i}$ le transformeur ne doit se rappeler, pour chacun des états q_k, que d'un chemin réussi aboutissant en q_k : celui dont le coût est minimal. En effet, d'après la propriété des préfixes, les sorties des chemins aboutissant en q_k, et de coûts supérieurs, ne peuvent pas être les préfixes d'une transformation optimale pour S. Après la lecture de chaque symbole de S, au plus p chemins sont conservés. Lorsque tous les symboles de S ont été parcourus, la transformation optimale est une des p transformations conservées.

Exemple 4.21 *Considérons la séquence S de* 1000 *bases représentée à la figure 4.11. Il s'agit du segment des bases* 63700 *à* 64699 *du chromosome* 11 *de la levure.*

Soit M = TTC le motif. La transformation optimale calculée par notre méthode fournit l'alignement représenté à la figure 4.12. Le segment souligné à la figure 4.11 correspond au segment mis en évidence à l'aide des caractères "-------" à la figure 4.12.

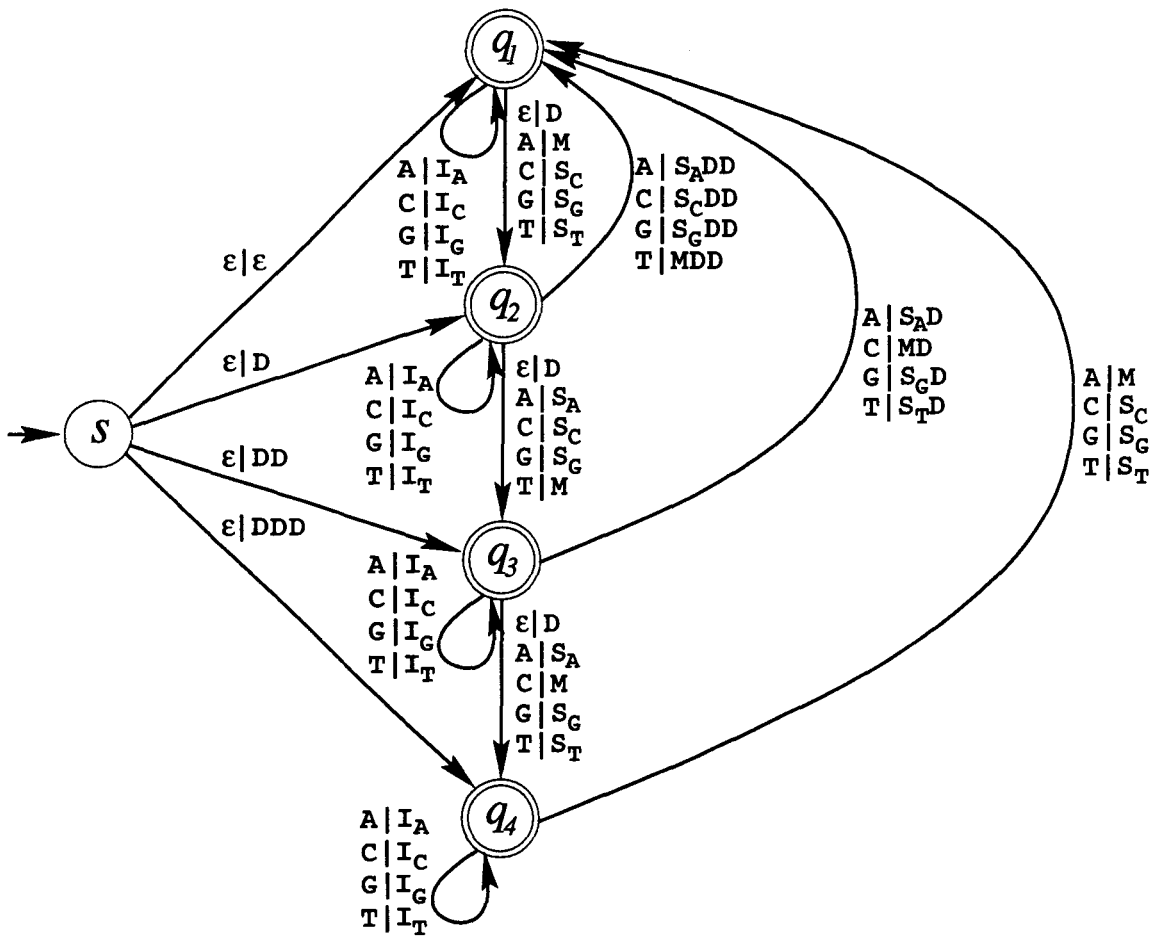


FIG. 4.10 - *Transformeur $\mathcal{T}_M$ pour $M = ATCA$*

TAATAGCATTAAGATTAAATGTATGTTTGATTAATTGCAATATCATGATGTTATAGAGTAACGTGGTATAATTACCACGA
TTCCGTCGAATTGTTTGCCACTTATCACTGGCGCCAAAATGCAGCAGTATGTAAAAAATGACGGTAATAAATATATAATA
CAAATGCATAGAGGAAAGGCTATAGAAATGAAGAGATTCATGAATTTTCAGTCAGGTTCTCCGACTTTTAATTGCTGTGC
CCCTCCTCCTGCTTCTTTTTTTCCTCTTCTTCGTTCTTCTTCTTTTCTTCCTCTTCCTGTTTCTTCTTTTCTTCATCTTC
ATTCTTCTTGTTTTTCTCTTCCTGCTTCTTTTTTTCCTCTTCTTCGTTCTTCTTCTTTTCTTCATCTTCATTCTTCTTAT
TTTCTTCATCTTCATTCTTCATCTTTTCTCCTTCTTCCTGCTTCTTCTTCTCTTCTTCTTCCTTCTTCTTTTTCTCCTCT
TCCTCCTGCTTCTTCTTTTCTTCTTCTTCCTTCTTTTTCTTCTCTTCTTCTTCCTTCTTTTTCTTCTCTTCCTCTTCCTT
CTTTTTCTTTTCTTCTTCTTCCTTCTTCTTCTTTTCCTCTTCTTCCTTCTTCTTCTTCTCCTCTTCCTCCTTCTTCTTCT
TCTCCCTCCGCTTTCTTTCTTCAAGTAACTTGGCATAAGCATAGTCCTCGTATACAATAATAGGGTCCTTATCAAACATT
TGAATACCCTTTCTGTTAACGTCGGATAAAGTTGGCGCAAAAAGGTCTAAACCATATGATTTTAAATCCAAAAAGGAGTT
GCTACTAAAGTTTAAATTTGATTTAAAATCCACAGGATTTACTTCAACAGTATTAGCAGGAGTATAATCTTCTGTGTTAT
CATCTAGAGTTGTTCCAGTTTCATCAAAAGTAATGTTGAAAAGTGAAAAATTCCTGTATTTCAAACTTTCGGACAGATCA
ATAATTTTTCCGACATATTCTTCATGAACAATCTTGGTAA

FIG. 4.11 - *Segment* [63700 – 64699] *du chromosome* 11 *de la levure*

4.5.3 Schéma de codage

Étant donné $T \in \mathcal{T}^*$ une transformation optimale qui construit $S \in \mathcal{N}^n$ à partir de M^∞, nous comprimons S par codage de la transformation T. Le schéma de codage utilisé ici doit être vu comme un exemple, d'autres schémas de codage peuvent facilement le remplacer. Cependant, puisque nous désirons optimiser le codage grâce à l'algorithme TURBOOPTLIFT, il nous faut, dès à présent, réserver un mot code spécifique pour l'annonce de rupture $a_\mathcal{R}$. De plus, certaines informations initiales, liées à l'optimisation par lifting, doivent être codées au début de la séquence comprimée.

Soit $C \in \mathcal{B}^+$ le codage complet de T. Puisque nous comprimons sans perte d'informations, l'algorithme devra être capable, étant donné la suite de bits C, de reconstruire la séquence S dans son intégralité. Le codage de la transformation T n'est pas suffisant pour reconstruire S puisque M^∞ doit être connu. Le codage C contient trois parties : $C = C_M.C_I.C_T$ avec

- C_M est le codage du motif M.
- C_I est le codage des informations initiales liées à l'optimisation par liftings.
- C_T est le codage de la transformation optimale.

Le codage C_M de M est simple, il suffit de faire précéder $NUC^*(M)$ du code auto-délimité de sa longueur (moins 1 car le motif possède toujours au moins une base) : $C_M = Fibo(p-1)NUC^*(M)$. Remarquons que *Fibo* est utilisé ici à titre d'exemple. N'importe quel autre codage auto-délimité d'entiers peut le remplacer.

Le mot code C_I correspond au codage auto-délimité du retard $r_\mathcal{R}$ utilisé par l'algorithme d'optimisation TURBOOPTLIFT (voir sa définition, partie II, chapitre 1) : $C_I = Fibo(r_\mathcal{R})$. Ce retard doit être connu de l'algorithme de décompression car les longueurs des ruptures sont codées de manière relative au retard $r_\mathcal{R}$. Dans le cas qui nous intéresse, on pose $r_\mathcal{R} = 2$ (voir section 4.5.5). Donc $C_I = Fibo(2) = 0011$. Nous considérons malgré tout que le retard n'est pas une constante et ceci pour rendre le processus d'optimisation paramétrisable.

Développons maintenant le codage de C_T. Puisque l'on désire exploiter le fait que la séquence S est une RTA de M, on va favoriser les matchs consécutifs. On suppose que

```
  1 T--T--C-TTc---TTct-TcT-TcTTc--TTc-TT-C--T-TC-T--TcTTcT----T--CtTc-T-Tc-TT-CttC--TTCt-TC---TTcTT--CttCTTcTt-CTt
  1 TaaTagCaTTaagaTTaaaTgTaTgTTtgaTTaaTTgCaaTaTCaTgaTgTTaTagagTaaCgTggTaTaaTTaCcaCgaTTCcgTCgaaTTgTTtgCcaCTTaTcaCTg

 24 -CttCt---T-CttC--T-TcT------T--C--T--Tc--T-TcT--T-Ct--T-CtT----------CT-Tct---Tc------TTC-T---TcTTCt-TC---TTCT
111 gCgcCaaaaTgCagCagTaTgTaaaaaaTgaCggTaaTaaaTaTaTaaTaCaaaTgCaTagaggaaaggCTaTagaaaTgaagagaTTCaTgaaTtTTCagTCaggTTCT

 42 tCttCTTcTTc-TT-CT-T-CttCTtCTtCTtCTTCTTcTTcTTCtTCTTCTTC-TTCTTCTTCTTcTTCTTCtTCTTCtT-CTTCTTCTTcTTCTTCtTCTTC-TTCTT
221 cCgaCTT-TTaaTTgCTgTgCccCTcCTcCTgCTTCTT-TTtTTCcTCTTCTTCgTTCTTCTTCTT-TTCTTCcTCTTCcTgCTTCTTCTT-TTCTTCaTCTTCaTTCTT
                                         ----------------------------------------------------------------------------

 76 CTTcTTcTTCtTCTTCtT-CTTCTTcTTcTTCtTCTTCTTC-TTCTTCTTCTTcTTCTTCtTCTTC-TTCTTCTTcTTcTTCTTCtTCTTC-TTCTTCtTCTTcTTCTtC
327 CTTgTTtTTC-TCTTCcTgCTTCTT-TTtTTCcTCTTCTTCgTTCTTCTTCTT-TTCTTCaTCTTCaTTCTTCTTaTT-TTCTTCaTCTTCaTTCTTCaTCTT-TTCTcC
    ----------------------------------------------------------------------------------------------------------------

112 TTCTTCtT-CTTCTTCTTCtTCTTCTTCTTC-TTCTTCTTcTTCT--TCTTCtTCtT-CTTCTTCTTcTTCTTCTTCTTC-TTCTTcTTCTTCtTCTTCTTCTTC-TTCT
432 TTCTTCcTgCTTCTTCTTC-TCTTCTTCTTCcTTCTTCTTtTTCTccTCTTCcTCcTgCTTCTTCTT-TTCTTCTTCTTCcTTCTTtTTCTTC-TCTTCTTCTTCcTTCT
    ----------------------------------------------------------------------------------------------------------------

146 TcTTCTTCtTCTTCtTCTTC-TTCTTcTTCTTcTTCTTCTTCTTC-TTCTTCTTCTTcTTCtTCTTCTTC-TTCTTCTTCTTCT--TCTTCtTC-TTCTTCTTCTTCTtC
539 TtTTCTTC-TCTTCcTCTTCcTTCTTtTTCTT-TTCTTCTTCTTCcTTCTTCTTCTT-TTCcTCTTCTTCcTTCTTCTTCTTCTccTCTTCcTCcTTCTTCTTCTTCTcC
    ----------------------------------------------------------------------------------------------------------------

181 tTCttCTT-CTT-CTTCt--T--CTT--CtT---C-T--TCtTC-T-T-Ct-Tc-T----TC-TTcTt---CtTcT---T-C--TT-CT-T---CtTCt--Tc---TT--
646 cTCcgCTTtCTTtCTTCaagTaaCTTggCaTaagCaTagTCcTCgTaTaCaaTaaTagggTCcTTaTcaaaCaTtTgaaTaCccTTtCTgTtaaCgTCggaTaaagTTgg

202 Ct--------TCTt--Ct-TcT---TcT---TC----------TT-CTtCT----TcT---TcT--TcT----TCttC----TT--CTTCttCt-TcTT--Ct----TcT
756 CgcaaaaaggTCTaaaCcaTaTgaaTtTaaaTCcaaaaaggagTTgCTaCTaaagTtTaaaTtTgaTtTaaaaTCcaCaggaTTtaCTTCaaCagTaTTagCaggagTaT

221 --TCTTCT-TcTTcT--TCTtc--TTcTTC---TT-CtTC-----T--TcTTct----Tc-----TTC-T-TcTT-Ctt-CTT-Ctt-Ct--TCt-Tc-TTcTTCtt-Ct
866 aaTCTTCTgTgTTaTcaTCTagagTTgTTCcagTTtCaTCaaaagTaaTgTTgaaaagTgaaaaaTTCcTgTaTTtCaaaCTTtCggaCagaTCaaTaaTTtTTCcgaCa

245 TcTTCTTCtT---Ct-TCTTctTc-
976 TaTTCTTCaTgaaCaaTCTTggTaa
```

FIG. 4.12 - *Alignement optimal de la séquence de la figure 4.11 avec* TTC^{∞}

la transformation T est une succession de zones de matchs consécutifs séparées les unes des autres par des transformations élémentaires : $T = \mathtt{MM}\ldots\mathtt{M}t_1\mathtt{MM}\ldots\mathtt{M}t_2\ldots\mathtt{MM}\ldots\mathtt{M}t_3\ldots$, avec $t_1, t_2, t_3, \ldots \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}, \mathtt{D}\}$. Les transformations élémentaires sont codées de la gauche vers la droite.

Une suite $\mathtt{M}^e$, avec $e \geq 0$ est appelée un *saut* de e bases. Elle est codée par $Fibo(e)$. Puisque l'on sait qu'une transformation élémentaire suit toujours un saut, il n'est pas nécessaire de signaler à quel type de mot code nous avons affaire. La seule exception est le premier mot code de C_T : il faut spécifier si le codage commence par un saut ou par une transformation élémentaire. Le codage C_T débute donc par le bit *Indicateur de Type (IT)* qui spécifie si le premier mot code correspond à un saut (valeur 0) ou à une transformation élémentaire (valeur 1). Viennent ensuite la succession de mots code de sauts (de type $Fibo(e)$), et de transformations élémentaires.

Le codage d'une transformation élémentaire $t_h \in \{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}, \mathtt{D}\}$ requiert un peu de réflexion. Il n'y a aucune raison, a priori, de favoriser le codage de l'une ou l'autre des transformations. Nous allons donc utiliser un codage en longueur fixe. Le nombre de bits à utiliser est $4 = \lceil \log 9 \rceil$ car $\#\{\mathtt{S_A}, \mathtt{S_C}, \mathtt{S_G}, \mathtt{S_T}, \mathtt{I_A}, \mathtt{I_C}, \mathtt{I_G}, \mathtt{I_T}, \mathtt{D}\} = 9$. On remarque intuitivement que ce codage n'est pas efficace car $\log 9$ est plus proche de 3 que de 4. Il est possible de réduire le nombre de transformations élémentaires de 9 à 7 grâce aux deux réflexions suivantes :

1. **Seulement trois substitutions doivent être considérées** : la transformation optimale ne contient jamais la substitution d'une base par la même base. Le coût du match est en effet moins élevé que le coût d'une substitution. Dès lors, on considère toujours la substitution d'une base par une des trois autres. Il n'y a donc que trois types de substitutions.

2. **Le nombre d'insertions peut être réduit à trois par une légère manipulation de la transformation optimale.** On peut en effet toujours supposer qu'une transformation optimale ne contient jamais l'insertion de la même base que la base courante de M^∞. Supposons que ce soit le cas. Prenons par exemple l'insertion d'une base T. Cela signifie, dans l'alignement optimal, que l'on se trouve en présence de la situation suivante :

$$\begin{array}{lcl} S & = & \ldots\mathtt{T}\ldots\mathtt{X}\ldots \\ M^\infty & = & \ldots\text{----}\mathtt{T}\ldots \end{array}$$

avec $\mathtt{X} \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, -\}$

Il est évident que $\mathtt{X} = \mathtt{T}$, autrement la transformation n'est pas optimale. Dès lors, on effectue la modification suivante :

$$\begin{array}{lcl} S & = & \ldots\mathtt{T}\ldots\mathtt{T}\ldots \\ M^\infty & = & \ldots\text{----}\mathtt{T}\ldots \end{array} \quad \Longrightarrow \quad \begin{array}{lcl} S & = & \ldots\mathtt{T}\ldots\mathtt{T}\ldots \\ M^\infty & = & \ldots\mathtt{T}\text{----}\ldots \end{array}$$

La nouvelle transformation est également optimale mais le T inséré n'est pas la base courante de M^∞ (si le cas se représente, il suffit de répéter le processus).

Dès lors, on suppose que l'insertion porte toujours sur une des trois autres bases et donc qu'il n'existe que trois types d'insertions.

Nous sommes donc en présence de sept types de transformations élémentaires à coder: trois substitutions, trois insertions et une délétion. Chaque transformation élémentaire peut donc être codée sur exactement trois bits. La figure 4.13 représente les mots code pour les trois types de substitutions. La figure 4.14 représente les mots code pour les trois types d'insertions.

Base de M^∞ \ Base de S	A	C	G	T
A		000	001	010
C	010		000	001
G	001	010		000
T	000	001	010	

FIG. 4.13 - *Mots code des substitutions*

Base de M^∞ \ Base de S	A	C	G	T
A		011	100	101
C	101		011	100
G	100	101		011
T	011	100	101	

FIG. 4.14 - *Mots code des insertions*

Le mot code correspondant à la délétion est 110.

Un des mots code de longueur 3 n'est pas utilisé, il s'agit de 111. Il est utilisé comme **annonce de rupture** dans l'étape suivante d'optimisation du codage par liftings: $a_{\mathcal{R}} = 111$.

4.5.4 Courbe de compression

Le schéma de codage étant défini, nous présentons la courbe de compression d'une séquence $S \in \mathcal{N}^n$ relativement au motif $M \in \mathcal{N}^p$.

Définition 4.8 *La* courbe de compression *de la séquence $S \in \mathcal{N}^n$ relativement au motif $M \in \mathcal{N}^p$ est l'application partielle $f \in \mathcal{F}^n$ définie par:*

- $f(0) = -|C_M.C_I.IT|$ *: c'est le nombre de bits codés avant de débuter le codage à proprement parler de T. D'après les définitions de C_M, C_I et IT, $f(0) = -(|Fibo(p-1)| + 2p + 4 + 1)$.*

- $f(n) = 2n - |C|$ *: c'est le gain en compression de la séquence S. Le terme $2n$ est le nombre de bits économisés à ne pas utiliser NUC^* pour coder S, le terme $|C|$ est le nombre de bits "perdus" à coder la transformation optimale T.*

- $f(i) = 2i - |C(i)|$, *avec $0 < i < n$: c'est le gain partiel en compression produit par le codage du préfixe $T_{..j}$ de la transformation optimale qui construit le préfixe $S_{..i}$ de S. Le terme $2i$ est le nombre de bits économisés à ne pas utiliser NUC^* pour coder $S_{..i}$, le terme $|C(i)|$ est le nombre de bits "perdus" à coder le préfixe $T_{..j}$ de la transformation optimale T.*

Il est important de remarquer que $f(i)$ n'est pas défini pour toute position i et ceci pour les trois raisons suivantes:

1. Le codage $Fibo(e)$ d'un saut MM...M = M^e de T ne forme qu'un seul mot code que l'on ne peut pas séparer en plusieurs entités. Il n'est donc pas possible de définir un gain partiel en compression pour un préfixe de S qui correspond, dans T, à une position interne d'un saut.

   Les valeurs $f(i)$ ne sont donc pas définies pour ces positions.

2. L'algorithme TURBOOPTLIFT, que nous utilisons par la suite, considère que tout point défini de l'application partielle f est un début potentiel de rupture. Or, le schéma de codage autorise le codage de l'annonce de rupture $\alpha_{\mathcal{R}}$ uniquement aux positions i qui correspondent à des transformations élémentaires dans T.

   Les valeurs $f(i)$ ne sont donc définies que pour les positions i qui correspondent, dans T, à des transformations élémentaires.

   Remarque 4.7 *Cette situation n'est pas gênante car, d'après le schéma de codage, tout match induit un gain en compression et toute autre transformation élémentaire induit une perte. Dès lors, il n'y a aucun intérêt à débuter une rupture en début d'un saut (excepté pour un saut nul, de mot code Fibo(0), qui doit obligatoirement séparer deux transformations élémentaires consécutives. De toutes façon, dans ce cas la rupture a tout intérêt à débuter à la position de début de la première des deux transformations élémentaires).*

3. La situation est ambiguë pour les délétions. Soit $T_{..j} = T_{..j-1}\mathtt{D}$ un préfixe de la transformation optimale qui se termine par une délétion. Soit $S_{..i}$ le préfixe de S qu'il construit. Par définition de la délétion (voir tableau 4.5, page 188) le préfixe de transformation $T_{..j-1}$ construit le même préfixe $S_{..i}$ de S. Dès lors, il n'est pas possible de donner une valeur unique au gain partiel pour le préfixe $S_{..i}$.

   La fonction $f(i)$ n'est donc pas définie aux positions qui correspondent à des délétions.

Remarque 4.8 *Dans le premier chapitre de cette partie, nous avions défini la courbe de compression pour chacune des positions d'une séquence* ***binaire****. Ici, nous nous trouvons en présence d'une séquence de* ***nucléotides*** *mais nous ne prenons pas la peine de la convertir en binaire pour définir la courbe de compression. Une telle conversion aurait doublé le nombre d'abscisses à considérer dans l'algorithme* TURBOOPTLIFT *alors que de toute façon, seules les positions paires peuvent être des débuts et des fins potentiels de ruptures.*

Exemple 4.22 *Reprenons la séquence S et le motif M = TTC de l'exemple 4.21. Puisque $|M| = p = 3$, $f(0) = -(|Fibo(2)| + 6 + 5) = -15$. La courbe de compression f qui correspond à la transformation optimale de la figure 4.12 est représentée à la figure 4.15. On voit clairement apparaître sur ce graphique deux segments décroissants et un segment croissant. Le segment croissant correspond au facteur souligné de la figure 4.11. C'est le seul facteur de S qui ressemble vraiment à une RTE de M : il apporte un gain substantiel. Les segments décroissants correspondent à des zones qui ne ressemblent absolument pas à des RTE de M : ils apportent des pertes considérables. Le gain global en compression est de -470 bits !*

4.5.5 Optimisation par liftings

Nous commençons par présenter le détail du codage d'une rupture, nous poursuivons avec la détermination de la courbe de rupture $\mathcal{R}$ et terminons par un exemple d'utilisation.

D'après la courbe de compression représentée à la figure 4.15, il est clair que l'optimisation par liftings peut améliorer le gain de compression global. Elle permet également de localiser, de manière optimale, les facteurs de la transformation optimale T qui produisent une réelle compression et ceux qui produisent une perte. Les premiers construisent, dans S, des facteurs

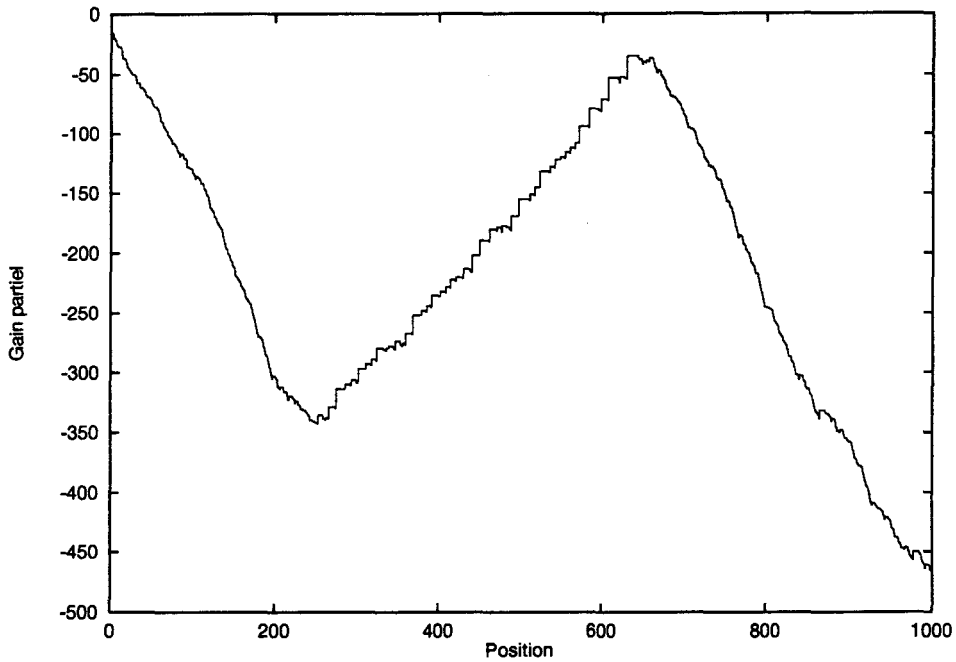


FIG. 4.15 - *Courbe de compression correspondant à la transformation optimale de la figure 4.12*

que l'on qualifie de VRTA (vraies zones RTA) et les derniers construisent, dans S, des facteurs que l'on qualifie de FRTA (fausses zones RTA).

La courbe de compression $f \in \mathcal{F}^n$ définie dans la section précédente vérifie toutes les propriétés des applications partielles qui sont optimisées par l'algorithme TURBOOPTLIFT. Pour l'appliquer, il nous faut une courbe de rupture DCL $\mathcal{R}$, de retard $r_{\mathcal{R}}$, et une annonce de rupture $a_{\mathcal{R}}$. Nous connaissons déjà l'annonce de rupture: le schéma de codage permet d'utiliser $a_{\mathcal{R}} = 111$. Le retard $r_{\mathcal{R}} = 2$ car on considère qu'une rupture doit porter sur au moins deux bases. Le choix est arbitraire mais le coût d'initialisation d'une rupture est tellement élevé qu'il est moins coûteux de coder une transformation élémentaire plutôt qu'une rupture de longueur 1. Ce paramètre a peu d'influence sur le déroulement de l'optimisation.

Rappelons que le but d'une rupture est de remplacer le codage défavorable d'un facteur de la séquence par le codage tel quel du facteur, précédé de sa longueur. Ici, la longueur se compte en nombre de bases, le codage tel quel d'un facteur est réalisé à l'aide de NUC^*.

Un problème épineux se pose: Soit $T = X.Y.Z$ une factorisation de la transformation optimale et $S = U.V.W$ une factorisation de la séquence S tels que:

- Le codage de Y est défavorable: il induit une perte en compression.
- La position du facteur V et la position du facteur W sont toutes deux définies dans la courbe de compression (c'est-à-dire Y peut être remplacé par une rupture).
- Le préfixe de transformation X construit le préfixe U de S.
- Le préfixe de transformation $X.Y$ construit le préfixe $U.V$ de S.

Alors, la seule connaissance du suffixe Z de T n'est pas suffisante pour construire le suffixe W de S. Il est en effet indispensable, pour appliquer correctement la suite de transformations élémentaires qui composent Z, de **connaître la phase du préfixe de transformation** $X.Y$, c'est-à-dire de connaître l'indice du caractère de M correspondant à la première transformation élémentaire de Z.

Dès lors, si l'on désire remplacer le codage de Y par le codage du facteur V précédé de sa longueur, alors la phase k du préfixe de transformation $X.Y$ doit être codée également. La phase k est codée sur exactement $\lceil \log p \rceil$ bits avec p la longueur du motif.

Le codage de la rupture est donc composé des quatre éléments suivants :

1. $a_{\mathcal{R}} = 111$: l'annonce de rupture.
2. La phase k du préfixe de transformation $X.Y$ codée sur $\lceil \log p \rceil$ bits.
3. Le codage auto-délimité de la longueur $|V|$ de manière relative au retard $r_{\mathcal{R}}$. Nous utilisons le code auto-délimité *Fibo* : $Fibo(|V| - r_{\mathcal{R}})$.

   Il s'agit d'un exemple, mais rappelons nous que le code doit être auto-délimité **et** ICL pour que la fonction de rupture $\mathcal{R}$ soit DCL. C'est le cas des codes *Fibo* et *PrefFibo* développés dans la partie I qui permettent de coder des entiers.
4. Le codage du facteur V : $NUC^*(V)$.

Le codage du facteur V n'induit ni perte ni gain car c'est la référence utilisée pour définir le gain. Par contre, chacun des trois autres éléments induisent une perte de compression. La courbe de rupture évalue cette perte en compression en fonction de la longueur l de la rupture.

La courbe de rupture $\mathcal{R} : \mathbb{N} \to \mathbb{Z}$ de retard $r_{\mathcal{R}}$ est définie par :

$$\begin{cases} \mathcal{R}(0) &= 0 \\ \mathcal{R}(l) &= -(|a_{\mathcal{R}}| + \lceil \log p \rceil + |Fibo(0)|) = -(5 + \lceil \log p \rceil), \forall l : 0 < l < r_{\mathcal{R}} \\ \mathcal{R}(l) &= -(|a_{\mathcal{R}}| + \lceil \log p \rceil + |Fibo(l - r_{\mathcal{R}})|) = -(3 + \lceil \log p \rceil + |Fibo(l - r_{\mathcal{R}})|), \forall l \geq r_{\mathcal{R}} \end{cases}$$

La fonction $\mathcal{R}$ est DCL car *Fibo* est ICL.

Toutes les conditions sont maintenant réunies pour optimiser le codage à l'aide de l'algorithme TURBOOPTLIFT: nous disposons d'une application partielle $f \in \mathcal{F}^n$ et d'une courbe de rupture DCL $\mathcal{R}$ de retard R. L'optimisation du codage à l'aide de TURBOOPTLIFT est totalement indépendante du motif M. Elle utilise uniquement la courbe de compression et la fonction de rupture. La courbe optimale, irréductible et minimale en ruptures parmi les courbes optimales irréductibles est calculée en temps $O(n \log n)$, quel que soit le motif M.

Exemple 4.23 *Prenons une fois de plus la séquence S et le motif M = TTC de l'exemple 4.15. Puisque $p = 3$, le codage de la phase est effectué sur 2 bits. L'application de* TURBOOPTLIFT *à la courbe f de la figure 4.15 fournit la courbe optimale $f^* \in Lift^*_{\mathcal{R}}(f)$. Elle est représentée à la figure 4.16. Le gain de la courbe optimisée est de 256 bits alors que la perte avant optimisation était de 470 bits. L'optimisation a donc produit une amélioration du gain de 726 bits.*

Deux ruptures ont été appliquées par l'algorithme. Chacune d'elles correspond à un facteur qualifié de FRTA. L'unique facteur correspondant à une portion d'application de f^* est qualifié de VRTA : le gain en compression qu'il produit est ***significatif à l'échelle de la taille de la séquence.***

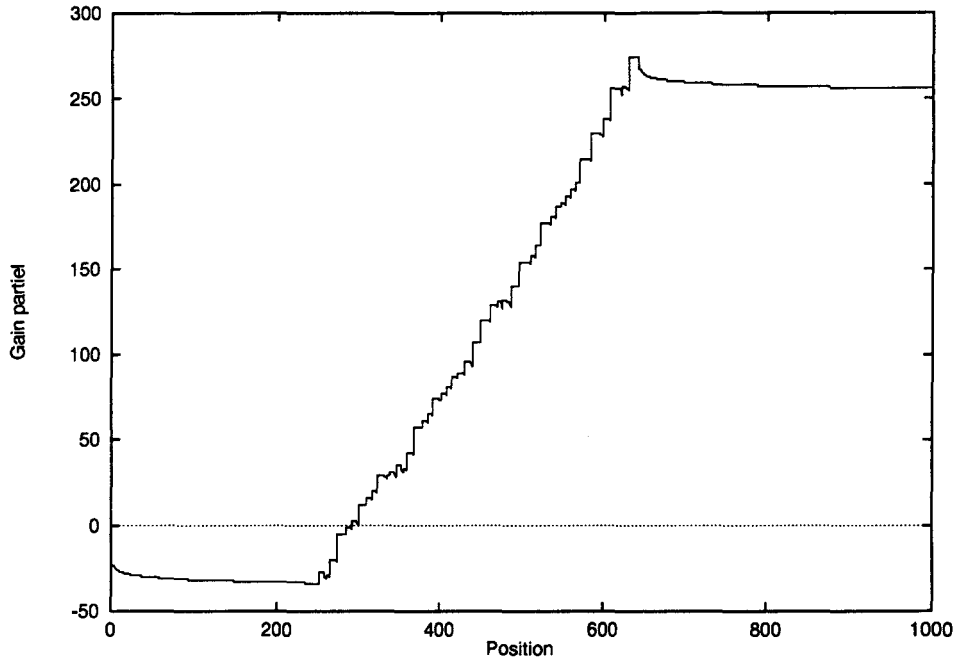


FIG. 4.16 - *Courbe de la figure 4.15 optimisée*

4.5.6 Implémentation et performances

Nous présentons ici quelques exemples d'application de notre méthode à la recherche de répétitions en tandem approximatives. Nous nous contentons de donner quelques exemples marquants, les qualités et les défauts de la méthode seront plutôt discutées dans la section 4.5.7. Ensuite, nous attirons l'attention sur un point délicat de l'implémentation de la méthode: le calcul des valeurs de la fonction de rupture.

4.5.6.1 Quelques exemples d'exécution

L'algorithme EXACTEMENTRTA, qui enchaîne la phase d'alignement, de codage et d'optimisation, a été implémenté en C sur un SUN Sparcstation 20 équipé de $96MB$ de mémoire. Nous présentons quelques exemples d'utilisation de EXACTEMENTRTA. Ici, la méthode utilisée pour manipuler les valeurs de la fonction de rupture est la méthode 3 décrite dans la section 4.5.6.2.

Exemple 4.24

Les exemples précédents traitent tous du même segment de 1000 bases du chromosome 11 de la levure et du motif M = TTC. Replaçons maintenant ce segment dans son contexte, c'est-à-dire considérons l'entièreté du chromosome 11 (666448 bases) et essayons de retrouver la zone VRTA précédemment détectée.

Pour le chromosome 11 et le motif M = TTC, les temps d'exécution sont:

8 secondes pour la phase d'**alignement**.
12 secondes pour la phase d'**optimisation**!

La figure 4.17 représente la courbe résultant de la phase de codage. La perte en compression est de 856843 bits. À l'œil nu, aucune régularité n'est détectée, elles sont toutes "écrasées"

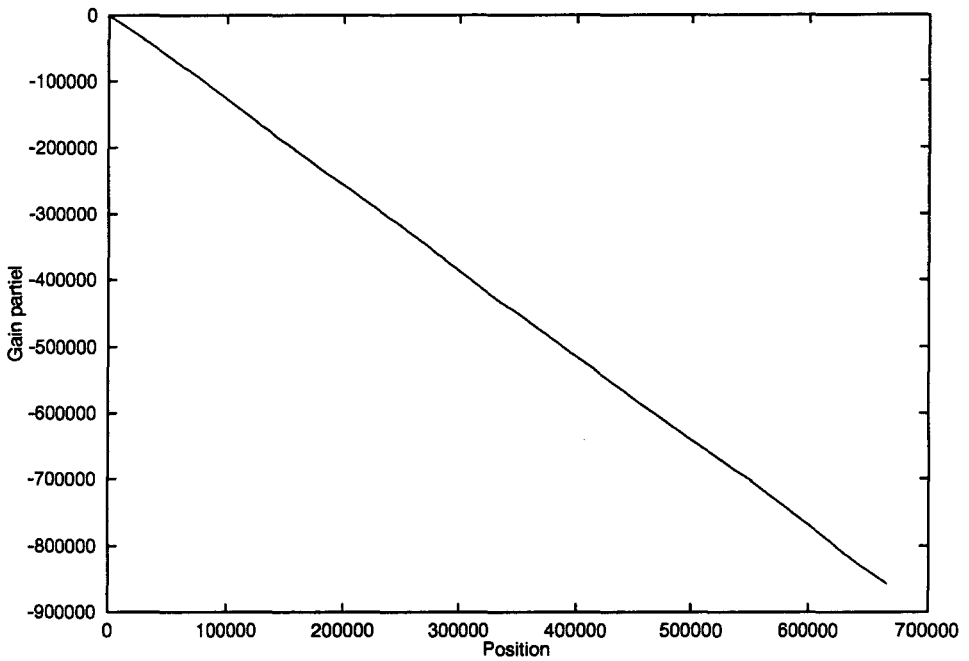


FIG. 4.17 - *Chromosome 11, motif TTC - avant lifting*

par l'échelle du graphique.

La figure 4.18 représente la même courbe optimisée par liftings. **Attention**! L'échelle de ce graphique est complètement différente de l'échelle du graphique de la figure 4.17. Le nouveau gain en compression est de 233 bits. Quatre ruptures ont été appliquées. Elles font ressortir les trois VRTA suivantes (sur le graphique, ce sont les trois segments qui paraissent verticaux à cause de l'échelle) :

Début : 63951, longueur : 392, gain produit : 308 : Il s'agit **exactement** du même segment que celui qui avait été détecté dans la fenêtre de 1000 bases.

```
       CTTCTTcTTcTTCtTCTTCTTC-TTCTTCTTCTTcTTCTTCtTCTTCtT-CTTCTTCTTcTTCTTCtTCTTC-TTCTTCTTcTTcTTCtTCTTCtT-CTTCTTcTTcTTC
63951  CTTCTT-TTtTTCcTCTTCTTCgTTCTTCTTCTT-TTCTTCcTCTTCcTgCTTCTTCTT-TTCTTCaTCTTCaTTCTTCTTgTTtTTC-TCTTCcTgCTTCTT-TTtTTC

       tTCTTCTTC-TTCTTCTTCTTcTTCTTCtTCTTC-TTCTTCTTcTTcTTCTTCtTCTTC-TTCTTCtTCTTcTTCTTCtCTTCTTCtT-CTTCTTCTTCtTCTTCTTCTTC-
64056  cTCTTCTTCgTTCTTCTTCTT-TTCTTCaTCTTCaTTCTTCTTaTT-TTCTTCaTCTTCaTTCTTCaTCTT-TTCTcCTTCTTCcTgCTTCTTCTTC-TCTTCTTCTTCc

       TTCTTCTTcTTCT--TCTTCtTCtT-CTTCTTCTTcTTCTTCTTCTTC-TTCTTcTTCTTCtTCTTCTTCTTC-TTCTTcTTCTTCtTCTTCtTCTTC-TTCTTcTTCTT
64162  TTCTTCTTtTTCTccTCTTCcTCcTgCTTCTTCTT-TTCTTCTTCTTCcTTCTTtTTCTTC-TCTTCTTCTTCcTTCTTcTTCTTC-TCTTCcTCTTCcTTCTTtTTCTT

       cTTCTTCTTCTTC-TTCTTCTTCTTcTTCtTCTTCTTC-TTCTTCTTCTT--TCTTCtTC-TTCTTCTTCTTCT
64269  -TTCTTCTTCTTCcTTCTTCTTCTT-TTCcTCTTCTTCcTTCTTCTTCTTCTccTCTTCcTCcTTCTTCTTCTTCT
```

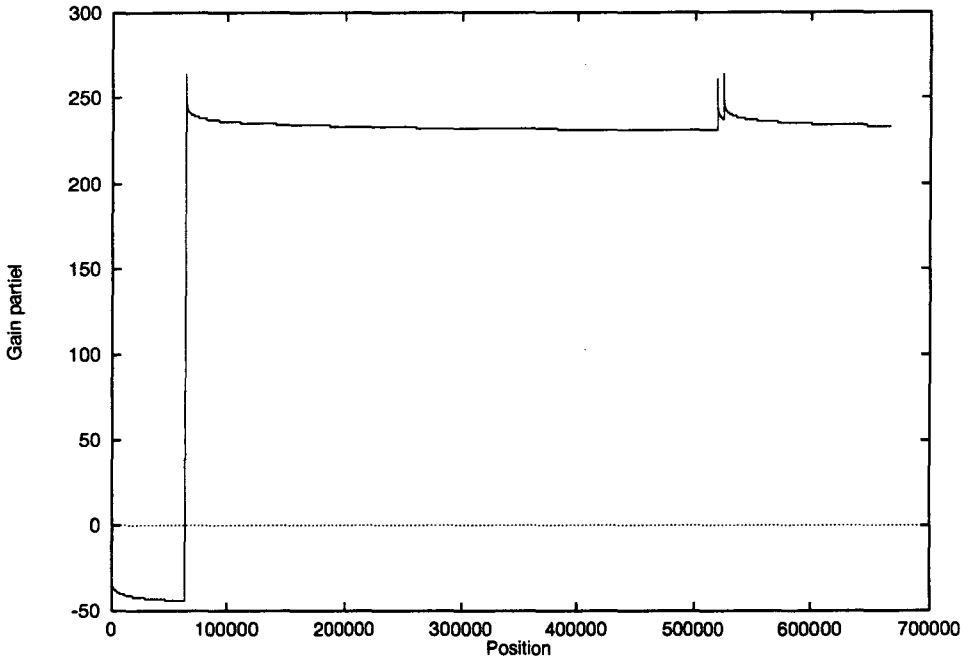


FIG. 4.18 - *Chromosome 11, motif TTC - après lifting*

Début : 519431, longueur : 45, gain : 30 :

```
        TCTTCTTCtTCtTCTTCtTCTTCtTCtTCtTCTTCtTCTTCTTCtTCT
519431  TCTTCTTCcTCcTCTTCcTCTTCcTCcTCTTCcTCTTCTTCcTCT
```

Début : 525127, longueur : 38, gain : 27 :

```
        TCTTCTTCTTCtTCTTCtTCtTCtTCttCTTCTTCTTC
525127  TCTTCTTCTTCgTCTTCgTCcTCgTCcgCTTCTTCTTC
```

La méthode a été capable de localiser rapidement et avec précision, dans l'entièreté du chromosome, un facteur régulier que l'on avait localisé dans une fenêtre relativement courte. Cela traduit le fait que EXACTEMENTRTA semble capable de localiser avec précision des régularités significatives à l'échelle du chromosome.

Exemple 4.25

Continuons avec le chromosome 11, mais cette fois en prenant le motif $M = \mathtt{T}$.

Pour un motif aussi court, même une séquence aléatoire de la longueur du chromosome 11 possède beaucoup de matchs. Notre méthode permet de sélectionner précisémment les facteurs pour lesquels la régularité est suffisamment forte.

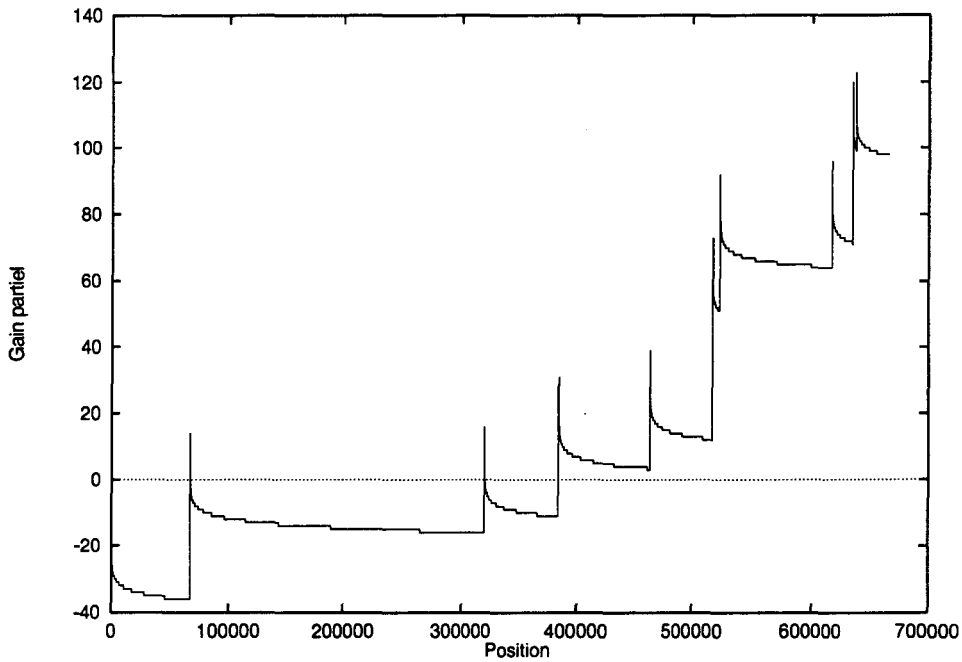


FIG. 4.19 - *Chromosome 11, motif T - après lifting*

La courbe de compression avant lifting ressemble à s'y méprendre à celle de la figure 4.17, nous ne la montrons pas. Par contre, la courbe optimisée (figure 4.19) met bien en évidence les 11 VRTA détectées. Le gain de la courbe optimale est 98.

La zone détectée la plus significative, au niveau du gain, est :

Début : 67763, longueur : 37, gain : 50 :

```
        TTTTT-TTTTTT-TTTTTTTTTTTTTTTTTTTTTTT
67763   TTTTTcTTTTTTcTTTTTTTTTTTTTTTTTTTTTTT
```

La zone détectée la moins significative est :

Début : 385229, longueur : 19, gain : 19 :

```
        TTTTTTTTTT--TTTTTT
385229  TTTTTTTTTTggTTTTTT
```

Exemple 4.26

Nous avons effectué une vaste recherche[5] de VRTA dans les 16 chromosomes de levure jusqu'aux motifs de longueur 5 mais rien, excepté ce qui a déjà été trouvé par Eric Rivals pour les petits motifs ([Riv96, RDD+97] et section 4.4), ne semble sortir du lot. Certains

5. Non exhaustive.

chromosomes possèdent des répétitions qui n'apportent qu'un gain de 2 ou 3 bits; elles ne sont donc pas significatives au sens de Milosavljević et Jurka [MJ93].

Le seul motif repéré est M = **ACACC** (et ses permutations cycliques) de longueur 5 qui possède des VRTA dans quelques chromosomes.

La courbe optimale, pour la recherche des VRTA de M = **ACACC** dans le chromosome 3 (315344 bases), est représentée à la figure 4.20. Le gain de la courbe optimale est 44. Une

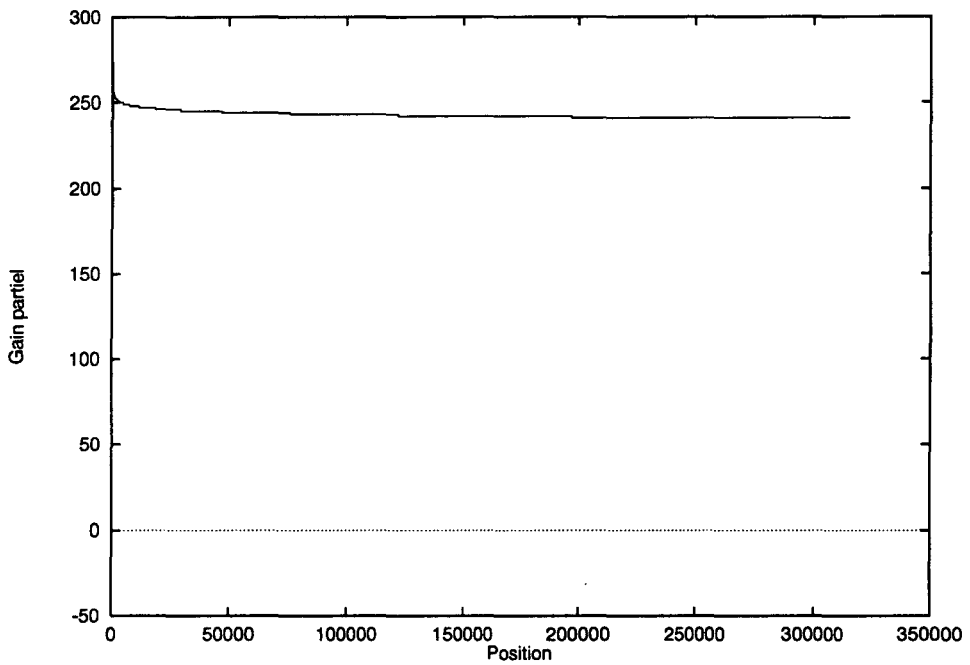


FIG. 4.20 - *Chromosome 3, motif ACACC - après lifting*

seule zone est répétée en tamdem :

Début : 1, longueur : 362, gain : 294 :

```
    aCaC-CACACCACACC-ACACCACACCaCAcCACACCACAcCACACCACACCaCAcCACACC-ACACCACACCACACC-ACACCACACCaCAcCACACC-ACACCaCAcC
  1 -CcCaCACACCACACCcACACCACACC-CA-CACACCACA-CACACCACACC-CA-CACACCcACACCACACCACACCcACACCACACC-CA-CACACCcACACC-CA-C

    ACACCACACCaCAC-CACACCACACCaCAC-CACACC-ACACCaCAcCACACCACACC-ACAC-CACACCACACCaCAC-CACACCACACCACACC-ACACCACACC-AC
101 ACACCACACC-CACaCACACCACACC-CACaCACACCcACACC-CA-CACACCACACCcACACaCACACCACACC-CACaCACACCACACCACACCcACACCACACCcAC

    ACCaCAcCACACCACACC-ACACC-ACACCaCAcCACaC-CACACCACACCaCAC-CACACCACACCaCAC-CACACC-ACACCACACCaCAcCACACCACACCaCAcCA
206 ACC-CA-CACACCACACCcACACCcACACCcCA-CACcCaCACACCACACC-CACaCACACCACACC-CACaCACACCcACACCACACC-CA-CACACCACACC-CA-CA

    CACC-ACACCaCAC-CACACCACACC-ACACCACACC-ACACCaCAcCACACC-ACACC
307 CACCcACACC-CACaCACACCACACCcACACCACACCcACACC-CA-CACACCcACACC
```

Nous nous trouvons en présence d'une région télomérique : la VRTA se situe au début du chromosome.

Exemple 4.27

Notre méthode détecte la répétition en tandem du tri-nucléotide M = CGG dans le gène FMR-1 (3765 bases) impliqué dans la maladie génétique humaine du retard mental "X-fragile" [VPS+91].

Le gène FMR-1 possède une VRTA de CGG :

Début : 1, longueur : 127, gain : 148 :

```
      --CGGcGGCGgC-G-GC--GGcGGCG-GCGGC-G-GCGGCGGCGGCGGCGGCGGCGGCGGCGGcGGCGGCGGCGGCGGCGGCGGCGGCGGcGGCGGCGGCGGCG
  1   gaCGGaGGCGcCcGtGCcaGGgGGCGtGCGGCaGcGCGGCGGCGGCGGCGGCGGCGGCGGCGGaGGCGGCGGCGGCGGCGGCGGCGGCGGaGGCGGCGGCGGCG

      GCGGCGGCGGCGGCGGC
111   GCGGCGGCGGCGGCGGC
```

4.5.6.2 Calcul des valeurs de la fonction de rupture $\mathcal{R}$

Jusqu'à présent, nous avons toujours supposé que la valeur d'une fonction de rupture se calculait en temps constant. En pratique, ce n'est pas toujours le cas. Cela dépend du code ICL utilisé pour auto-délimiter les longueurs des ruptures. Si la longueur de la représentation auto-délimitée d'un entier est calculable en temps constant, alors il n'y aucun problème, l'algorithme fournit le résultat optimal en temps $O(n\mathcal{R}(n))$.

Si la longueur de la représentation n'est pas calculable en temps constant, alors la complexité de l'algorithme va s'en ressentir. Les courbes de ruptures sont utilisées de manière intensive par l'algorithme TurboOptLift et les performances peuvent être moins bonnes si le calcul des valeurs de la courbe de rupture est trop coûteux.

Le problème peut être contourné en initialisant, avant l'optimisation par liftings, un tableau d'entiers $AD[0 \to n]$ destiné à contenir les longueurs des représentations auto-délimitées des entiers. Il est raisonnable de penser que ce tableau peut-être construit en temps $\theta(n)$ en parcourant les indices i de 0 à n. C'est le cas des deux codes *Fibo* et *PrefFibo*. Malheureusement, en pratique, un tel tableau est volumineux et donc lourd à manipuler par le système d'exploitation. Par exemple, la séquence du chromosome 4 de la levure contient 1522191 bases. Si chaque entier est codé sur 4 octets, le tableau nécessite 6.2 MB de mémoire !

En se basant sur le code *Fibo*, on envisage quatre manières pour calculer la longueur de la représentation auto-délimitée d'un entier i. Nous les comparons sur un cas concret après les avoir décrites.

Méthode 1 : On recalcule $|Fibo(i)|$ à chaque fois que l'on en a besoin en parcourant tous les nombres de Fibonacci inférieurs à i. Chaque calcul demande $O(\log i)$ étapes. La complexité de l'algorithme d'optimisation est alors en temps $O(n(\log n)^2)$.

La méthode ne nécessite aucune mémoire supplémentaire pour calculer $|Fibo(i)|$.

Méthode 2 : On recalcule $|Fibo(i)|$ à chaque fois que l'on en a besoin grâce à la formule : $F_j = \frac{\phi^j}{\sqrt{5}}$ arrondi à l'entier le plus proche, avec $\phi = \frac{1+\sqrt{5}}{2}$.

Cette méthode ne nécessite aucune mémoire supplémentaire pour calculer $|Fibo(i)|$. Chaque nombre de Fibonacci est calculé en temps constant. Malheureusement, l'exponentiation en nombre flottant est très coûteuse en temps.

Intervalle	Longueur de la représentation
$[0,0]$	2
$[1,1]$	3
$[2,3]$	4
$[4,7]$	5
$[8,12]$	6
...	...
$[..,n]$	$\|Fibo(n)\|$

FIG. 4.21 - *Intervalles des longueurs* $|Fibo(i)|$

Méthode 3 : On calcule le vecteur $AD[0 \rightarrow n]$ comme décrit ci-dessus. On accède à la valeur $|Fibo(i)|$ en temps constant mais cela nécessite le stockage d'un très gros tableau en mémoire.

Méthode 4 :

Plutôt qu'utiliser un vecteur de nombres entiers pour stocker la longueur de la représentation de chaque nombre, on regroupe tous les nombres dont la représentation a la même longueur en intervalles.

Exemple 4.28 *Pour le code Fibo, on définit les intervalles comme sur la figure 4.21.*

Puisque le code est ICL, les intervalles sont de plus en plus longs et la longueur de la représentation des nombres d'un intervalle est celle de l'intervalle précédent plus 1. Chaque intervalle est codé par ses deux bornes. Puisque les intervalles ne se chevauchent pas, ils sont ordonnés strictement. On peut alors les mémoriser dans un arbre binaire de recherche (voir figure 4.22). Chacun des nœuds correspond à un intervalle; on y mémorise les bornes de l'intervalle ainsi que la longueur des mots code correspondant. Toute la descendance d'un nœud par son fils de gauche concerne des longueurs de mots code plus petites que celle du nœud, toute la descendance par son fils de droite concerne des longueurs plus grandes. L'arbre possède de l'ordre de $O(\log n)$ nœuds. S'il est bien équilibré, la recherche de $|Fibo(i)|$ est réalisée en un temps $O(\log \log n)$. La complexité temporelle de l'algorithme d'optimisation est alors $O(n(\log n)(\log \log n))$.

Nous avons réalisé quatre implémentations de l'algorithme EXACTEMENTRTA : une pour chacune des quatre méthodes. Les différences de temps d'exécution sont assez frappantes.

La séquence test utilisée est le chromosome 2 de la levure (813137 bases) et le motif choisit est $M =$ `A` (c'est un petit motif, donc les courbes de ruptures sont souvent consultées, voir section 4.5.7). Le tableau 4.23 reprend les résultats d'exécution de l'algorithme EXACTEMENTRTA.

Bien que les courbes de ruptures ne soient manipulées que dans la dernière phase de EXACTEMENTRTA, les différences de temps d'exécution sont marquantes. Cela met d'autant mieux en évidence le soin qu'il faut accorder au choix de la méthode de calcul des valeurs des courbes de rupture. On remarque immédiatement l'inadéquation de la méthode 2 qui effectue trop de calculs sur des nombres flottants. Les méthodes 1 et 4 ont des temps comparables,

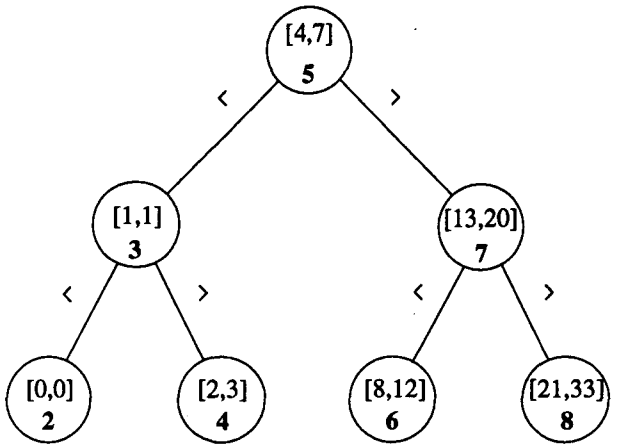


FIG. 4.22 - *Arbre équilibré de mémorisation des intervalles des mêmes longueurs* $|Fibo(i)|$

Méthode	Temps (en sec.)
1	636
2	3360
3	175
4	489

FIG. 4.23 - *Temps d'exécution de* EXACTEMENTRTA *sur le chromosome 2 de la levure avec le motif* $M = A$*, pour les quatre méthodes de calcul des longueurs des représentations auto-délimitées par Fibo*

ce qui n'est pas surprenant puisque leur complexité en temps est très proche. **La meilleure méthode est la méthode 3 qui calcule le vecteur de toutes les longueurs** $|Fibo(i)|$. Elle ne semble pas souffrir de l'utilisation abondante de la mémoire. En fait, la mémoire utilisée par l'étape d'optimisation n'est pas très importante par rapport à la première phase d'alignement. Si la longueur du motif est p, alors la phase d'alignement doit conserver en mémoire p transformations (chacune d'une phase différente).

4.5.7 Conclusions : points forts et points faibles de la méthode

La méthode EXACTEMENTRTA permet de rechercher exactement les RTA, c'est-à-dire toutes les variantes des répétitions en tandem exactes, tant qu'une séquence peut être considérée comme telle. En outre, la méthode n'est pas tributaire des effets de bords liés à des choix de fenêtre. Les paramètres de EXACTEMENTRTA apparaissent clairement:

- La forme de la fonction de rupture.
- La courbe de compression, qui elle-même dépend de:
  - Le schéma de codage utilisé.
  - La transformation optimale qui elle-même dépend de:
    - La séquence S.

- Le motif M.
- Le choix de la fonction de coût des transformations.

La méthode est "ré-itérable" : les zones de ruptures peuvent être concaténées et soumises à nouveau à l'algorithme avec un autre motif M'. De ce fait, il est possible de rechercher les répétitions en tandem de plusieurs motifs par "ré-itération".

EXACTEMENTRTA est un outil puissant de détection de RTA, dans la mesure ou les zones détectées ont une significativité d'un point de vue non seulement local mais également global. Ce n'est pas le cas lorsque l'on travaille par décomposition d'une séquence en fenêtres indépendantes. De plus, la décomposition en fenêtres peut ne pas détecter des RTA chevauchant deux fenêtres contiguës.

De plus, les limites des zones sont déterminées sans fixer de seuil. L'algorithme les fixe de manière optimale. Il y a un manque, en biologie moléculaire, de méthodes de ce type, capables de localiser précisémment le début et la fin des zones régulières (quels que soient les types de régularités) sans devoir fixer de seuil arbitraire.

L'algorithme est très rapide, sauf dans le cas où le motif est très court. Ce phénomène est dû au grand nombre de ruptures potentielles présentes à tout moment dans LRP. Comme le motif est petit, le nombre de matchs consécutifs est élevé, ce qui provoque une oscillation régulière et aléatoire de la valeur de la courbe de compression. Il en résulte, lors de l'optimisation, un grand nombre de ruptures potentielles qui se trouvent simultanément dans LRP (on sait malgré tout que ce nombre est majoré par $|Fibo(n)|$).

Lorsqu'une zone ne ressemble absolument pas à une RTE du motif, elle est très vite parcourue parce qu'elle correspond à un morceau de courbe qui décroît d'une façon régulière. De ce fait, très peu de nouvelles ruptures sont introduites dans LRP. Pendant le parcours de la zone, il y a une rupture qui se trouve dans LRP et qui domine toutes les autres.

Pour valider notre méthode, nous l'avons appliquée à des séquences engendrées aléatoirement[6]. Les séquences étaient de longueurs $500, 1000$ et 1000000. Tous les motifs ont été expérimentés jusqu'à la longueur 5. La majorité des expérimentations n'ont détecté aucune VRTA ! Dans quelques cas seulement, des zones de longueurs inférieures à 10 symboles ont été mises en évidence, dans ces cas les gains produits n'ont jamais dépassé 8 bits (le maximum était atteint pour une séquence de longueur 1000000 et un motif de longueur 2).

6. Selon une loi de distribution uniforme des nucléotides.

Chapitre 5

Localisation de longues répétitions approximatives

Nous présentons ici une esquisse de problème qui semble pouvoir être résolu à l'aide d'une adaptation de la méthode d'optimisation par liftings. Il s'agit d'une utilisation de la méthode des liftings un peu en marge de l'usage que l'on en a fait dans le reste du travail. Il n'est pas question, à proprement parler d'une optimisation globale, mais plutôt d'une délimitation "en-ligne" de facteurs réguliers. Il s'agit de localiser, grâce à la compression, de longues répétitions approximatives.

Dans ce chapitre, tout est basé sur la séquence HUMGHCSA que nous avons déjà évoquée dans le chapitre 3, à la page 160. Elle contient plusieurs longues répétitions approximatives, d'environ 20000 bases !

Les études que nous mentionnons très rapidement ont été menées en collaboration avec Eric Rivals il y a quelques années. Elles ont resurgi il y a quelques temps suite au développement de la méthode d'optimisation par liftings.

5.1 Détection du phénomène

La séquence HUMGHCSA, de 66495 bases, possède de nombreuses répétitions. L'algorithme BIOCOMPRESS-2 la comprime d'ailleurs de 34.63% (voir chapitre 3, section 3.4.2). Nous avons tenté de comprendre pourquoi.

Considérons le graphique qui représente le nombre de (`A` ou `T`) moins le nombre de (`C` ou `G`) rencontrés depuis le début de la séquence. Il est représenté, pour la séquence HUMGHCSA à la figure 5.1.

On voit clairement que des zones de plusieurs milliers de nucléotides se ressemblent. Il s'agit vraisemblablement de longs facteurs répétés approximativement. Nous allons essayer de comprimer la séquence en exploitant cette propriété.

5.2 Localisation précise des répétitions

Pour localiser précisément les limites des facteurs approximativement répétés, nous utilisons la technique du "dot plot". Le "dot plot" est un tableau de pixels dont les lignes et les colonnes sont indicées par des positions dans la séquence. À une ligne correspond un nucléotide de la séquence. À une colonne correspond également un nucléotide de la séquence.

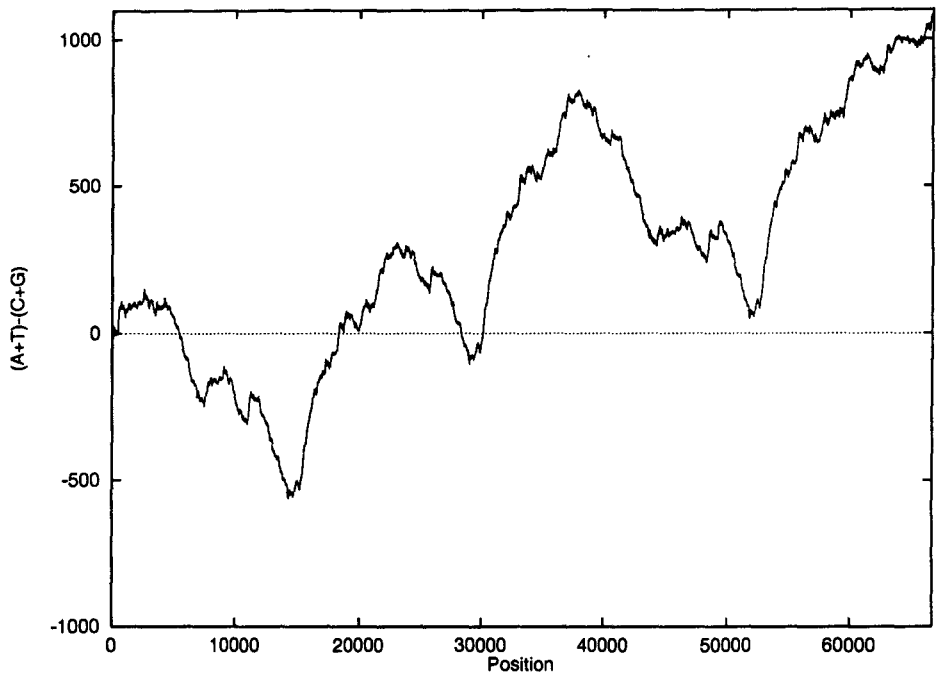


FIG. 5.1 - *Comptage du nombre de* $(A+T)-(C+G)$

Pour une ligne et une colonne données, si les deux nucléotides sont identiques, alors un point est affiché à l'intersection de la ligne et de la colonne. La figure 5.2[1] représente une partie du "dot plot" de la séquence HUMGHCSA. Bien entendu, une diagonale principale apparaît puisqu'un point est toujours affiché lorsque l'indice de la ligne est égal à l'indice de la colonne. Des segments continus, parallèles à la diagonale principale, désignent des répétitions exactes dans la séquence : sur plusieurs positions consécutives, les nucléotides qui correspondent à la ligne et à la colonne sont identiques.

Lorsque les segments sont parfois interrompus sur quelques nucléotides, il s'agit de répétitions approximatives. Nous avons localisé, à l'aide du "dot plot", les longues répétitions approximatives mises en évidence à la figure 5.1. Il s'agit d'un long travail qui a permis de dégager la structure de la séquence HUMGHCSA : deux facteurs, nommés respectivement B et D sont des copies approximatives des facteurs A et C. Les quatre segments ont une longueur approximative de 20000 bases.

La figure 5.3 présente les emplacements des différents facteurs dans la séquence.

5.3 Compression

Maintenant que nous avons localisé les facteurs approximativement répétés, nous sommes en mesure d'exploiter la propriété pour comprimer. Nous codons le facteur B (resp. D), à l'aide d'un codage économe de la liste des transformations élémentaires qui permettent de le construire à partir du facteur A (resp. C). La liste des transformations est trouvée grâce à un alignement par la méthode de programmation dynamique classique. On constate que le segment B a conservé un grand nombre de facteurs plus ou moins longs du segment A. Il s'agit

1. Le "dot plot" a été affiché grâce au programme du domaine public GDE.

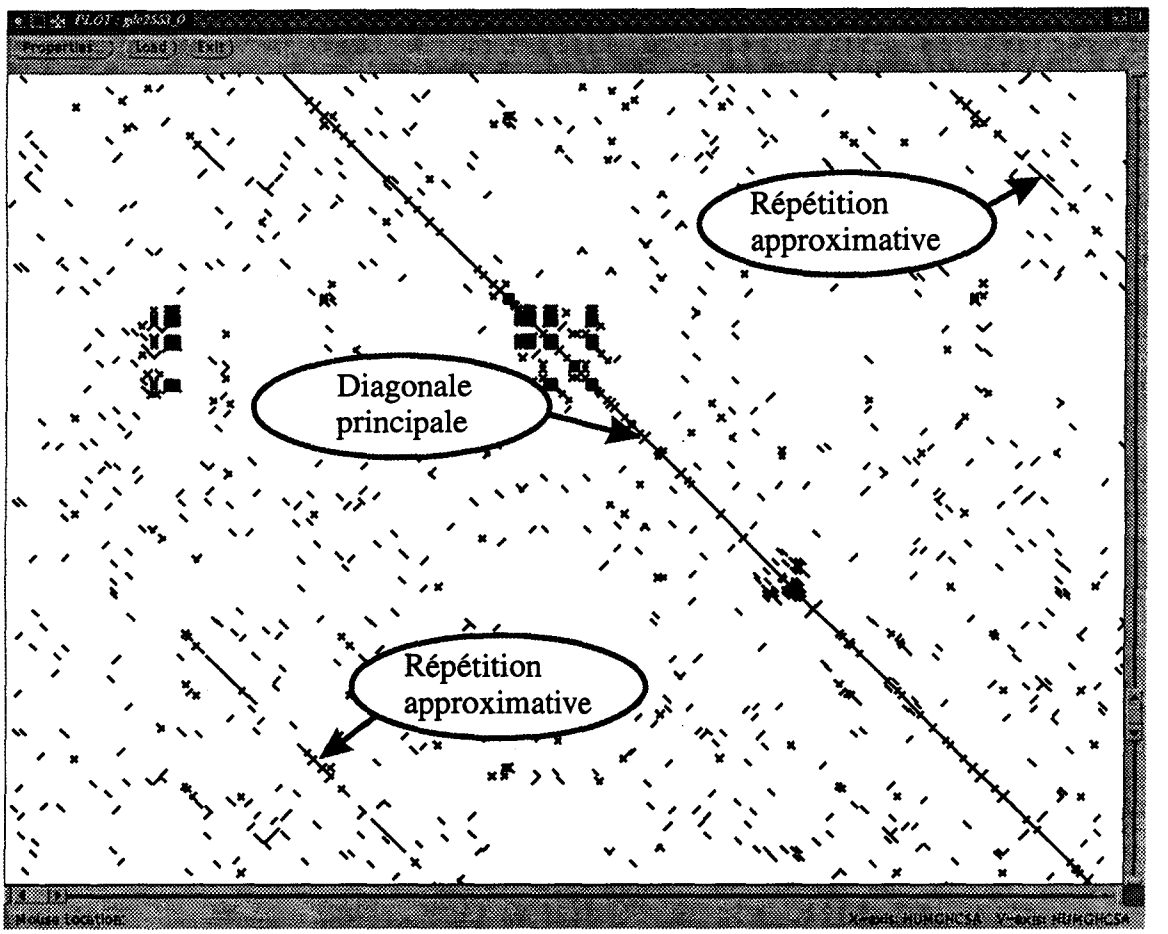


FIG. 5.2 - *"Dot plot" de la séquence HUMGHCSA*

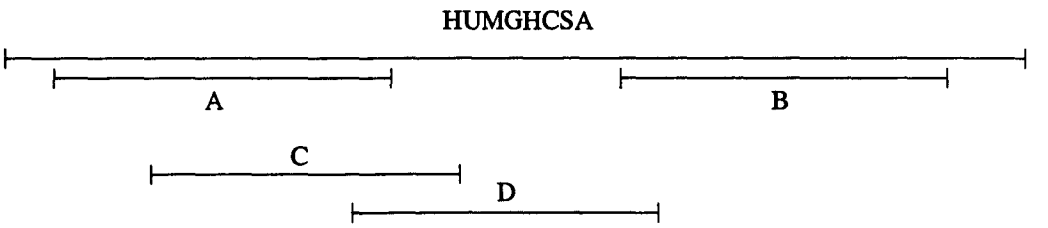


FIG. 5.3 - *Emplacements des facteurs approximativement répétés de HUMGHCSA*

vraisemblablement de deux copies d'un même fragment qui ont ensuite évolué séparément par mutations ponctuelles. Cette propriété est exploitée pour comprimer; les longs facteurs conservés sont favorisés.

Nous atteignons de cette façon un taux de compression de 40%, c'est-à-dire beaucoup plus que ce qui était produit par les compresseurs exploitant des répétitions exactes. Cela signifie que la propriété qui consiste à contenir de longues répétitions approximatives est bien adaptée à cette séquence.

5.4 Perspectives de recherche : localisation par liftings

Le phénomène doit se produire dans d'autres séquences; nous envisageons donc de développer une méthode de localisation de longues répétitions approximatives. Actuellement, nous avons développé une méthode, basée sur l'utilisation de l'arbre des suffixes, qui recherche des longues répétitions approximatives que l'on peut exploiter pour comprimer. Les résultats actuels ne nous satisfont pas (nous sommes loin des 40% attendus).

La démarche que nous avons suivie est de partir d'un long facteur répété de manière exacte, appelé *point d'ancrage*, et de tenter d'en étendre les deux occurrences vers la gauche et vers la droite par alignement. Nous sommes confrontés à un problème de seuil qui nous empêche d'atteindre les 40% attendus : où faut-il arrêter l'extension à gauche et l'extension à droite?

L'idée que nous avons de l'application de la méthode des liftings est la suivante : dans la méthode habituelle, la séquence est parcourue de la gauche vers la droite et les ruptures sont appliquées lorsque c'est possible. On part d'une courbe complète et le résultat est une courbe optimale. Dans le cas présent, si l'on reprend l'approche qui repose sur l'utilisation d'un point d'ancrage, et l'extension vers la droite et vers la gauche, on ne dispose pas d'une courbe complète mais seulement d'une portion de courbe de compression locale qui est construite au fur et à mesure. Pour l'instant, contentons-nous de considérer que l'on essaie uniquement d'étendre la zone vers la droite. À chaque fois que l'on progresse dans l'extension, un petit segment de courbe de compression vient s'ajouter à la portion que nous avions déjà. Si, à un moment donné, une rupture potentielle, dont l'origine appartient à la portion de courbe calculée lors de l'extension, réduit la courbe, alors il faut arrêter l'extension vers la droite; il existe en effet au moins une rupture qui donne une meilleure compression (voir figure 5.4). La position qui marque l'extrémité droite de la zone régulière est l'abscisse d'origine de la première rupture qui réduit la portion de courbe.

Il n'est donc pas question d'optimisation globale d'une courbe, mais simplement de la délimitation d'une zone à l'aide des liftings.

Pour l'extension vers la gauche, on "retourne" le problème : la zone est étendue de la droite vers la gauche, la forme à donner à la courbe de rupture est une rotation à 180 degrés de la forme habituelle. La rupture "réduit" la portion de courbe lorsqu'elle devient plus petite (voir figure 5.5).

La position qui marque l'extrémité gauche de la zone régulière est l'abscisse d'origine de la première rupture qui réduit la portion de courbe.

D'une façon générale, les liftings nous permettront de trouver l'étendue à gauche et l'étendue à droite d'une zone régulière dont on connaît une position centrale. La délimitation se fera "en-ligne".

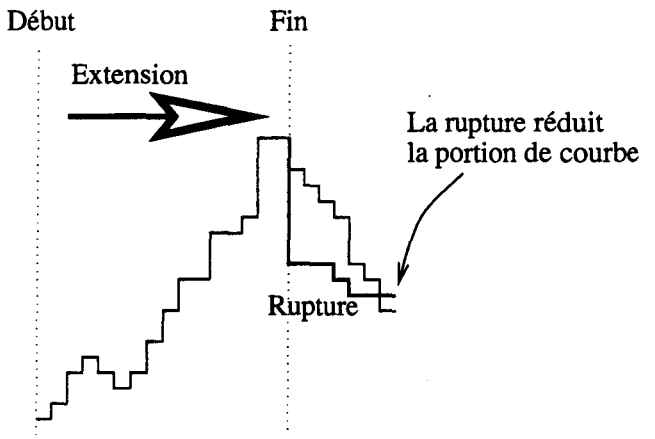


FIG. 5.4 - *Rupture appliquée lors de l'extension vers la droite*

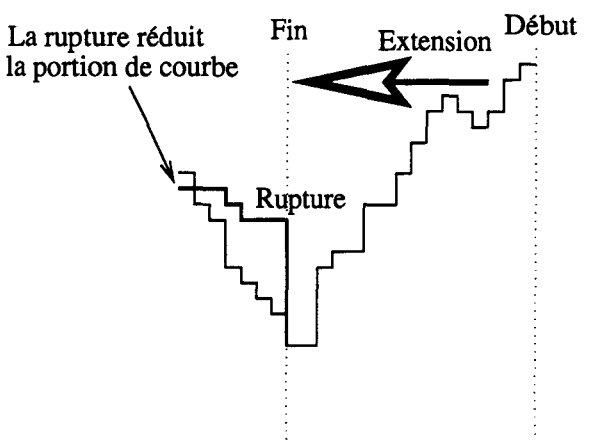


FIG. 5.5 - *Rupture appliquée lors de l'extension vers la gauche*

Bibliographie

[Abr63] ABRAMSON, N. – *Information Theory and Coding.* – New York, McGraw-Hill, 1963.

[AF87] APOSTOLICO, A., ET FRAENKEL, A. – Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inform. Theory*, vol. 33, n° 2, 1987, pp. 238–245.

[AGM+90] ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., ET LIPMAN, D. J. – A Basic Local Alignment Search Tool. *J. Mol. Biol.*, vol. 215, 1990, pp. 403–410.

[AIL+88] APOSTOLICO, A., ILIOPOULOS, C., LANDAU, G. M., SCHIEBER, B., ET VISHKIN, U. – Parallel construction of a suffix tree with applications. *Algorithmica*, vol. 3, 1988, pp. 347–365.

[ALS96] ANDERSSON, A., LARSSON, N. J., ET SWANSON, K. – Suffix trees on words. *Lecture Notes in Computer Science*, vol. 1075, 1996, pp. 102–??

[Apo85] APOSTOLICO, A. – The myriad virtues of suffix trees. *Dans: Combinatorial Algorithms on Words*, édité par Apostolico, A., et Galil, Z., pp. 85–96. – Springer-Verlag, Berlin, 1985.

[BBC92] BEAUQUIER, D., BERSTEL, J., ET CHRÉTIENNE, P. – *Eléments d'algorithmique.* – Masson, 1992.

[BBE+85] BLUMER, A., BLUMER, J., EHRENFEUCHT, A., HAUSSLER, D., CHEN, M. T., ET SEIFERAS, J. – The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.*, vol. 40, 1985, pp. 31–55.

[BCW90] BELL, T. C., CLEARY, J. G., ET WITTEN, I. H. – *Text Compression.* – Prentice Hall, 1990.

[Ben97] BENSON, G. – Sequence alignment with tandem duplication. *Dans: Recomb 97.* ACM Press, pp. 27–36. – Santa Fe, New Mexico - USA, January 1997.

[BM77] BOYER, R. S., ET MOORE, J. S. – A fast string searching algorithm. *Comm. ACM*, vol. 20, n° 10, 1977, pp. 762–772.

[Boh97] BOHÊME, O. – *L'arbre des suffixes: étude, construction et applications.* – Mémoire de fin d'études, Université de Mons-Hainaut, 1997.

[BP85] BERSTEL, J., ET PERRIN, D. – *Theory of Codes.* – New York, NY, Academic Press, 1985.

[Bru91] BRUYÈRE, V. – *Codes.* – Thèse de Doctorat, Université de Mons-Hainaut, 1991.

[BW94] BENSON, G., ET WATERMAN, M. S. – A method for fast database search for all k-nucleotide repeats. *Nucleic Acids Res.*, vol. 22, n° 22, November 1994, pp. 4828–4836.

[CL92] CHANG, W. I., ET LAMPE, J. – Theoretical and empirical comparisons of approximate string matching algorithms. *Dans : Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, édité par Apostolico, A., Crochemore, M., Galil, Z., et Manber, U. pp. 175–184. – Tucson, AZ, 1992.

[CL94] CHANG, W. I., ET LAWLER, E. L. – Sublinear approximate string matching and biological applications. *Algorithmica*, vol. 12, n° 3/4, 1994, pp. 327–344.

[CL96] CROCHEMORE, M., ET LECROQ, T. – *Text data compression algorithms.* – Rapport technique n° LIR 96.15 Informatique Fondamentale, Université de Rouen, 1996.

[CL97] CROCHEMORE, M., ET LECROQ, T. – Pattern matching and text compression algorithms. *Dans : The Computer Science and Engeneering Handbook*, édité par Tucker Jr., A. B., chap. 8, pp. 163–202. – Boca Raton, CRC Press, 1997.

[CMM+96] CAMPUZANO, V., MONTERMINI, L., MOLTO, M., PITANESE, L., ET COSSEE, M. – Friedreich's ataxia: Autosomal recessive disease caused by an intronic gaa triplet repeat expansion. *Science*, vol. 271, 1996, pp. 1423–1427.

[CP91] CROCHEMORE, M., ET PERRIN, D. – Two-way string-matching. *J. Assoc. Comput. Mach.*, vol. 38, n° 3, 1991, pp. 651–675.

[CPF+92] CASKEY, C., PIZZUTI, A., FU, Y.-H., FENWICK JR., R., ET NELSON, D. – Triplet repeat mutations in human disease. *Science*, vol. 256, 1992, p. 784.

[CR94] CROCHEMORE, M., ET RYTTER, W. – *Text algorithms.* – Oxford University Press, 1994.

[DAAa94] DUJON, B., ALEXANDRAKI, D., ANDRÉ, B., ET AL. – Complete DNA sequence of yeast chromosome XI. *Nature*, vol. 369, 1994, pp. 371–378.

[Dan93] DANCHIN, A. – Le séquençage de petits génomes. *La Recherche*, no251, février 1993, pp. 222–232.

[DDDR] DELGRANGE, O., DAUCHET, M., DELAHAYE, J.-P., ET RIVALS, E. – An effective algorithm for some sequence analysis and sequence comparison by optimal compression. – En préparation.

[Del94] DELAHAYE, J.-P. – *Information, complexité et hasard.* – Hermès, 1994.

[Del95] DELAHAYE, J.-P. – La compression des données. *Pour la Science*, vol. 217, novembre 1995, pp. 180–184.

[Del96a] DELGRANGE, O. – La compression de données. *Dans : Actes du Colloque Scientifique International Post-universitaire (CSIPWIC)*, édité par Aghion, J. Université de Liège, pp. 84–109. – Mons, Belgique, 12-16 août 1996.

[Del96b] DELGRANGE, O. – La compression informatique. *Dans : Compte-rendus de la neuvième journée de Mathématique et de Sciences*, édité par Noel, G., et Wautelet, M. Université de Mons-Hainaut. – Mons, Belgique, 4 avril 1996.

[EHJ+92] EDWARDS, A., HAMMOND, H., JIN, L., CASKEY, C., ET CHAKRABORTY, R. – Genetic variation at five trimeric and tetrameric tandem repeat loci in four human population groups. *Genomics*, vol. 12, 1992, pp. 241–253.

[Eli75] ELIAS, P. – Universal codeword sets and representations of the integers. *IEEE Trans. Inform. Theory*, vol. 33, n° 1, 1975, pp. 194–203.

[EM96] EL-MABROUK, N. – *Recherche approchée de motifs. Application à des séquences biologiques structurées.* – Thèse de Doctorat, Université de Paris VII, décembre 1996.

[ER78] EVEN, S., ET RODEH, M. – Economical encoding of commas between strings. *Comm. ACM*, vol. 21, n° 4, 1978, pp. 315–317.

[FAA+94] FELDMANN, H., AIGLE, M., ALJINOVIC, G., ANDRÉ, B., BACLET, M., ET AL. – Complete DNA sequence of yeast chromosome II. *EMBO Journal*, vol. 13, n° 5, 1994, pp. 795–809.

[Fal73] FALLER, N. – An adaptative system for data compression. *Dans : Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, pp. 593–597, 1973.

[FDRR86] FREIZNER-DEGEN, S., RAJPUT, B., ET REICH, R. – The human tissue plasminogen activator gene. *J. Biol. Chem.*, vol. 261, 1986, pp. 6972–6985.

[FLSS92] FISCHETTI, V. A., LANDAU, G. M., SCHMIDT, J. P., ET SELLERS, P. H. – Identifying periodic occurrences of a template with applications to protein struture. *Dans : Proceedings of the 3rd Annual Symposium of Combinatorial Pattern Matching*, édité par Apostolico, A., Crochemore, M., Galil, Z., et Manber, U. pp. 111–120. – Tucson, AZ, 1992.

[FPF+92] FU, Y.-H., PIZZUTI, A., FENWICK JR., R., KING, J., RAJNARAYAN, S., DUNNE, P., DUBEL, J., NASSER, G., ASHIZAWA, T., DEJONG, P., WIERINGA, B., KORNELUK, R., PERRYMAN, M., EPSTEIN, H., ET KASKEY, C. – An unstable triplet repeat in a gene related to myotonic muscular distrophy. *Science*, vol. 255, 1992, pp. 1256–1258.

[Gal78] GALLAGER, R. – Variations on a theme by Huffman. *IEEE Trans. Inform. Theory*, vol. 24, n° 6, November 1978, pp. 668–674.

[GBN94] GUSFIELD, D., BALASUBRAMANIAN, K., ET NAOR, D. – Parametric optimization of sequence alignment. *Algorithmica*, vol. 12, n° 3/4, 1994, pp. 312–326.

[GT93a] GRUMBACH, S., ET TAHI, F. – Compression of DNA sequences. *Dans : Proceedings of the IEEE Data Compression Conference 1993.*

[GT93b] GRUMBACH, S., ET TAHI, F. – A new challenge for compression algorithms: Genetic sequences. *Journal of Information Processing and Management*, 1993.

[GT95] GRUMBACH, S., ET TAHI, F. – Compression et compréhension de séquences nucléotidiques. *Technique et science informatique*, vol. 14, 1995, pp. 217–233.

[Gua80] GUAZZO, M. – A general minimum-redundancy source-coding algorithm. *IEEE Trans. Inform. Theory*, vol. 26, n° 1, 1980, pp. 15–25.

[HD94] HÉNAUT, A., ET DANCHIN, A. – Escherichia coli in silico. *Dans : Escherichia Coli and Salmonella typhimurium cellular and molecular biology.* – Washington DC, American Society for Microbiology, 1994.

[HM91] HELD, G., ET MARSHALL, T. R. – *Data Compression.* – John Wiley & Sons LTD, 1991, third édition.

[HSHG84] HAMADA, H., SEIDMAN, M., HOWARD, B., ET GORMAN, C. – Enhanced gene expression by the poly(dT-dG) poly(dC-dA) sequence. *Molecular and Cellular Biology*, vol. 4, 1984, pp. 2622–2630.

[HSSP88] HELLMAN, L., STEEN, M., SUNDVALL, M., ET PETTERSSON, U. – A rapidly evolving region in the immunoglobulin heavy chain loci of rat and mouse: postulated role of $(dC\text{-}dA)_n$ $(dG\text{-}dT)_n$ sequences. *Gene*, vol. 68, 1988, pp. 93–100.

[HU79] HOPCROFT, J. E., ET ULLMAN, J. D. – *Introduction to automata theory, languages and computations.* – Reading, MA, Addison-Wesley, 1979.

[Huf52] HUFFMAN, D. – A method for the construction of minimum-redundancy codes. *Proc. Institute of Electrical and Radio Engineers*, vol. 40, n° 9, September 1952, pp. 1098–1101.

[Hun93] HUNTINGTON'S DISEASE COLLABORATIVE RESEARCH GROUP. – A novel gene containing a trinucleotide repeat that is expanded and unstable on Huntington's disease chromosome. *Cell*, vol. 72, 1993, pp. 971–983.

[Ja94] JOHNSTON, M., ET AL. – Complete nucleotide sequence of Saccharomyces cerevisiae chromosome VIII. *Science*, vol. 265, n° 20, 1994, pp. 77–82.

[KMP77] KNUTH, D. E., MORRIS, JR, J. H., ET PRATT, V. R. – Fast pattern matching in strings. *SIAM J. Comput.*, vol. 6, n° 1, 1977, pp. 323–350.

[Knu73a] KNUTH, D. E. – *The art of computer programming: fundamental algorithms.* – Reading, MA, Addison-Wesley, 1973, volume 1.

[Knu73b] KNUTH, D. E. – *The art of computer programming: Sorting and searching.* – Reading, MA, Addison-Wesley, 1973, volume 3.

[Kra49] KRAFT, L. – *A device for quantizing, grouping and coding amplitude modulated pulses.* – Cambridge, Mass., Thèse, Dept. of Electricel Engineering, M.I.T., 1949.

[Lar96] LARSSON, N. J. – Extended application of suffix trees to data compression. *Dans : Proceedings of the IEEE Data Compression Conference 1996*, pp. 190–199.

[Lef96] LEFEBVRE, F. – A grammar-based unification of several alignment and folding algorithms. *Dans : Fourth International Conference on Intelligent Systems for Molecular Biology.* – AAAI Press. Available at: ftp://lix.polytechnique.fr/pub/lefebvre.

[Lev66] LEVENSHTEIN, V. I. – Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, vol. 10, n° 8, 1966, pp. 707–710.

[Lew92] LEWIN, B. – *Gènes.* – Médecine et Sciences Flammarion, 1992, volume III.

[LH87] LELEWER, D. A., ET HIRSCHBERG, D. S. – Data compression. *ACM Computing Surveys*, vol. 19, n° 3, 1987, pp. 261–296.

[LHYN95] LOWENSTERN, D., HIRSH, H., YIANILOS, P., ET NOORDEWIER, M. – *DNA Sequence Classification Using Compression-Based Induction.* – Rapport technique n° 4, DIMACS, Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore, April 1995. Available at: http://dimacs.rutgers.edu/TechnicalReports/1995.html.

[LP85] LIPMAN, D. J., ET PEARSON, W. R. – Rapid and sensitive protein similarity searches. *Science*, vol. 227, 1985, pp. 1435–1441.

[LV93] LI, M., ET VITÁNYI, P. M. – *An Introduction to Kolmogorov Complexity and its Applications.* – Springer-Verlag, 1993.

[LWGE93] LU, Q., WALLRATH, L., GRANOK, H., ET ELGIN, S. – $(\mathtt{CT})_n(\mathtt{GA})_n$ repeats and heat shock elements have distinct roles in chromatin structure and transcriptional activation of the Drosophila hsp26 gene. *Molecular and Cellular Biology*, vol. 13, 1993, pp. 2802–2814.

[LWL+91] LA SPADA, E., WILSON, E., LUBAHN, D., HARDING, A., ET FISCHBECK, K. – Androgen receptor gene mutation in X-linked spinal and bulbar muscular atrophy. *Nature*, vol. 352, 1991, p. 77.

[McC76] MCCREIGHT, E. M. – A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.*, vol. 23, n° 2, 1976, pp. 262–272.

[Mil93] MILOSAVLJEVIĆ, A. – Discovering sequence similarity by the algorithmic significance method. *Dans : First International Conference on Intelligent Systems for Molecular Biology.* pp. 284–291. – AAAI Press.

[MJ93] MILOSAVLJEVIĆ, A., ET JURKA, J. – Discovering simple DNA sequences by the algorithmic significance method. *CABIOS*, vol. 9, n° 4, 1993, pp. 407–411.

[Mor68] MORRISON, D. R. – PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. Assoc. Comput. Mach.*, vol. 15, 1968, pp. 514–534.

[MR80] MAJSTER, M. E., ET RYSER, A. – Efficient on-line construction and correction of position trees. *SIAM J. Comput.*, vol. 9, n° 4, 1980, pp. 785–807.

[Mye91] MYERS, E. W. – *An Overview of Sequence Comparison Algorithms in Molecular Biology.* – Rapport technique n° 29, Department of Computer Science, University of Arizona, 1991. Available at http://www.cs.arizona.edu/people/gene.

[Nel93] NELSON, M. – *La compression de données.* – Dunod, 1993.

[NW70] NEEDLEMAN, S. B., ET WUNSCH, C. D. – A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, vol. 48, 1970, pp. 443–453.

[OdAAC+92] OLIVER, S., DER AART, Q. V., AGOSTINI-CARBONE, M., AIGLE, M., ET AL. – The complete DNA sequence of yeast chromosome III. *Nature*, vol. 357, 1992, pp. 383–401.

[ODH95] OLLIVIER, E., DELORME, M.-O., ET HÉNAUT, A. – DosDNA occurs along yeast chromosomes, regardless of functional significance of the sequence. *Compte Rendu de l'Académie des Sciences*, vol. 318, 1995, pp. 599–608.

[Pea90] PEARSON, W. R. – Rapid and sensitive sequence comparison with FASTP and FASTA. *Meth. Enzymol.*, vol. 183, 1990, pp. 63–98.

[PL88] PEARSON, W. R., ET LIPMAN, D. J. – Improved tools for biological sequence comparison. *Proc. Nat. Acad. Sci. U.S.A.*, vol. 85, 1988, pp. 2444–2448.

[PLRN87] PARDUE, M., LOWENHAUPT, K., RICH, A., ET NORDHEIM, A. – $(\mathrm{dC\text{-}dA})_n$ $(\mathrm{dG\text{-}dT})_n$ sequences have evolutionarily conserved chromosomal locations in drosophila with implications for roles in chromosome structure and function. *The EMBO Journal*, vol. 6, 1987, pp. 1781–1789.

[RDD+97] RIVALS, É., DELGRANGE, O., DELAHAYE, J.-P., DAUCHET, M., DELORME, M.-O., HÉNAUT, A., ET OLLIVIER, E. – Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences. *CABIOS*, vol. 13, n° 2, Avril 1997, pp. 131–136.

[RDDD95] RIVALS, É., DELGRANGE, O., DELAHAYE, J.-P., ET DAUCHET, M. – A first step toward chromosome analysis by compression algorithms. *Dans : International IEEE Symposium on Intelligence in Neural and Biological Systems*, édité par Bourbakis, N. G., et Head, T. IEEE Computer Society Press, pp. 233–239. – Herndon-Washington DC, may 29–31 1995.

[RDDD96] RIVALS, E., DAUCHET, M., DELAHAYE, J.-P., ET DELGRANGE, O. – Compression and genetic sequence analysis. *Biochimie*, vol. 78, 1996, pp. 315–322.

[RDDD97] RIVALS, E., DAUCHET, M., DELAHAYE, J.-P., ET DELGRANGE, O. – Discerning repeats in DNA sequences with a compression algorithm. – 1997. Unpublished.

[RHYS93] RICHARDS, R., HOLMAN, K., YU, S., ET SOUTHERLAND, G. – Fragile X syndrome unstable element, p(CCG)$_n$, and other simple tandem repeat sequences are binding sites for specific nuclear protein. *Hum. Mol. Genet.*, vol. 2, 1993, pp. 1429–1435.

[Riv94] RIVALS, E. – Compression pour l'analyse de séquences. *Dans : Actes de la rencontre pour les Recherches de Motifs dans les Séquences GREG*, édité par Crochemore, M. Institut Gaspard Monge, Université de Marne-La-Vallée, pp. 69–81. – Marseille, février 1994.

[Riv96] RIVALS, É. – *Algorithmes de compression et applications à l'analyse de séquences génétiques.* – Thèse de Doctorat, Université des Sciences et Technologies de Lille, janvier 1996.

[RL79] RISSANEN, J., ET LANGDON JR., G. – Arithmetic coding. *IBM J. Res. Develop.*, vol. 23, n° 2, 1979, pp. 149–162.

[Rod82] RODEH, M. – A fast test for unique decipherability based on suffix trees. *IEEE Trans. Inform. Theory*, vol. 28, 1982, pp. 648–651.

[RS92] RICHARDS, R., ET SUTHERLAND, G. – Heritable unstable DNA sequences and human genetic disease. *Cell*, vol. 70, 1992, p. 709.

[Sag96] SAGOT, M.-F. – *Ressemblance lexicale et structurale entre macromolécules - Formalisation et approches combinatoires.* – Thèse de Doctorat, Université de Marne-La-Vallée, 9 juillet 1996.

[Sel74] SELLERS, P. H. – On the theory and computation of evolutionary distance. *SIAM J. Appl. Math.*, vol. 26, 1974, pp. 787–793.

[Sha48] SHANNON, C. – The mathematical theory of communication. *Bell System Technical J.*, vol. 27, 1948, pp. 379–423,623–656.

[SK83] SANKOFF, D., ET KRUSKAL, J. B. – *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison.* – Reading, MA, Addison-Wesley, 1983.

[SM95] SEARLS, D. B., ET MURPHY, K. P. – Automata-theoretic models of mutation and alignment. *Dans : Third International Conference on Intelligent Systems for Molecular Biology.* pp. 341–349. – AAAI Press.

[Ste94] STEPHEN, G. A. – *String searching algorithms.* – World Scientific Press, 1994.

[Sto88] STORER, J. A. – *Data Compression: Methods and Theory.* – Rockville, MD, Computer Sciences Press, 1988.

[SW49] SHANNON, C., ET WEAVER, W. – *The mathematical theory of communication.* – Urbana, IL, University of Illinois Press, 1949.

[SW92] SINDEN, R., ET WELLS, R. – DNA structure, mutations, and human genetic disease. *Curr. Opin. Biotechnol.*, vol. 3, 1992, p. 612.

[TL94] Time-Life (édité par). – *L'héritage génétique.* – Amsterdam, Time-Life, 1994, *À la découverte de l'être humain.*

[Tri90] TRIFONOV, E. – Making sense of the human genome. *Dans : Human Genome Initiative and DNA Recombination*, édité par Sarma, R. H., et Sarma, M. H., pp. 69–77. – Adenine Press, 1990.

[Ukk92] UKKONEN, E. – Constructing suffix trees on-line in linear time. *Dans : Proceedings of the IFIP 12th World Computer Congress*, édité par van Leeuwen, J. pp. 484–492. – Madrid, 1992.

[Ukk93] UKKONEN, E. – Approximate string matching over suffix trees. *Dans : Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, édité par Apostolico, A., Crochemore, M., Galil, Z., et Manber, U. pp. 228–242. – Padova, Italy, 1993.

[Ukk95] UKKONEN, E. – On-line construction of suffix trees. *Algorithmica*, vol. 14, n° 3, 1995, pp. 249–260.

[VPS+91] VERKERK, A., PIERETTI, M., SUTCLIFFE, J., FU, Y., KUHL, D., PIZZUTI, A., REINER, O., RICHARDS, S., VICTORIA, M., ZHANG, F., EUSSEN, B., VAN OMMEN, G., BLONDEN, A., RIGGINS, G., CHASTAIN, J., KUNST, C., GALJAARD, H., CASKEY, C., NELSON, D., OOSTRA, B., ET WARREN, S. – Identification of a gene (FMR-1) containing a CGG repeat coincident with a breakpoint cluster region exhibiting length variation in fragile X syndrome. *Cell*, vol. 44, 1991, pp. 388–396.

[Wat89] WATERMAN, M. S. – *Mathematical methods for DNA sequences.* – Boca Raton, CRC Press, 1989.

[WC53] WATSON, J. D., ET CRICK, F. H. – Molecular structure of nucleic acids. A structure for desoxyribose nucleic acid. *Nature*, no171, 1953, pp. 737–738.

[Wei73] WEINER, P. – Linear pattern matching algorithm. *Dans : Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11. – Washington, DC, 1973.

[Wel84] WELCH, T. – A technique for high-performance data compression. *IEEE Computer*, vol. 17, n° 6, june 1984, pp. 8–19.

[WF75] WAGNER, R. A., ET FISCHER, M. – The string-to-string correction problem. *J. Assoc. Comput. Mach.*, vol. 21, 1975, pp. 168–173.

[WM89] WEBER, J., ET MAY, P. – Abundant class of human DNA polymorphisms which can be typed using polymerase chain reaction. *Am J. Human Genet.*, vol. 44, 1989, pp. 388–396.

[WNC87] WITTEN, I., NEAL, R., ET CLEARY, J. – Arithmetic coding for data compression. *Comm. ACM*, vol. 30, n° 6, 87, pp. 520–540.

[WS93] WELLS, R., ET SINDEN, R. – Genome rearrangement and stability. *Dans : Genome Analysis, 7*, édité par Davies, K., et Warren, S., chap. Defined Ordered Sequence DNA, DNA Structure and DNA-directed Mutation. – Cold Spring Harbor Laboratory Press, 1993.

[YWvR91] YEE, H., WONG, A., VAN DEN SANDE, J., ET RATTNER, J. – Identification of novel single stranded $d(TC)_n$ binding proteins in several mamalian species. *Nucleic Acids Res.*, vol. 19, 1991, pp. 949–953.

[Zip90] ZIPSTEIN, M. – *Les méthodes de compression de textes algorithmes et performances.* – Thèse de Doctorat, Université Paris VII, 1990.

[ZL77] ZIV, J., ET LEMPEL, A. – A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, vol. 23, n° 3, may 1977, pp. 337–343.

[ZL78] ZIV, J., ET LEMPEL, A. – Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, vol. 24, n° 5, september 1978, pp. 530–536.

Table des figures

Liste des tableaux

Annexes

Annexe A

Algorithme de McCreight de construction de l'arbre des suffixes

Cette annexe présente l'algorithme de construction d'arbre des suffixes conçu par Mc Creight en 1976 [McC76]. Les notations utilisées sont celles introduites dans la partie I, chapitre 1, section 1.3.

Les trois algorithmes les plus connus (celui de Weiner, développé en 1973 [Wei73], celui de McCreight en 1976 [McC76] et celui de Ukkonen en 1992 [Ukk92, Ukk95]), construisent l'arbre en temps $O(n)$. Celui de McCreight nécessite moins de mémoire qui celui de Weiner. Celui de Ukkonen construit l'arbre *en ligne*, c'est-à-dire au fur et à mesure de la lecture, de gauche à droite, des symboles du texte. Cet algorithme étant relativement complexe et, puisque la construction en ligne n'est pas utile dans notre cas, nous ne l'aborderons pas. Nous présentons donc celui de McCreight.

Soit $x = x_1 x_2 \dots x_n$, avec $x_i \neq x_n, \forall i : 1 \leq i < n$. L'arbre des suffixes de x est noté $ST(x)$.

Avant de détailler l'algorithme qui utilise astucieusement une propriété des suffixes, nous présentons un algorithme intuitif appelé *algorithme en force brute* (pour "brute force" en anglais). Cet algorithme construit $ST(x)$ en temps $O(n^2)$ mais il permet de présenter l'amélioration de McCreight et de définir certains concepts utiles.

A.1 Algorithme en force brute

Une idée simple pour construire $ST(x)$ est de partir d'un arbre trivial ne contenant qu'un nœud et d'y insérer successivement les chemins correspondant à tous les suffixes $x_{i..}$ de x.

Considérons la fonction BRUTEFORCEST présentée à la figure A.1. Les suffixes de x

```
BRUTEFORCEST(x, n)
1  ST_0 ← INITARBRE()
2  Pour i ← 1 Jusque n
3  Faire ST_i ← INSÈRESUF(ST_{i-1}, x_{i..})
4  Retourner ST_n
```

FIG. A.1 - *Algorithme en force brute*

sont insérés dans l'arbre des suffixes du plus long au plus court. L'ordre dans lequel ils sont

insérés n'a pas vraiment d'importance à ce stade-ci mais l'algorithme de McCreight qui sera présenté au point A.2 exploite spécifiquement cet ordre. Pour cette raison, nous avons choisi ici d'insérer les suffixes en suivant le même ordre.

Dans cet algorithme, ST_i désigne l'*arbre partiel des suffixes* de x après la $i^{\text{ème}}$ étape de la construction de $ST(x)$. L'arbre ST_n est l'arbre final $ST(x)$. La fonction INITARBRE crée l'arbre initial ST_0 ne contenant qu'un seul nœud : la racine du futur arbre des suffixes.

Nous détaillons maintenant la fonction INSÈRESUF qui crée l'arbre ST_i par insertion du suffixe $x_{i..}$ dans ST_{i-1}.

Soit $head_i$ le plus long préfixe de $x_{i..}$ qui soit également préfixe d'un suffixe $x_{j..}$ avec $j < i$. Puisque $j < i$, $fact(x_{j..})$ est une feuille de ST_{i-1} et $head_i$ est donc le plus long préfixe de $x_{i..}$ qui possède un locus étendu dans ST_{i-1}. Soit $tail_i$ le mot tel que $x_{i..} = head_i.tail_i$. Puisqu'aucun suffixe de x n'est préfixe d'un autre suffixe de x, $tail_i \neq \epsilon$. La première étape de INSÈRESUF est donc la localisation du locus étendu de $head_i$ dans ST_{i-1}. Si ce locus étendu n'est pas le locus de $head_i$, alors ce dernier doit être créé. Le nouveau nœud servant de locus à $head_i$ est inséré au milieu de l'arc joignant son locus contracté à son locus étendu. Les étiquettes des deux arcs ainsi créés sont obtenues par séparation de l'étiquette de l'ancien arc à l'endroit ou les caractères ne correspondent plus avec ceux de $x_{i..}$. Pour terminer, la feuille $locus(x_{i..})$ est insérée comme fils de $locus(head_i)$, l'arc qui les relie est étiqueté par $tail_i$. L'arbre ST_i est alors construit.

Cette méthode est illustrée à la figure A.2 qui présente les étapes successives de la construction de $ST(\texttt{ACAC\$})$.

Pour réaliser son travail, INSÈRESUF dispose des deux fonctions générales COUPEARC et LOCUSPRÉFIXECOMMUN. Supposons qu'un nœud g possède un arc sortant $arc(g, f)$ tel que le mot w soit préfixe propre de $etiq(arc(g, f))$. L'appel COUPEARC(g, w) crée un nouveau nœud h qui coupe l'arc $arc(g, f)$ en deux arcs $arc(g, h)$ et $arc(h, f)$ de telle sorte que $etiq(arc(g, h)) = w$ et l'ancienne étiquette $etiq(arc(g, f))$ soit égale à $w.etiq(arc(h, f))$ (voir figure A.3). Si l'accès à un fils spécifique d'un nœud se fait en temps $O(1)$ (voir 1.3.4), alors le travail de COUPEARC se réalise également en temps $O(1)$. COUPEARC retourne le nœud créé.

Soit g la racine d'un arbre des suffixes partiel et w un mot qui n'est le préfixe d'aucun suffixe représenté dans l'arbre. Soit *head* le plus long préfixe de w qui possède un locus étendu dans l'arbre de racine g : $w = head.tail$. L'appel LOCUSPRÉFIXECOMMUN$(g, w, IndiceTail)$ retourne le locus de *head* dans l'arbre de racine g (voir figure A.4). LOCUSPRÉFIXECOMMUN fait éventuellement appel à COUPEARC pour créer le locus s'il n'existait pas. Le but de cette fonction dans l'algorithme glouton est de rechercher et éventuellement de construire le locus de $head_i$ lors de l'exécution de INSÈRESUF.

La fonction retourne également, via son troisième paramètre *IndiceTail*, l'indice, dans w, du début du mot *tail*. L'algorithme de LOCUSPRÉFIXECOMMUN est présenté à la figure A.5. La fonction ACCÈDEFILS accède au fils d'un nœud étant donnée la première lettre de l'arc qui y mène. Si ce fils n'existe pas, elle retourne NIL. À partir de la racine g, le chemin étiqueté par le plus long préfixe de w est parcouru caractère par caractère. Si le processus s'arrête sur un nœud spécifique parce qu'il ne possède pas le fils adéquat, le plus long préfixe a été trouvé et son locus existe déjà. Cela signifie que deux suffixes insérés auparavant dans l'arbre possédaient déjà ce mot comme plus long préfixe commun. Si la fin du préfixe est détectée au milieu d'un arc, alors COUPEARC est appelé pour créer le nouveau locus.

Dans le pire des cas, tous les caractères de w, excepté le dernier, seront parcourus. LOCUSPRÉFIXECOMMUN s'exécute donc en temps $O(|w|)$.

Grâce à l'appel $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(racine, x_{i..}, IndiceTail)$, la fonc-

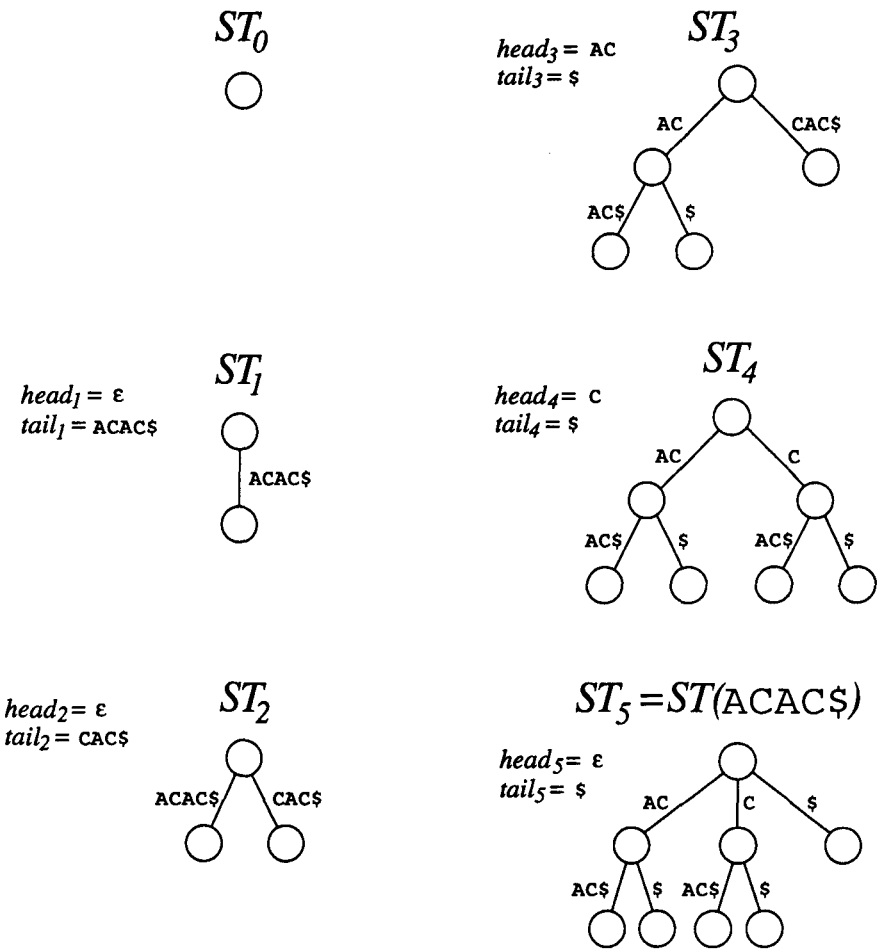


FIG. A.2 - *Les 5 étapes de la construction de ST(ACAC$)*

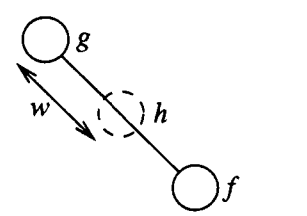


FIG. A.3 - COUPEARC(g, w)

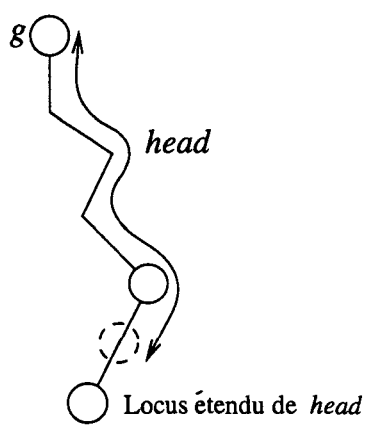


FIG. A.4 - LOCUSPRÉFIXECOMMUN$(g, w, IndiceTail)$

```
LocusPréfixeCommun(g, w, IndiceTail)
 1  NœudCourant ← g
 2  i ← 1                                  /* Position courante dans w */
 3  Trouve ← Faux
 4  TantQue Pas(Trouve)
 5  Faire h ← AccèdeFils(NœudCourant, w_i)
 6                                         /* On accède au fils grâce à la première lettre w_i */
 7       Si h = Nil
 8          Alors IndiceTail ← i
 9                Retourner NœudCourant
10          Sinon j ← 1                   /* Position courante dans l'arc */
11                u ← etiq(arc(NœudCourant, h))
12                TantQue (j ≤ |u|) et (u_j = w_i)
13                Faire i ← i + 1         /* Comparaison */
14                      j ← j + 1
15                Si j > |u|              /* Discordance */
16                   Alors NœudCourant ← h
17                   Sinon Trouve ← Vrai
18  IndiceTail ← i
19  Retourner CoupeArc(NœudCourant, w_{i-j+1..i-1})
```

FIG. A.5 - *Fonction* LocusPréfixeCommun *de recherche du locus du plus long préfixe de w présent dans l'arbre de racine g*

tion InsèreSuf connaît maintenant $locus(head_i)$. Il lui suffit alors de créer la nouvelle feuille correspondant au suffixe $x_{i..}$ et de créer et étiqueter l'arc qui la lie à son père $locus(head_i)$. L'étiquette de l'arc est déterminée par le couple $(IndiceTail, n)$. La création de la feuille et l'arc se fait en temps $O(1)$ par un appel de type CréeNœud$(locus(head_i), IndiceTail, n)$ où CréeNœud(p, i_1, i_2) crée un fils du nœud p, l'étiquette du nouvel arc étant déterminée par le couple d'indices (i_1,i_2) dans x.

La fonction LocusPréfixeCommun étant appelée pour chaque suffixe $x_{i..}$ de x, l'algorithme glouton fonctionne en temps $O(|x_{1..}| + |x_{2..}| \ldots + 1) = O(n^2)$. Cette borne maximale est atteinte par exemple pour $x =$ **AAAAAAAAAA$**.

A.2 Algorithme de McCreight

L'algorithme de McCreight [McC76] suit le même schéma que l'algorithme en force brute : les suffixes de x sont insérés dans $ST(x)$ en allant du plus long $(x_{1..})$ au plus court $(x_{n..})$. À chaque étape i, l'algorithme recherche (et éventuellement construit) le nœud $locus(head_i)$ et crée ensuite la feuille correspondant au suffixe $x_{i..}$. Pour gagner du temps lors de la recherche de $head_i$, l'algorithme de McCreight utilise une propriété qui lie le suffixe $x_{i..}$ au suffixe précédent $x_{i-1..}$.

Nous commençons la présentation par deux propriétés générales de l'arbre des suffixes qui seront souvent utilisées par la suite. La première proposition présentée découle immédiatement du schéma d'insertion des suffixes.

A.2.1 Propriétés des nœuds internes

Proposition A.1 *Tout nœud interne de $ST(x)$ a été créé à une étape i parce qu'il était le locus de $head_i$.*

Preuve. Vrai par construction de $ST(x)$. □

La proposition A.2 permet de définir, pour chaque nœud interne, son *lien suffixe* qui permettra de court-circuiter la recherche des $head_i$.

Proposition A.2 *Si un nœud interne g, locus de $a.\gamma$, avec $a \in \mathcal{A}$ et $\gamma \in \mathcal{A}^*$ est un nœud interne de $ST(x)$ alors le nœud h, locus de γ, existe et est également un nœud interne de $ST(x)$.*

Preuve. Soit $l = |a\gamma|$. D'après la proposition A.1, il existe i tel que $head_i = a\gamma$. Cela signifie qu'il existe $j < i$ tel que $a\gamma$ est le plus long préfixe commun entre $x_{i..}$ et $x_{j..}$: $x_{i..i+l-1} = x_{j..j+l-1} = a\gamma$ mais $x_{i+l} \neq x_{j+l}$. Dans ce cas, γ est le plus long préfixe commun entre $x_{i+1..}$ et $x_{j+1..}$: $x_{i+1..i+l-1} = x_{j+1..j+l-1} = \gamma$ mais $x_{i+l} \neq x_{j+l}$. Puisque $j+1 < i+1$, lors de l'insertion du suffixe $x_{i+1..}$, il y aura séparation entre $chemin(racine, locus(x_{i+1..}))$ et $chemin(racine, locus(x_{j+1..}))$ au nœud $locus(\gamma)$ (voir figure A.6). □

Remarque A.1 *Dans la preuve de la proposition A.2, le mot γ n'est pas forcément égal à $head_{i+1}$. En effet, il se peut qu'un suffixe $x_{k+1..}$ avec $k+1 < i+1$ ait un plus long préfixe commun avec $x_{i+1..}$ qui soit plus long que celui que partagent $x_{i+1..}$ et $x_{j+1..}$. Pour cela, il suffit simplement que $x_{k..} = b.\gamma.\alpha.\beta$ et $x_{i..} = a.\gamma.\alpha.\delta$ avec $b \in \mathcal{A}$ et $a \neq b$. Dans ce cas, $x_{k..}$ et $x_{i..}$ n'ont aucun préfixe en commun mais $x_{k+1..}$ et $x_{i+1..}$ ont $\gamma\alpha$ en commun (voir figure A.6).*

Dès lors, tout nœud interne $g = locus(a\gamma)$ possède un nœud équivalent $h = locus(\gamma)$. Cependant, les caractéristiques de g et de h ne sont pas identiques : h peut posséder plus de fils ou plus de descendants que g. En effet, soit T_1 le sous-arbre dont g est la racine. Soit T le sous-arbre dont h est la racine (voir figure A.7). Supposons que f, locus de $b\gamma$ avec $b \in \mathcal{A}$ et $b \neq a$ existe et soit racine du sous-arbre T_2, alors le sous-arbre T est en quelque sorte une "fusion" entre T_1 et T_2.

A.2.2 Liens suffixes

Nous pouvons maintenant ajouter, au stockage de chaque nœud interne $g = locus(a\gamma)$, un pointeur vers le nœud $h = locus(\gamma)$.

L'algorithme de McCreight se sert efficacement de ce nouveau pointeur appelé *lien suffixe*. Il doit également, au fur et à mesure de la création des nœuds internes, donner une valeur correcte à ces liens. On note $h = liensuf(g)$ si $g = locus(a\gamma)$ et $h = locus(\gamma)$. On note également $h = suf(a\gamma)$.

Si à la fin de la construction de l'arbre des suffixes, tous les nœuds internes sauf la racine possèdent un lien suffixe valide, ce n'est pas le cas lors des étapes intermédiaires. Ainsi, la proposition A.3 montre que $liensuf(locus(head_{i-1}))$ ne peut recevoir une valeur correcte qu'à la fin de l'étape i.

Proposition A.3 *À la fin de l'étape i, après insertion de $x_{i..}$ dans l'arbre des suffixes intermédiaire, si $head_{i-1} \neq \epsilon$, alors la valeur à donner à $liensuf(locus(head_{i-1}))$ est connue.*

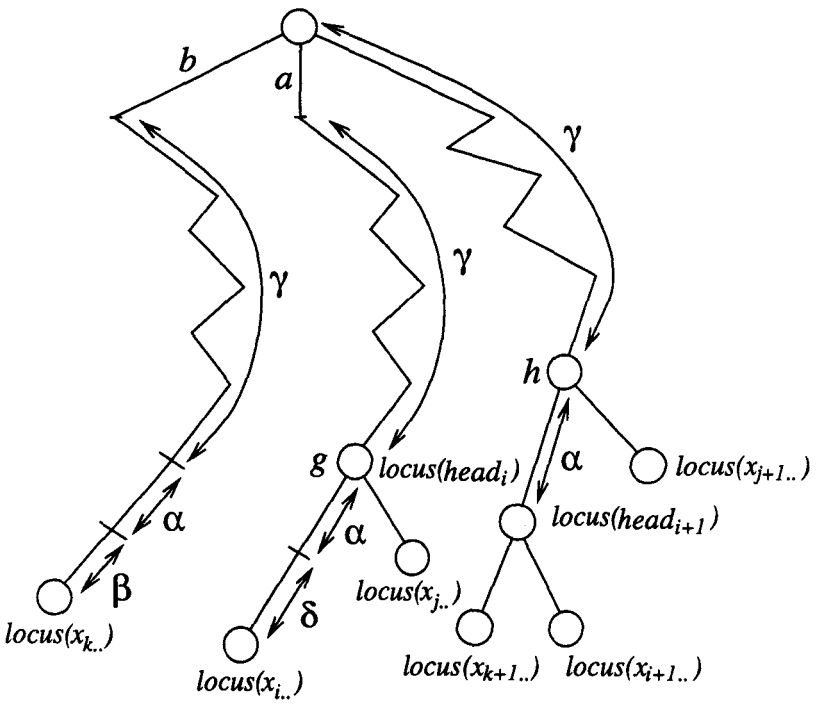


FIG. A.6 - *Insertion de* $x_{i+1..}$

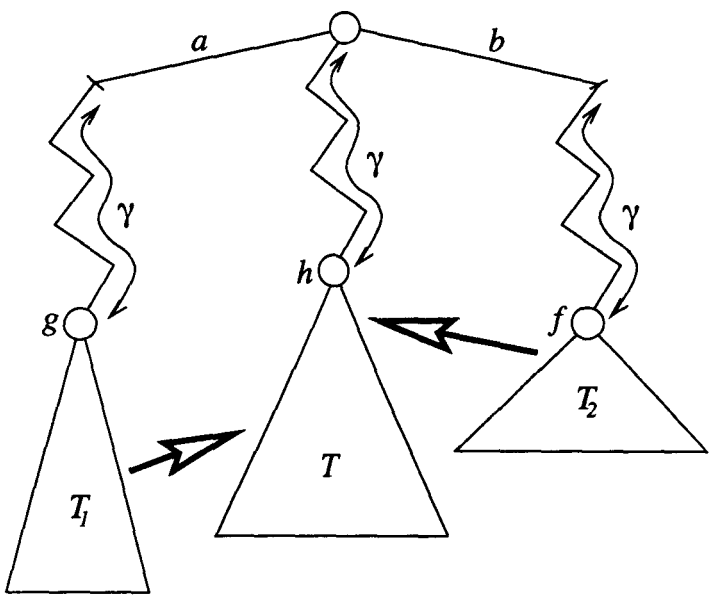


FIG. A.7 - *T fusion de* T_1 *et de* T_2

Preuve. Puisque $head_{i-1} \neq \epsilon$, posons $head_{i-1} = a.\gamma$ avec $a \in \mathcal{A}$ et $\gamma \in \mathcal{A}^*$. Il existe donc $j-1 < i-1$ tel que $a\gamma$ soit le plus long préfixe commun à $x_{j-1..}$ et $x_{i-1..}$. Donc, γ est le plus long préfixe commun à $x_{j..}$ et $x_{i..}$.

- Si $\gamma = head_i$, alors $locus(\gamma)$ est créé à l'étape i.
- Si $\gamma \neq head_i$, alors γ est préfixe de $head_i$ et $head_i$ est le plus long préfixe commun entre $x_{i..}$ et $x_{k..}$ avec $k < i$. Dès lors, γ est le plus long préfixe commun entre $x_{j..}$ et $x_{k..}$. Le nœud $locus(\gamma)$ existe donc, il a été créé à une étape antérieure. □

A.2.3 L'algorithme

Plaçons-nous après l'étape $i-1$. Nous devons maintenant insérer $x_{i..}$. Pour cela, nous pouvons nous aider des valeurs de tous les $liensuf(g)$ avec g nœud interne autre que la racine et autre que $locus(head_{i-1})$. Pour que ce soit vrai à chaque étape, il est indispensable après insertion de $x_{i..}$ dans l'arbre des suffixes, que l'algorithme donne la valeur adéquate à $liensuf(locus(head_{i-1}))$.

Recherchons maintenant $head_i$. Supposons, dans un premier temps, que ni $locus(head_{i-1})$ ni $pere(locus(head_{i-1}))$ ne soient la racine. Soit $head_{i-1} = a\gamma\delta$ avec $a \in \mathcal{A}; \gamma, \delta \in \mathcal{A}^*$ et $\delta = etiq(arc(pere(locus(head_{i-1})), locus(head_{i-1})))$ (voir figure A.8).

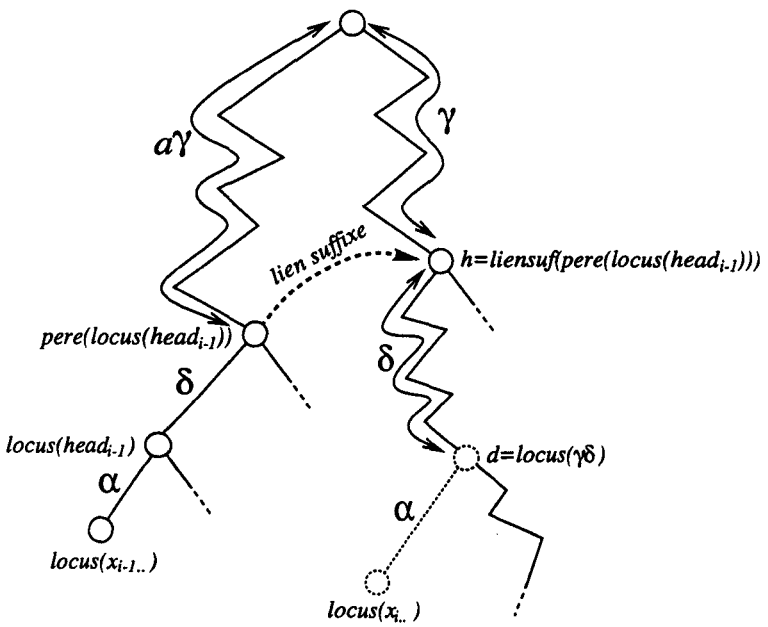


FIG. A.8 - *Arbre intermédiaire durant l'étape i*

Le mot $\gamma\delta$ est préfixe de $x_{i..}$ car $a\gamma\delta$ est préfixe de $x_{i-1..}$. Soit le mot α tel que $x_{i-1..} = a\gamma\delta\alpha$ et $x_{i..} = \gamma\delta\alpha$.

Puisque $fact(pere(locus(head_{i-1}))) = a\gamma$, alors $h = liensuf(pere(locus(head_{i-1})))$ est le locus de γ. Nous savons qu'il existe grâce à la proposition A.3. On accède donc au nœud h à partir de $pere(locus(head_{i-1}))$ en temps constant. Une première amélioration par rapport à l'algorithme en force brute consiste à rechercher $head_i$ non plus à partir de la racine mais à partir de h : $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(h, \delta\alpha, IndiceTail)$.

Les propositions A.4 et A.5 permettent d'accélérer également la recherche de $head_i$ à partir du nœud h.

Proposition A.4 *Le mot $\gamma\delta$ est préfixe d'un suffixe déjà représenté dans l'arbre actuel.*

Preuve. Puisque $a\gamma\delta = head_{i-1}$, $a\gamma\delta$ est le plus long préfixe commun entre $x_{i-1..}$ et $x_{j..}$ avec $j < i-1$. Donc $\gamma\delta$ est préfixe du suffixe $x_{j+1..}$ inséré à l'étape $j+1 < i$. □

Proposition A.5 *Si le locus de $\gamma\delta$ n'existe pas dans l'arbre actuel, alors $head_i = \gamma\delta$.*

Preuve. Puisque $a\gamma\delta$ est le plus long préfixe commun entre $x_{i-1..}$ et $x_{j..}$ avec $j < i-1$, $\gamma\delta$ est le plus long préfixe commun entre $x_{i..}$ et $x_{j+1..}$. Si le locus de $\gamma\delta$ n'existe pas, alors $\gamma\delta$ est le plus long préfixe commun entre $x_{i..}$ et tout mot de l'arbre actuel, en particulier $x_{j+1..}$, ce qui correspond à la définition de $head_i$. □

Si le locus de $\gamma\delta$ existe déjà dans l'arbre, cela signifie que $\gamma\delta$ n'est pas forcément le plus long préfixe commun entre $x_{i..}$ et un mot représenté dans l'arbre actuel (le raisonnement est identique à celui de la remarque A.1). Il faut alors rechercher $head_i$ en partant du nœud $d = locus(\gamma\delta)$ en parcourant les caractères de α un à un: $locus(head_i) \leftarrow$ LocusPréfixeCommun$(d, \alpha, IndiceTail)$.

Puisque $\gamma\delta$ possède un locus étendu dans l'arbre, la recherche (et la création éventuelle) de $d = locus(\gamma\delta)$ à partir du nœud h peut se faire beaucoup plus rapidement qu'en utilisant LocusPréfixeCommun. Il suffit, à partir de h, de suivre le chemin étiqueté par δ, en ne testant que les premières lettres des étiquettes des arcs. Lorsque tout le mot δ a été lu, si on se trouve au milieu d'un arc, alors CoupeArc est appelé pour créer le nouveau nœud d. Dans le cas contraire, cela signifie que le nœud existe déjà.

Soit RechercheRapide la fonction capable d'effectuer une telle recherche. Son algorithme est présenté à la figure A.9.

```
RechercheRapide(g, w)
 1  NœudCourant ← g
 2  i ← 1                                  /* Position courante dans w */
 3  Trouve ← Faux
 4  TantQue Pas(Trouve)
 5  Faire Si i > |w|                       /* Le locus de w existe */
 6          Alors Retourner NœudCourant
 7          Sinon h ← AccèdeFils(NœudCourant, w_i)
 8                                         /* On accède au fils grâce à la première lettre w_i */
 9                l ← |etiq(arc(NœudCourant, h))|
10                Si l > |w| − i + 1       /* L'arc est trop long */
11                   Alors Trouve ← Vrai
12                   Sinon NœudCourant ← h /* L'arc est trop court */
13                         i ← i + l
14  Retourner CoupeArc(NœudCourant, w_i..)
```

FIG. A.9 - *Fonction* RechercheRapide *qui recherche (et construit) le locus de w dans l'arbre de racine g*

La recherche et construction éventuelle de $d = locus(\gamma\delta)$ se fait donc simplement, à partir du nœud h par l'appel: $d \leftarrow$ RechercheRapide(h, δ). Si un nouveau nœud est créé, alors

c'est $head_i$: $head_i \leftarrow d$ (proposition A.5), sinon il faut rechercher $head_i$ en suivant les caractères de α un à un à partir de d : $head_i \leftarrow$ LocusPréfixeCommun$(d, \alpha, IndiceTail)$. Lorsque $locus(head_i)$ est connu, il suffit alors de lui créer le fils $locus(x_{i..})$ comme nouvelle feuille : CréeNœud$(locus(head_i), j, n)$ où $j = IndiceTail + |\gamma\delta| + i - 1$ si LocusPréfixeCommun a été appelé ou $j = n - |\alpha|$ si LocusPréfixeCommun n'a pas été appelé.

Dans tous les cas, l'appel à RechercheRapide a permis de localiser $d = locus(\gamma\delta)$ qui est la valeur correcte à donner au lien suffixe de $locus(head_{i-1})$.

Il nous reste à comprendre ce qu'il faut faire lorsque $pere(locus(head_{i-1}))$ est la racine ou n'existe pas.

Si $pere(locus(head_{i-1}))$ est la racine, alors la démarche est légèrement différente du cas général puisque le lien suffixe de la racine n'existe pas. Posons $head_{i-1} = a\delta$ et $x_{i..} = \delta\alpha$ avec $a \in \mathcal{A}$ et $\delta, \alpha \in \mathcal{A}^*$. En suivant le même raisonnement que dans le cas général (propositions A.4 et A.5), δ est préfixe d'un mot qui existe déjà dans l'arbre et si son locus n'existe pas alors $head_i = \delta$. On cherche donc $d \leftarrow$ RechercheRapide$(racine, \delta)$. Si d existait déjà alors $locus(head_i) \leftarrow$ LocusPréfixeCommun$(d, \alpha, IndiceTail)$, autrement $locus(head_i) \leftarrow d$. Il suffit alors, comme dans le cas général, de créer la feuille $locus(x_{i..})$ de père $locus(head_i)$. Le lien suffixe de $locus(head_{i-1})$ est le nœud d.

Si $locus(head_{i-1})$ est la racine, alors $pere(locus(head_{i-1}))$ n'existe pas et nous ne disposons d'aucune information nous permettant de court-circuiter la recherche de $head_i$. On fera alors comme dans l'algorithme en force brute :
$locus(head_i) \leftarrow$ LocusPréfixeCommun$(racine, x_{i..}, IndiceTail)$ suivi de la création de la nouvelle feuille $locus(x_{i..})$. Dans ce cas, il ne faut pas créer de lien suffixe pour $locus(head_{i-1})$ car il n'est pas défini pour la racine.

La fonction de construction de l'arbre des suffixes par la méthode de McCreight est présentée à la figure A.10. Dans cet algorithme, une structure de donnée d'arbre des suffixes est implicitement connue des fonctions InitArbre, LocusPréfixeCommun, RechercheRapide et CréeNœud qui la consultent et la modifient. Les $locus(\ldots)$ désignent les nœuds de cette structure implicite et *racine* désigne sa racine.

Pour illustrer le processus, reprenons l'exemple proposé par McCreight [McC76, Ste94] : $x =$ `BBBBBABABBBAABBBBB$` $\in \{$`A`, `B`, `$`$\}^*$.

Plaçons-nous à la fin de l'étape 13, nous devons insérer le suffixe $x_{14..} =$ `BBBBB$` (voir figure A.11). En reprenant les notations de l'algorithme, nous avons $\gamma =$ `B`, $\delta =$ `BB` et $\alpha =$ `BB$`. Le nœud $h = liensuf(locus(pere(head_{13})))$ est indiqué sur la figure. La recherche $d \leftarrow$ RechercheRapide(h, δ) nous amène droit sur un nœud qui existait déjà. La recherche de $head_{14}$ se poursuit donc par : $locus(head_{14}) \leftarrow$ LocusPréfixeCommun$(d, \alpha, IndiceTail)$ qui construit $locus(head_{14})$ sur l'arc joignant le locus du suffixe $x_{1..}$ à son père. La nouvelle feuille représentant $x_{14..}$ est alors créée.

A.2.4 Étude de complexité de l'algorithme

Nous supposons que l'alphabet est déterminé et contient un petit nombre de lettres. L'accès à un fils d'un nœud se fait donc en temps constant. Si ce n'est pas le cas, les résultats de complexité présentés ici doivent être multipliés par la taille $\#\mathcal{A}$ de l'alphabet ou par $\log \#\mathcal{A}$ selon la structure de données utilisée (voir section 1.3.4, partie I).

Théorème A.6 *L'algorithme de McCreight construit l'arbre des suffixes de $x = x_1 x_2 \ldots x_n$, avec $x_n \neq x_i, \forall i : 1 \leq i < n$, en temps $O(n)$.*

```
McCreightST(x, n)
 1  InitArbre()
 2  head_0 ← ε
 3  locus(head_0) ← racine
 4  Pour i ← 1 Jusque n
 5  Faire Si locus(head_{i-1}) = racine
 6          Alors locus(head_i) ← LocusPréfixeCommun(racine, x_{i..}, IndiceTail)
 7                j ← IndiceTail + i − 1
 8          Sinon Si pere(locus(head_{i-1})) = racine
 9                   Alors u ← head_{i-1}
10                         δ ← u_{2..}      /* On supprime la première lettre */
11                         h ← racine
12                         γ ← ε
13                   Sinon δ ← etiq(arc(pere(locus(head_{i-1})), locus(head_{i-1})))
14                         h ← liensuf(pere(locus(head_{i-1})))
15                         γ ← etiq(racine, h)
16                α ← etiq(arc(locus(head_{i-1}), locus(x_{i-1..})))
17                d = RechercheRapide(h, δ)
18                Si d n'a qu'un seul fils
19                   Alors              /* Cas où d vient d'être créé */
20                         locus(head_i) ← d
21                         j ← n − |α| + 1
22                   Sinon locus(head_i) ← LocusPréfixeCommun(d, α, IndiceTail)
23                         j ← IndiceTail + |γδ| + i − 1
24                liensuf(locus(head_{i-1})) ← d
25        CréeNœud(locus(head_i), j, n)
```

FIG. A.10 - *Algorithme de McCreight*

Preuve. Fixons-nous à l'étape i. Toutes les instructions se font en temps constant excepté les appels à RECHERCHERAPIDE et LOCUSPRÉFIXECOMMUN. Comptabilisons séparément les opérations effectuées par RECHERCHERAPIDE et par LOCUSPRÉFIXECOMMUN.

Pour plus de facilités dans les notations, posons $pere_i = fact(pere(locus(head_i)))$.

A.2.4.1 RECHERCHERAPIDE

Il s'agit de rechercher rapidement le locus de $\gamma\delta$ à partir du nœud h (voir figure A.8) : $d \leftarrow$ RECHERCHERAPIDE(h, δ).

Le temps de travail de la fonction est proportionnel au nombre de nœuds intermédiaires visités : le temps est en $O(|\delta|)$.

1er cas : h n'est pas la racine, c'est-à-dire que $pere_{i-1} \neq \epsilon$.

Dès lors, écrivons $O(|\delta|) = O(|\gamma\delta| - |\gamma|) = O(|\gamma\delta| - |pere_{i-1}| + 1) = O(|\gamma\delta| - |pere_{i-1}|)$.

A. Si RECHERCHERAPIDE ne crée pas de nouveau nœud, on a $|\gamma\delta| \leq |pere_i|$ et donc le temps de travail de RECHERCHERAPIDE est en $O(|pere_i| - |pere_{i-1}|)$.

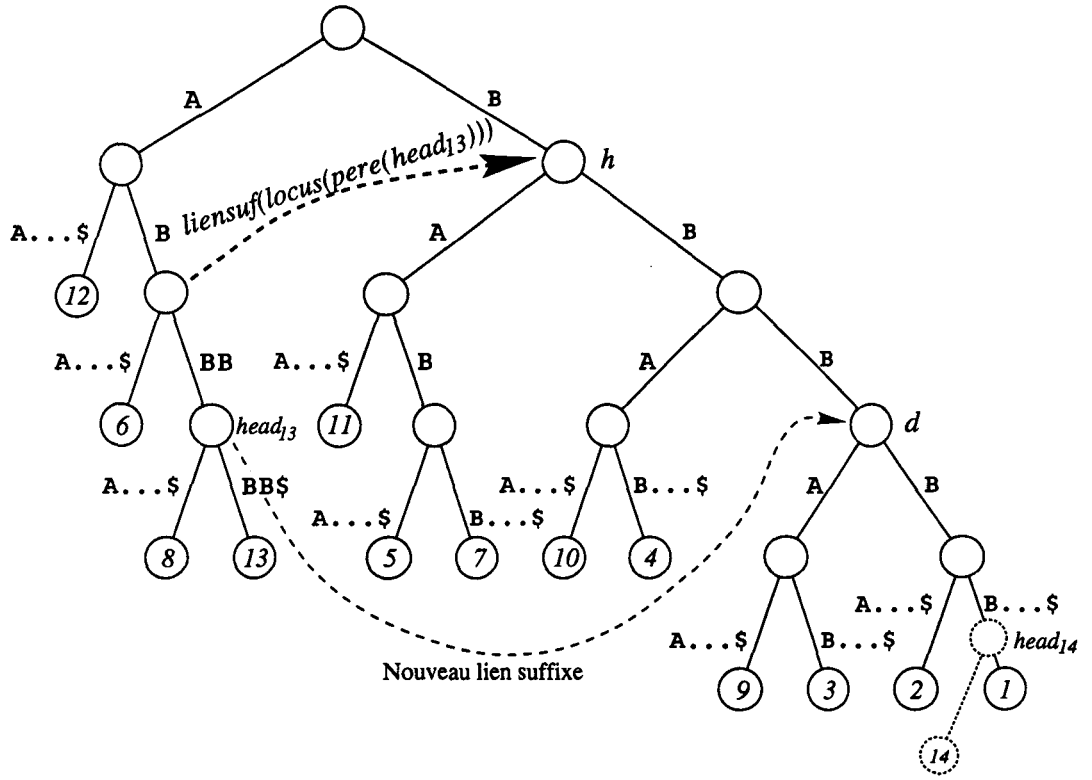


FIG. A.11 - *Arbre intermédiaire pour* $x =$ **BBBBBABABBBAABBBBB$** *lors de la* $14^{ème}$ *étape*

B. Si RECHERCHERAPIDE crée un nouveau nœud, alors $\gamma\delta = head_i$ et puisque le temps de travail pour passer de $locus(pere_i)$ à $locus(head_i)$ est constant, le temps de travail de RECHERCHERAPIDE est également en $O(|pere_i| - |pere_{i-1}|)$.

$2^{ème}$ cas : h est la racine. On a donc $|pere_{i-1}| = |\gamma| = 0$ et donc, on peut également écrire que le temps de travail de RECHERCHERAPIDE est en $O(|pere_i| - |pere_{i-1}|)$.

A.2.4.2 LOCUSPRÉFIXECOMMUN

Il s'agit de trouver le locus de $head_i$ lorsque RECHERCHERAPIDE a été appelée pour localiser d ou lorsque $locus(head_{i-1})$ est la racine.

1^{er} cas : $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(d, \alpha, IndiceTail)$.

Soit $\alpha = \alpha'.tail_i$. Le temps de travail de LOCUSPRÉFIXECOMMUN est en $O(|\alpha'|)$. Or $|\alpha'| = |\gamma\delta\alpha'| - |a\gamma\delta| + 1 = |head_i| - |head_{i-1}| + 1$ et donc le temps de travail est en $O(|head_i| - |head_{i-1}|)$.

$2^{ème}$ cas : $locus(head_i) \leftarrow$ LOCUSPRÉFIXECOMMUN$(racine, x_{i..}, IndiceTail)$.

Le temps de travail de LOCUSPRÉFIXECOMMUN est en $O(|head_i|) = O(|head_i| - |head_{i-1}|)$ puisque $|head_{i-1}| = 0$.

A.2.4.3 Temps global

Le temps consacré à l'insertion d'un suffixe $x_{i..}$ est donc en $O(|pere_i| - |pere_{i-1}|) + O(|head_i| - |head_{i-1}|) + O(1)$.

Le temps total de l'algorithme de McCreight est en

$$\begin{aligned}
&\sum_{i=1}^{n} O(|pere_i| - |pere_{i-1}|) + \sum_{i=1}^{n} O(|head_i| - |head_{i-1}|) + \sum_{i=1}^{n} O(1) \\
&= O(|pere_n| - |pere_0|) + O(|head_n| - |head_0|) + O(n) \\
&= O(n)
\end{aligned}$$

En vertu du fait que $|pere_n| = |pere_0| = |head_n| = |head_0| = 0$. □

Annexe B

Article [RDDD95] des actes de la conférence "International IEEE Symposium on Intelligence in Neural and Biological Systems"

A First Step Toward Chromosome Analysis by Compression Algorithms *†

E. Rivals
J-P. Delahaye
M. Dauchet
L.I.F.L.
Université de LILLE I
Villeneuve d'Ascq FRANCE 59655
email: rivals@lifl.fr

O. Delgrange

Service Informatique
Université de Mons-Hainaut
Mons BELGIQUE 7000
delgrang@lifl.fr

Abstract

In this paper, we use Kolmogorov complexity and compression algorithms to study DOS-DNA (DOS: defined ordered sequence). This approach gives quantitative and qualitative explanations of the regularities of apparently regular regions. We present the problem of the coding of approximate multiple tandem repeats in order to obtain compression. Then we describe an algorithm that allows to find efficiently approximate multiple tandem repeats. Finally, we briefly describe some of our results. Area: DNA sequence processing.

1 Introduction

Complex genomes contain numerous repeated sequences, the biological significance of which remains obscure. Recently, it has been shown that several human diseases are the result of changes in such sequences, thus it has become urgent to undertake a systematic study of their properties [10]. Certain diseases are the results of amplifications in the sequences which contain bases organised according to symmetrical elements, the DOS-DNA (DOS: defined ordered sequence, see [20]). DOS-DNA is characterized by the presence of palindromes, multiple tandem repeats, mono or oligonucleotide repetitions. Distortions may occur in these regular motifs. Experimental ([3], [7]) and computer ([19], [12], [18], [15], [9], [11], [5]) methods have been used to detect DOS-DNA. These methods use statistical (chi-square like) or combinatorial (matching like) methods.

*Partially supported by the "GDR 1029 Informatique et Genomes" and by the "Groupement de Recherches et d'Etudes sur les Génomes"

†Published in N.G. Bourbakis, editor, *First International IEEE Symposium on Intelligence in Neural and Biological Systems*, pages 233–239, Whashington DC, USA, may 1995. IEEE Computer Society Press.

In this paper, we use Kolmogorov complexity and compression algorithms as tools to study DOS-DNA (also see [6], [16]). This approach gives quantitative (a near-objective measure) and qualitative explanations of the regularities of apparently regular regions. The fact that this method is based on an approximation of the absolute notion of the Kolmogorov complexity, allows us to consider it as an alternative of purely statistical methods.

Kolmogorov complexity of a sequence s, $K(s)$ (see [14], [21] and [4]) is by definition the length of the shortest program (for a fixed computer) that prints s. This definition is invariant within a constant if one changes the universal computer. This is why $K(s)$ is considered as the absolute measure of the information content of s. Unfortunately $K(s)$ is not computable (no algorithm can compute the function $s \rightarrow K(s)$). But every compression algorithm actually gives an approximation of $K(s)$: the length $K_C(s)$ of the compressed sequence of s by a compression algorithm C is an upper bound of $K(s)$ and can thus be considered as a tentative measure of the information content of s (if the compression algorithm C is improved, the measure $K_C(s)$ converges to K(s)). With this idea, we can consider that DOS-DNA is DNA that is compressible, and that identification and localisation of DOS-DNA can be achieved by compression algorithms.

Some repeated sequences such as $s = CGACGACGACGACGA$ are clearly compressible (the use of the fact that s is 5 times CGA can be coded in a form that is shorter than s). Some other sequences such as $s' = CTACGAGACGATCGA$ (the same as s with one substitution, one deletion and one insertion) are not really interesting since the coding of their descriptions takes more digits than to copy them. Compression algorithms thus give an objective

criterion to distinguish real regularities from pseudo-regularities, and thus to distinguish DOS-DNA from other DNA. The best notion of random sequence is based on the complexity of Kolmogorov (see [14], [4]), the idea is roughly speaking: a random sequence is a sequence such that $K(s)$ is approximately the length of s.

In the first part, we present the problem of the coding of approximate multiple tandem repeats in order to obtain compression. In the second part, we describe an algorithm that allows to find approximate multiple tandem repeats efficiently. In the third part, we briefly describe some of our results. Another paper will give a more detailed biological interpretation of our results [8].

2 Coding Scheme for the Sequence Windows

Under the general title of the Coding Scheme, we present here the detail of the structure of the compressed sequence, in order to show that it is possible to restore the original sequence from it (i.e. that we do not forget any information) and also to provide an intuitive meaning of the regularities detected by the algorithm.

2.1 General Presentation

We address the problem of finding all the **approximate tandem repeats (ATR)** of a motif m within a text S. Here, *tandem repeats* means *side to side repetition of a particular motif m of a definite length.* We design several algorithms for different motif lengths. Here we explain the algorithm for motifs of length 3. The sequence S in which we want to detect Approximate Tandem Repeats (ATR) and the motif m of those ATR are both parameters of the algorithm.

The complete compression algorithm is divided in two main procedures. First, **the detection procedure** investigates the sequence to find the subwords that best match m^p for any p greater than a limit. The algorithm detects all subwords (streches of contiguous letters in S) of the sequence that resemble a tandem repeat of m, and aligns each of them with the word m^p to identify all differences. It then produces a list of differences' positions and types between the subword and m^p. The value of p is guessed (to each tandem repeat-like subword corresponds a different value of p).

Then, the coding procedure uses the result of the detection procedure to output a compressed version of the original sequence **without loss of information**. The goal of this part of the algorithm is to produce two files: the first one contains a list of codes to describe all the tandem repeats found in the sequence, the second is the concatenation (in the order of the sequence) of all characters that do not belong to the tandem repeats. This two file organization is adopted for experiment purpose: it is interesting to iterate compressors on the non coded part of the sequence, in order to look for regularities of a second order. Assume that the ancestor sequence of S enclosed some regularities, later some tandem repeats could have been independently inserted in those regularities, making them harder to locate. This is what we call second order regularities.

The definition of similarity is of major importance to understand the results of the algorithm. It is formalized in the next section, but we give an informal presentation of it now. A subword of S would be said similar to m^p, if it adopts the tandem repeats structure. In other words, if it has a subword that is similar to m nearly every $|m|$ [1] characters. This is quite different than to say that two words are similar because they have a specific character at position i, another specific character at position $i+1$, and so on ... without relation between positions. So, in our similarity definition, the errors are defined, their amount is limited relatively to a motif and not to the whole sequence. Notice that this specificity also simplifies the algorithm.

When do we consider a subword of a sequence as similar to m^p?

If we call the subword T, we shall view T itself as a succession of smaller subwords $T = t_1 \cdots t_p$. T is said to be similar to m^p iff:

- all t_i have either length $|m|$ or $|m|+1$ or $|m|-1$
- all t_i ought to be equal to m corrected by at most one substitution or [2] one deletion, and eventually followed by an insertion
- it begins and ends with an exact motif i.e. t_1 and t_p are exactly equal to m (this is later explained by technical consideration).

For instance, if $m = ACG$, either ACT or AT or $ACGT$ are qualified as similar to m but neither CGA nor CCC would.

This approach is motivated by the belief that specific regularities (like tandem repeats) need specific similarity definition, and ought not be searched with generally used and useful common distances. Both, the algorithm and its results are simplified by this view-point.

[1] $|m|$ denote the length of the text m

[2] It is an exclusive or.

Each type of possible motifs has an objective coding cost: a motif with one substitution or deletion costs 7 bits, and a motif with a substitution/deletion plus an insertion costs 14 bits.

2.2 Detail of the Coding Scheme

To achieve the traduction of the sequence in its compressed format, the coding procedure takes into account the following informations output by the detection procedure:

- the basic motif m
- the number of tandem repeats like subwords, later called **zones**
- their positions in the original sequence
- for each zone
  - the value of p, in the word m^p to which they best match
  - the optimal list (i.e. of lowest cost) of errors: position and type

Here comes an example of a window taken out of the 11th chromosome of Saccharomices cerevisiae from position 576,000 to 576,499, to illustrate our coding scheme. In this window, two ATRs of the motif CTG were found. The first one has been matched with the word made of 29 consecutives copies of CTG, and 9 errors were identified. A second zone of CTG tandem repeat-like was found, it is only 9 bases long and is error-free. Their respective beginning positions in the window are 112 and 235.

In the following schema, the beginning and the end of the zones are marked by < and >, while errors are denoted by *.

```
ATGAGTATTTTTCCAGAAATTACAAAACACAAATAAG

TAGAATGTGTTTAGTGTATTTGTAATTCATAGACTAT

TTATTGATGCTTATCTATTATACCTGGGGTTTCTGCT

TCGTCCTCGTCCTCGTCATCATCGTCGTCGTCGTCGT
<    *     *     *  *
CGTCGTCGTCGTCGTCGTCGTCGTCATCGTCGTCATC
                           *        *
GTCATCATCATCGTTGTTGTTTTCGTTCTTATCTTCT
   *  *  *    >
GTATCTTCTTCTTCGTCGTCGTCTTCTTCCTCATCCT
            <         >
CATCATCGTCCTCTTCCTCGTTATTTTTTGGGTACCC
ACCCAATTTCATAACGATTTTGTTAACAATCTCATTC
CAGTCCATCATACCCCCGAGACCAATATCACCATTTA
TATCCATAACTTTCTCAGACGGCTTGAAAACAGTGAC
CCAGGACATTTCCTCCTTTATCGTTTGTAGTTGCTTC
TTTGTCATCATAGAATTAAAAGTGCTACTCACCATAG
AAGGTGCCAAAAGAATGGG
```

The output code takes this format:

- one file contains the remaining sequence: the concatenation of the subwords that are not zones.

```
ATGAGT ... CTGCTT +
TGTTGT ... CTTCTT +
CTTCTT ... AATGGG
```

- the second file contains a structured list of binary codes that we write hereafter in comprehensive manner [3]. Several lines are used for a presentation purpose, because they correspond to natural "parts" of the complete code. The first part (the first line) tells us about the organization of the complete code: the motif and his length, the number of zones and so the number of parts of the code. The second part takes two lines to describe the first zone stucture, while the last part tells us about the second zone in the same way. A zone description begins with general informations about the zone followed by the list of descriptions of each error.

```
3; CGT; 1; 1;
length of the motif, the motif itself,
number of zones minus one,
choice between 3 or 4 bits code

112; 29; 9;
zone 1: relative window position,
#occurrences of m, #errors

(2,12);(2,12);(2,11);(2,11);
(14,11);(3,11);(2,11);(1,11);
(1,11);
zone 1 errors list:
(relative pos. of 1's motif,
error 1 type); etc.

36; 3; 0;
zone 2: relative window position
(235 -112 - 29 * 3),
#occurrences of m matched,
#errors (without errors list)
```

[3] All codes are given in integer or string format, followed by the ';' which means that we know where they end.

The major issue in the design of the code is to set the length of the codes for all the items. We just give a few interesting examples.

Some items' length are "arbitrarily" defined. For instance, the number of possible zones in a 500 bases long window takes two bits. To be compressed the sequence must contain at least one zone, so only the number of zone minus one is coded. So the number of coded zones is limited to 4.

The item position in window takes 9 bits and can code from 0 to 511, i.e. more than the length of the sequence. The number of repeats takes 8 bits because the $3 * 256 > 500$ and also because it is possible to find very long tandem repeats. On the contrary, the item number of errors has not a limited length but depends on the number of repeats. Actually, a short zone cannot afford too much errors, contrary to a long zone. Under 3 repetitions of m there is no way to gain any bit with or without any error. Until 5 repeats you cannot allow any error. Nearly under 8 repeats, the maximum error number is 3, etc. So, the length of the item's code is always determined dynamically depending on the number of repeats.

Each error is associated with a motif, so we give the relative position of the edited motif depending on the preceding error. The default code length is 3 bits, but sometimes like here, the interval between 2 errors exceeds 8 motifs. So we use a 4 bits code and indicate that choice by a 1-bit code in the first part of the code (choice between 3/4 bits code).

In our example, the gain worked out for the first zone is 80, for the second it is 0, so for the whole window it is: $80 + 0 - 11 = 69$, which corresponds to a compression rate of 7%.

3 An Heuristic Algorithm

In this section, we present the algorithm we design to locate the zones of ATR. This is a practical algorithm based on some heuristics and it produces satisfactory results from a biological point of view. We begin by a presentation of the specific problem, its restrictions and then we give a succint description of the algorithm.

3.1 Problem Definition

Let $S = s_1 s_2 \dots s_n$ be a text and $m = m_1 m_2 \dots m_l$ be a pattern. Both are words over the same alphabet A. We denote the length of a word w by $|w|$. Thus $|S| = n$ and $|m| = l$. We denote as $w[i, j]$ the subword of w, starting at position i and ending at position j. The i^{th} character of w is denoted by $w[i]$.

A *Perfect* Tandem Repeat (PTR) of a word m is a concatenation of m, p times ($p > 0$). We denote it by m^p. An ATR of a word m is a word "similar" to a PTR of m. Thus u is an ATR of m if it exists $p > 0$ so that u is "similar" to m^p. We clarify later what we mean by "similar".

Given the text S, the problem consists in locating non overlapping zones of ATR in S. Thus we must find a sequence of positions $(i_1, j_1, \dots, i_z, j_z)$ with $1 \leq i_1 < j_1 < \dots < i_z < j_z \leq n$ such that the words $S[i_1, j_1]$, $S[i_2, j_2]$, ..., $S[i_z, j_z]$ are ATR of some words $w_1, \dots, w_z$.

Most of the other combinatorial algorithms already developped are exhaustive approaches based on dynamic programming. Their time complexity rises up to $O(n^2 \log(n))$ [17], even for limited motif lengths [1], while our algorithm is in linear time complexity. Moreover, some methods only search for patterns repeated twice [17], [13].

3.2 Restrictions

We restrict the problem to the following points:

1. The alphabet for the text S and the patterns $w_1, \dots, w_z$ is $A = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ since the text is a subword of a DNA contig.

2. The text is short enough to consider that all the zones we search are ATR of a same pattern m. Thus given the text S, we must find the pattern m and the sequence of positions $(i_1, j_1, \dots, i_z, j_z)$ such that each zone $S[i_k, i_k]$ is an ATR of m.

   In practice, a long DNA sequence is splited into short length windows and the algorithm is applied independently on each window. Therefore, the zones of independent windows may be considered as ATR of different patterns.

3. The length l of the pattern is known at the beginning of the algorithm. In practice, the patterns are very short ($l \leq 5$) and the algorithm is tried for each possible pattern length.

3.3 Similarity Between a Word and a Tandem Repeat

Here, we define what we mean exactly by a word u is similar to the PTR m^p. The word u is similar to the PTR m^p if it can be written as $u = u_1 u_2 \dots u_p$, each subword u_k being obtained by a limited set of mutations from the word m. The length of the pattern m determines the set of authorized mutations. In the following, we present the algorithm for patterns of length 3, it can be easily adapted to another length.

The list of authorized mutations comes from experimental observations. If $m = \alpha\beta\gamma$, with $\alpha, \beta, \gamma \in A$, then the authorized forms of u_k are:

1. The word m itself:
   $u_k = \alpha\beta\gamma \Rightarrow |u_k| = 3$

2. The word m with the deletion of one base:
   $u_k = \alpha\beta$ or $u_k = \alpha\gamma$ or $u_k = \beta\gamma \Rightarrow |u_k| = 2$

3. The word m with the substitution of one base:
   $u_k = \mu\beta\gamma$ or $u_k = \alpha\mu\gamma$ or $u_k = \alpha\beta\mu \Rightarrow |u_k| = 3$

4. One of the three above forms plus the insertion of one base after the word:
   $u_k = \alpha\beta\gamma\rho \Rightarrow |u_k| = 4$
   $u_k = \alpha\beta\rho$ or $u_k = \alpha\gamma\rho$ or $u_k = \beta\gamma\rho \Rightarrow |u_k| = 3$
   $u_k = \mu\beta\gamma\rho$ or $u_k = \alpha\mu\gamma\rho$ or $u_k = \alpha\beta\mu\rho \Rightarrow |u_k| = 4$

Notice that all these forms keep at least 2 bases from the original word m.

3.4 Framework of the Algorithm

The goal of the algorithm is to detect the ATR zones in order to produce a compressed version of the sequence. Therefore, we apply some heuristics to obtain maximal compression gains.

The complete algorithm is decomposed into four steps:

1. The choice of the pattern m.

2. The localisation of all occurences of m in the text.

3. The construction of all PTR zones. This can be done from the informations collected at step 2.

4. The construction of the ATR zones using the localisations of the PTR zones.

We develop each of these steps.

3.4.1 Choice of the pattern m

For the experience, we only looked for one pattern: this restriction only applied to these experiments. To compress the sequence, we suppose that each ATR zone is equal to m^p modulo a set of mutations. Therefore we only code the set of mutations. In order to maximize the number of perfect matches between the text and the pattern in the ATR, we choose the pattern which has the maximum number of occurrences in the text.

Thus, the algorithm considers all subwords of length 3 of the text and count the number of occurrences of the 64 possible patterns. The pattern with the maximal count is choosen.

3.4.2 Localisation of all occurrences of the pattern

Each ATR zone must begin and end with a perfect match of the pattern. This is deduced from technical considerations: if a zone $u = u_1u_2 \dots u_p$ does not begin (resp. end) with a perfect match, i.e. $u_1 \neq m$ ($u_p \neq m$), then the ATR $u' = u_2 \dots u_p$ ($u' = u_1u_2 \dots u_{p-1}$) produces a better compression gain. For this we suppose that the coding of a mutation is more expensive than the coding of the word u_1 (u_p), for all possible u_1 (u_p).

To locate all the potential beginnings of ATR zones, we locate all the occurrences of the pattern.

3.4.3 Construction of PTR zones

At this stage, the algorithm gathers together all side by side occurrences of the pattern in order to construct the PTR zones. Since we want to maximize the compression gain, we always consider all PTR as included in some ATR. Thus, we must locate them.

3.4.4 Construction of the ATR zones using the PTR zones

The fourth step tries to join the PTR together, using several words $u_k \neq m$ which have the form described in subsection 3.3. Thus, all consecutive PTR zones are considered two by two, from left to right in order to construct longer ATR zones.

Given $S[i_a, j_a]$ and $S[i_b, j_b]$ two PTR zones, how do we join them together ?

The idea is to lengthen the zone $S[i_a, j_a]$ to the right while it is possible. Thus, at each step we try to increase the right limit j_a of the first zone. When $j_a = i_b - 1$ then the two zones have been joined.

At each step, we denote by u_k the word $S[j_a+1, i_b-1]$ which separates the first zone from the second one and by $l_{sep} = i_b - j_a - 1$ its length. Three cases must be considered:

Case 1: $l_{sep} < 2$

The two zones cannot be joined because their separation is not long enough. There is no mutation of m that would produce a word of length 1.

Case 2: $2 \leq l_{sep} \leq 4$

If the word u_k approximately matches to m, the two zones are joined in one step. If there are several kinds of mutations which can produce u_k from m, then the cheapest one (from a compression point of view) is chosen. If u_k cannot be

obtained by any kind of mutation then the two zones cannot be joined.

Case 3: $l_{sep} > 4$

The two zones cannot be joined in one step. Therefore, among the 13 authorized mutations of m, the algorithm selects those which produce a prefix of u_k.

If such mutations do not exist, then the two zones cannot be joined.

Otherwise, the algorithm chooses the mutation which produces the greatest gain, i.e. the mutation which generates the longest word $u_k[1, o]$ (with $o < l_{sep}$) for the lowest cost. Thus the first zone is lengthened and becomes the subword $S[i_a, j_a + o]$.

When two zones cannot be joined, then the first one is restored in its original form, i.e. in the longest for which it begins and ends with a perfect match of m.

4 Experiments and Results

With the exhaustive sequencing of many complete eukaryotic chromosomes, questions about chromosomal organisation and evolution are now possible to address. Long after the pioneer steps of the discovery of isochores by Bernardi [2], compositional biases and more complex sequence inhomogeneities issues have already been approached by statistical analysis [11].

The approach of compression algorithms brings a new class of tools to detect specific types of regularities with an informational significance given by the compression rate. The study of distribution of regularities and their site-specificity are relevant to both organizational chromosomal constraints and large size evolutionnary events.

In these experiments, we use algorithms to identify ATR based on small motifs (mono-, di- and trinucleotides) on complete chromosomes of Saccharomyces cerevisiae. As ATR are local regularities, the experience applies the 3 algorithms to 500-base long contiguous windows [4], *only the best compression result on each window is given.* Here we focus only on the 11th chromosome of SC for place limitation and also because other ones entail quantitatively similar results.

Calculus of the compression gain: in an obvious way one base needs 2 bits to be coded, so each 500-base long window has an initial length of 1000 bits. Two files result from a compression:

- one contains the remaining sequence of length $n \leq 500$ that is encoded in $2n$ bits
- the code-file of length p in bits, includes the description of the removed ATR.

So the compression gain in bits is: $1000 - (2n + p)$ and the compression (or reduction) rate is given by: $\frac{100*(1000-(2n+p))}{1000}$. All the results are now expressed by compression gain in bits.

Chromosome 11 is divided into 1332 windows, 113 of which have been compressed by at least one algorithm. The mean gain is about 15 bits. We recapitulate the windows count as well as the mean and max gain values by algorithms.

Windows count on chromosome 11 of Saccharomyces cerevisiae:

Motif Length	#	%	Gain	
			Avg	Max
1	79	69	7	46
2	14	12	19	89
3	20	17	41	292

Most of the time, a motif of length 1 gives the best compression (79 windows, i.e. 69 % of the compressed segments) with a mean gain of 7 bits ; while algorithms for di- and trinucleotides motifs perform fewer but better compressions: respectively 12 and 17% of the compressed windows with an average gain of 19 and 41 bits and a maximum gain up to 89 and 292 bits. They probably do not reveal the same sort of biological signals.

Hereunder, the figure 1 displays at each beginning position of a segment along the sequence, a vertical line from the x-axis of height: the gain of the corresponding segment.

The ATR zones avoid no cluster of the sequence, and are distributed from 3′ all the way down to 5′ end. Occurrence sites of the tandem repeats are found in all sort of regions:

- either in telomeric extremities of the chromosome: e.g. a telomeric tandem repeat of AC of gain 32 (37 repeats with 11 mutations)
- or bording ORF or protein coding regions: an ATR of AT is positionned near dihydroororate dehydrogenase locus with gain 29 (19 repeats with 3 mutations)
- or inside coding regions: in the locus of YKL202w hypothetical protein a 114 repeats of TTC with 51 mutations of gain 292 (which can also be partly seen as a 154 poly-T with 55 mutations of gain 6).

[4] also called segments.

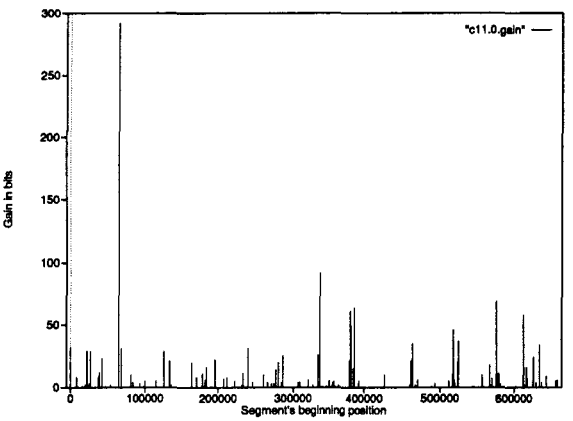


Figure 1: Position and gain of the compressed segments along Chromosome 11

References

[1] G. Benson and M.S. Waterman. *A Method for Fast Database Search for All k-nucleotide Repeats.* 1994.

[2] G. Bernardi. The Isochore Organization of the Human Genome. *Journal of Molecular Evolution*, Vol. 28, pp. 7-18, 1989.

[3] A. Behn-Krappa, J. Mollenhauer, and W. Doerfler. Triplet Repeat Sequences in Human DNA Can Be Detected by Hybridization to a Synthetic (5'-cgg-3')17 Oligodeoxyribonucleotide. *FEBS Letters*, Vol. 333, pp. 248-250, 1993.

[4] J-P. Delahaye. *Information, complexité et hasard.* Hermès, Paris, 1994.

[5] E. Gilson, W. Saurin, D. Perrin, S. Bachelier, and M. Hofnung. *Nucleic Acids Research*, Vol. 19, pp. 1375-1383, 1991.

[6] Stéphane Grumbach and Fariza Tahi. Compression of DNA Sequences. In *Data Compression Conference*, 1993.

[7] H. Hummerich, S. Baxendale, R. Mott, S.F. Kirby, M.E. MacDonald, J. Gusella, H. Lehrach, and G.P. Bates. Distribution of Trinucleotide Repeat Sequences accross a 2Mbp Region Containing the Huntington's Disease Gene. *Human Molecular Genetics*, Vol. 3, pp. 73-78, 1994.

[8] A. Henaut, M.O. Delorme, E. Ollivier, E. Rivals, O. Delgrange, J-P. Delahaye, and M. Dauchet. Extended DOS-DNA Analysis of Yeast Chromosomes. unpublished.

[9] J. Han, C. Hsu, Z. Zhu, J.W. Longshore, and W.H. Finley. Over-representation of the Disease Associated (CAG) and (CGG) Repeats in the Human Genome. *Nucleic Acids Research*, Vol. 22, pp. 1735-1740, 1994.

[10] A. Henaut and E. Ollivier. DOS-DNA Occurs every 35kb in Chromosomes of Saccharomyces Cerevisiae. unpublished.

[11] S. Karlin, B.E. Blaisdell, R. Sapolsky, L. Cardon, and C. Burge. Assessments of DNA Inhomogeneities in Yeast Chromosome III. *Nucleic Acids Research*, Vol. 21, pp. 703-711, 1993.

[12] M-Y. Leung, B.E. Blaisdell, C. Burge, and S. Karlin. An Efficient Algorithm for Identifying Matches with Errors in Multiple Long Molecular Sequences. *Journal of Molecular Biology*, Vol. 221, 1367-1378, 1991.

[13] G.M. Landau and J.P. Schmidt. *An Algorithm for Approximate Tandem Repeats.* Proc. 4th Symp. Combinatorial Pattern Matching. Springer-Verlag, LNCS, Vol. 648, pp. 120-133, 1993.

[14] M. Li and P.M.B. Vitanyi. *Introduction to Kolmogorov Complexity.* Springer-Verlag, 1993.

[15] A. Milosavljevič and J. Jurka. Discovering Simple Dna Sequences by the Algorithmic Significance Method. *CABIOS*, Vol. 9(4), pp. 407-411, 1993.

[16] E. Rivals, O. Delgrange, M. Dauchet, and J-P. Delahaye. Compression and Sequence Comparison. DIMACS Workshop on Sequence Comparison, nov. 1994.

[17] J.P. Schmidt. *All Shortest Paths in Weighted Grid Graphs and its Application to Finding All Approximate Repeats in Strings.* 1994

[18] P. Salomon, J.C. Wootton, A.K. Konopka, and L. Hansen. On the Robustness of Maximum Entropy Relationships for Complexity Distributions of Nucleotide Sequences. *Computer Chem.*, Vol. 17, pp.135-148, 1993.

[19] D. Tautz, M. Trick, and G.A. Dover. Cryptic Simplicity in DNA is a Major Source of Genetic Variation. *Nature*, Vol. 332 pp. 652-656, 1986.

[20] R.D. Wells and R.R. Sinden. *Genome Rearrangement and Stability*, chapter Defined Ordered Sequence DNA, DNA Structure, and DNA-directed Mutation. Genome Analysis. Cold Spring Harbor Laboratory Press, 1993.

[21] P.H. Yockey. *Information Theory and Molecular Biology*. Cambridge Univ. Press, 1992.

Annexe C

Article [RDD+97] de la revue CABIOS

CABIOS

Vol. 13 no. 2 1997
Pages 131–136

Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences

E.Rivals[1,4], O.Delgrange[2], J.-P.Delahaye[1], M.Dauchet[1], M.-O.Delorme[3], A.Hénaut[3] and E.Ollivier[3]

Abstract

Motivation: *Compression algorithms can be used to analyse genetic sequences. A compression algorithm tests a given property on the sequence and uses it to encode the sequence: if the property is true, it reveals some structure of the sequence which can be described briefly, this yields a description of the sequence which is shorter than the sequence of nucleotides given* in extenso. *The more a sequence is compressed by the algorithm, the more significant is the property for that sequence.*
Results: *We present a compression algorithm that tests the presence of a particular type of* dos*DNA (defined ordered sequence-DNA): approximate tandem repeats of small motifs (i.e. of lengths <4). This algorithm has been experimented with on four yeast chromosomes. The presence of approximate tandem repeats seems to be a uniform structural property of yeast chromosomes.*
Availability: *The algorithms in C are available on the World Wide Web (URL: http: www.lifl.fr rivals Doc RTA).*
Contact: *E-mail:rivals@lifl.fr*

Introduction

A compression algorithm detects significant patterns in a text, if it encodes such patterns, and achieves by the way a concise description of the whole text. The shorter the output description is, the more significant the patterns are. Consequently, for a given text, the significance of the detected patterns is measured by the compression rate of the text, i.e. the ratio between the output and original description sizes.

In more general terms, a compression scheme detects a property of the text, and this property is taken into account to produce a new description. If the output description is longer than the original one, then the text cannot be compressed with this property. In other words, the property is irrelevant for the text. The more the text is compressed, the more it verifies the studied property. A property could be the repetition of a pattern, but also a statistical bias in the use of the alphabet symbols (statistical compressors like Huffman encoding treat this case).

The general ideal that links significance and the reduction of the description size is also called algorithmic significance (Milosavljević and Jurka, 1993a,b), or the minimal length description principle in machine learning (Rissanen, 1985), or Occam's Razor principle in Kolmogorov complexity and information theory fields (Cover and Thomas, 1991; Li and Vitanyi, 1993; Delahaye, 1994). This principle is used to justify the minimal edit distance in sequence alignments and the parsimony principle in the construction of phylogenetic trees.

Milosavljević and Jurka (1993a) prove a theorem that states to what significance level the text verifies a property, depending on how much it is compressed. To test this algorithmic significance method, Milosavljević and Jurka (1993a) designed an algorithm based on a compression scheme (Ziv and Lempel, 1978), which identifies repeated subwords in DNA sequences.

Grumbach and Tahi (1994), Rivals *et al.* (1995) and Rivals (1996) advocate the use of more efficient compression algorithms for genetic sequence analysis.

There exists biological evidence that genomes include sequences of low complexity, such as various distant repeats (LINEs or SINEs elements), tandem repeats of an oligonucleotide (e.g. mini- or micro-satellites), genetic palindromes, etc. Such simple loci have recently been named *dos*DNA (defined ordered sequences), because their bases are organized according to symmetrical elements (Wells and Sinden, 1993). Mutations may occur in the sequence of those repeats. For a long time, inverted repeats were studied essentially for their structural properties and simple sequence repeats were used for the construction of genetic maps, with no consideration of their biological meaning. A fact made biologists and computer scientists have new interest in *dos*DNA: it has been shown that *dos*DNA was involved in specific mutagenic events that can lead to human diseases (Wells and Sinden, 1993).

Although complete chromosomes of eukaryotic organisms have been sequenced, neither the repartition of approximate tandem repeats (ATR) nor their importance in chromosome organization have been addressed. The repartition of various types of *dos*DNA along the third chromosome of yeast has

Laboratoire d'Informatique Fondamentale de Lille, CNRS URA 369, Villeneuve d'Ascq 59655, France
[2]Université de Mons-Hainaut, Mons, B7000, Belgium
[3]Université de Versailles Saint-Quentin, Versailles 78035, France
[4]To whom correspondence should be addressed

been studied by statistical methods (Karlin *et al.*, 1993). Han *et al.* (1994) explored the presence of trimer ATR only in sequences of human genes. Our objective is to focus on the identification, in yeast chromosomes, of significant ATR of mono-, di- and trimer motifs, in order to gather information about expansion of short motifs ATR in eukaryotes, and to determine their influence on chromosome organization.

A model of DNA sequence evolution underlies the algorithm we designed. In a DNA sequence, duplication of a short motif may create a tandem repeat of this motif, which can later be altered by point mutations (like substitutions, insertions and deletions). This model of an evolutionary mechanism, that we do want to be restrictive, is the core of our compression tool for sequence analysis.

When the algorithm is applied to a sequence, it locates and encodes ATR of a given motif u (the length of u is <4). Such zones made of an ATR of u are described by a binary code, in which the evolution model appears as a possible creation process for those zones. In fact, a code for a zone means: 'at position i in the sequence, the motif u has been replicated n times, afterwards the corresponding segment has been modified by the following mutations (the list of point mutations is then given). If the text is effectively compressed, then this creation process is a better explanation of the ATR appearance than a random generation process. In other words, those ATR zones are not random.

Our work consists of (i) the design of a compression algorithm based on Locate_ATR(t, u) which looks for ATR of a motif u (also denoted by u-ATR) in a text t and (ii) the practical application of this algorithm to the sequence of four complete yeast chromosomes. It is the first time, to our knowledge, that a compression algorithm has been conceived specifically to fit the requirements of a genuine biological application like *dos*DNA identification. The algorithm and the encoding of ATR zones are explained in the next section, while the Discussion gives a short review of the results from our experiments on yeast chromosomes.

Compression scheme for ATR

Here we present a compression algorithm which manages to compress a text if it contains significant segments that are ATR of a short motif. In fact, we conceive three similar algorithms, one for each studied motif length: 1, 2, or 3. We explain the algorithm for trimer-ATR. In the following, let t and u be texts over the alphabet A, u is called the motif. The algorithm includes two procedures: Locate_ATR(t, u), which locates ATR zones in the text, and an encoding procedure which outputs a new version of the text. In this output version, the u-ATR zones of t are described in special code that is not simply the translation in bits of their sequence, but indicates how to rebuild them. This allows the decompression process to recover the original sequence, the compression scheme is said to be lossless. We describe this encoding by an example. First we explain precisely what means approximate for the zones the algorithm looks for. Finally, we detail Locate_ATR and give its complexity.

An alignment of two sequences, where n is the length of the longest one, costs $O(n^2)$, but the alignment of a sequence with the sequence of a tandem repeat of a motif u (i.e. u^p) only costs $O(n)$ using the wraparound dynamic programming procedure (Fischetti *et al.*, 1992). Benson and Waterman (1994) designed an algorithm that selects ATR according to their similarity score, if the ATR are aligned with a periodic pattern of the motif (wraparound dynamic programming is used for that purpose). In Benson and Waterman (1994), ATR zones are output if their score is greater than an arbitrary threshold value, which is given as a parameter. In our method, the compression gain is required to be greater than the 0 bits threshold. The user does not have to choose this value. Moreover, it is less arbitrary [than in the method of Benson and Waterman (1994)] because, in terms of compression rate, there are not many ways to design an efficient code.

Approximate tandem repeats

A u-ATR is a text that is similar to a perfect tandem repeat of the motif u (a u-PTR). Our algorithms use a formal definition for this notion of approximation: a text is a u-ATR if it can be decomposed into a sequence of words that are all similar to u. As u is a parameter of Locate_ATR(t, u), the words which are similar to u are given by a finite set $Sim(u)$. We give the definitions for a u-PTR, $Sim(u)$ and a u-ATR.

Definition 1: t is a perfect tandem repeat of u or u-PTR if: $\exists\, p > 1 : t = u^p$

We define $del(u)$ as the set of all words obtained by applying a deletion to u; respectively, $sub(u)$ is the set of words obtained thanks to a substitution. $ins(u)$ contains any word made from a word of $\{u\} \cup sub(u) \cup del(u)$ followed by a single-letter insertion. Then $Sim(u)$ is the union of those sets and the singleton $\{u\}$.

Definition 2:

$$del(u) = \{l \in A^*, |l| = |u| - 1, \exists j \in [1, |u|] : \forall i \in [1, j-1], l[i] = u[i] \; \forall i \in [j, |u| - 1], l[i] = u[i+1]\}$$

$$sub(u) = \{l \in A^*, |l| = |u|, \exists j \in [1, |u|], l[j] \neq u[j], \forall i \neq j \; l[i] = u[i]\}$$

$$ins(u) = \{v = la : l \in \{u\} \cup sub(u) \cup del(u); a \in A\}$$

then $Sim(u) = \{u\} \cup sub(u) \cup del(u) \cup ins(u)$

For instance, if the motif u is GTA and

$$L = \{GTA, GT, GA, TA$$
$$ATA, CTA, TTA, GAA, GCA,$$
$$GGA, GTC, GTG, GTT\}$$

we have $Sim(GTA) = L \cup (L \cdot a)$ where $a \in \{A, C, G, T\}$.

Our neighbourhood definition differs from the classic definition (which allows a substitution or an indel at any position) by the way it deals with insertions, which can be a second mutation on the motif. In practice, the algorithm produces nearly the same results with our neighbourhood definition as with the classic definition. It proceeds from the compression threshold: if words that contain two mutations are used too often in the factorization, the zone gain becomes negative.

Definition 3: Let u, z be texts over A. z is an ATR of motif u or u-ATR if:

$$\exists p > 0, \exists v_1, \ldots, v_p \in Sim(u): \; z = v_1 v_2 \ldots v_p$$

Our algorithm detects compressible ATR; later, we see that such ATR must begin and end with exact motifs (i.e. v_1 and v_p must be equal to u). The process of finding a decomposition with words in $Sim(u)$ for a text z, or the actual decomposition that results from this process, are both called a factorization of z.

Encoding of sequence

When the ATR zones of a text t are known, the encoding procedure outputs a compressed version of the text that is divided into two parts. First, a description of the ATR zones is given by a specific code, then other segments in the sequence are concatenated and encoded by the simple translation of each base in a two-bit code. Therefore, we only detail the specific code employed to represent ATR zones.

If the complete output version, i.e. remaining sequence plus the ATR zones code, is shorter than the original sequence, then the global gain of t is positive. The gain of t is the difference in bits between the length of those versions of t. As each ATR segment detected in t corresponds to a specific part of the whole code, we define in the same way a zone gain. This is used to guide the algorithm Locate_ATR(t, u). Note that the gain of t is the sum of the gain for each zone, minus a constant number of bits.

We will see in the Discussion that the compression algorithm is applied to consecutive 500 bp windows of a chromosome. The value of 500 bp does not have much influence on the results of our algorithm and is chosen because ATR are local structures in a sequence. In order to encode ATR zones as briefly as possible, the encoding must take into account this 500 bp value. In this case, the most economical encoding is to use a fixed-length format for many of the code items.

The code for the description of the ATR zones of a text t

Fig. 1. Example of two ATR in a sequence.

contains the following. (i) global information on t: the motif length, the chosen motif u, the number of u-ATR zones. (ii) For each ATR zone: the offset position of the zone first nucleotide, the number of repeats of u given by the factorization, the number of mutations also determined by the factorization, the explicit list of all mutations. (iii) For each mutation in a zone list, a couple of integers indicate: the relative index of the motif on which the mutation is located, e.g. 2 is decoded as 'from current position, advance by two copies of the motif', the mutation itself given by an index in a look-up table that records all possible mutations.

Figure 1 gives an instance of a sequence which is compressible. ATR zones of motif TCG are in upper case, and Figure 2 shows the corresponding encoding for the ATR zones.

Gain of a zone is computed by the difference in bits between the length of the encoding of a zone (for instance the triplet *36*; *3*; *0*; for zone 2 in Figure 2) and its original sequence. Indeed, *Gain* is the number of bits saved by coding a zone using a u-ATR as compared to coding it letter by letter. Therefore, *Gain* is a linear function in the length of the zone in nucleotides and in its number of mutations. Each mutation must be written in the *errors list* of the zone and costs a fixed number of bits.

For trimer motifs, the code of a mutation takes 7 bits, the code for information about the ATR zone, except the mutation list, costs altogether 22 bits. Thus, if m is its number of mutations, the length of the code for the ATR zone is $7m + 22$ in bits. The standard description of the zone sequence, using two bits for each base, takes $2n$ bits, if n is the length of the zone. To calculate the gain of a zone z, we use the following equation which substracts the encoded description length from the original description length:

$$Gain(z) = 2n - 7m - 22$$

Localization of ATR zones

For a given motif, the goal of the algorithm Locate_ATR is to

3 TCG 1
length of the motif, the motif itself, number of zones minus one

112; 29; 9
zone 1: relative window position, #words in the factorization, #errors

2,12);(2,12);(2,11);(1,11);(4,11);(3,11);(2,11);(1,11);(1,11);
zone 1 errors list: (relative number of error 1's motif, error 1 type), etc

46; 1; 0;
zone 2: relative window position (235 - 112 - 29 * 3), #words in the factorization, #errors (without errors list, of course)

Fig. 2. Encoding of ATR zones for the sequence in Figure 1. The code is italicized and is given in a readable format instead of the binary format.

locate its ATR zones that are compressible, i.e. zones with positive gain. As u is a parameter of Locate_ATR(t, u), we simply write ATR or PTR instead of u-ATR or u-PTR.

Notation 1: $t[i, j]$ denotes a substring of t from position i to position j inclusive (with $i \leq j$). We denote by Locate_ATR(t, u) the result of our algorithm. $t[i, j] \in$ Locate_ATR(t, u) if: $t[i, j]$ is a u-ATR and $Gain(t[i.j]) \geq 0$: it does not exist $i \leq k < l \leq j$ such that $(i, j) \neq (k, l)$ and $t[k.l]$ is a u-ATR and $Gain(t[k, l]) > Gain(t[i.j])$ (non-inclusive condition); it does not exist $k \leq i \leq j \leq l$ such that $(i.j) \neq (k.l)$ and $t[k.l]$ is a u-ATR and $Gain(t[k.l]) > Gain(t[i.j])$ (maximality condition).

Only potentially compressible zones are examined by the algorithm. For small motif lengths, compressibility requires the following necessary condition for a zone z: its factorization must begin and end with an exact motif. This is because a mutation that must be written in the errors list costs too much in bits (i.e. costs more than what is obtained when the nucleotides of one motif are encoded: for a trimer, it costs 3 nuc. $\times$ 2 = 6 bits and a mutation costs 7 bits). By extension, an ATR that belongs to Locate_ATR(t, u) must begin and end with a PTR which is as large as possible (i.e. maximal PTR) because of the maximality condition [proofs of those properties can be found in Rivals (1996), but are not given here]. For clarity, any word u^p with $p \geq 1$ is said to be a u-PTR.

Given the previous necessary condition, the algorithm looks for ATR that are between two separate maximal PTR. Then the first step of the algorithm is to locate all maximal PTR in t, which is done in one pass over t. It builds a list of PTR which are ordered on their beginning position (note that they do not overlap). Let $r_1, \ldots, r_k$ denote the consecutive maximal PTR of t, w_i the subword between r_i and r_{i+1} for any i.

The main idea is the following. For each PTR in the list, the algorithm tries to 'join' the next PTR on its right, by attempting to factorize the subword that lies in between. If the factorization is possible, the subword $r_i \cdot w_i \cdot r_{i+1}$ forms an ATR zone. If its gain is (i) positive and (ii) greater than $Gain(r_i) + Gain(r_{i+1})$ then $r_i \cdot w_i \cdot r_{i+1}$ becomes the current zone and the algorithm attempts to join it with r_{i+2}, and so on. This procedure achieves a maximal right extension of an ATR, if repeated while right junctions are possible and while the gain grows. We call this procedure *Extension_Max* and give its pseudo-code below.

```
EXTENSION_MAX
Begin
  w ← r_i
  next ← i+1
  While (Junction of w and r_next is possible
    and Gain(w.w_next-1.r_next) > Gain(w) + Gain(r_next)) Do
    w ← w.w_next-1.r_next
    next ← next + 1
  Done
End EXTENSION_MAX.
```

Fig. 3. Algorithm *Extension_Max*.

Therefore, to compute Locate_ATR(t, u), it suffices to apply *Extension_Max* to each PTR in the order of the list (i.e. to each r_i for i ranging from 1 to k). The zone being built is joined to the PTR on its right, which is deleted from the list, until factorization is impossible or the gain decreases.

An important procedure is the factorization. As $Sim(u)$ is a given finite set, an optimal factorization procedure exists (Bell *et al.*, 1990; Crochemore and Rytter, 1994), but to build the quickest algorithm possible, we use an on-line heuristic factorization procedure, which provides good results.

It has been decided to implement a heuristic algorithm which is practically close to the optimal algorithm. Let us denote by Locate_ATR_H(t, u) the heuristic algorithm. In Locate_ATR_H(t, u), the heuristic chosen is to perform a greedy factorization of a subword between two PTR instead of an optimal one. Our implementation decision takes into account the requirement to deal with a large amount of data as quickly as possible (the largest yeast chromosome exceeds 800 000 bp and our tests involve >2 300 000 bp of DNA sequences).

The following heuristic is applied when factorizing large segments (> 4 bp) between two PTR. The segment is factorized into a sequence of words in $Sim(u)$. When a mutation is found at a given position, only the three or four nucleotides around are examined to decide which is the appropriate word in $Sim(u)$: then the costliest word of $Sim(u)$ is chosen accordingly. In fact, the decision is taken according to the local context and is never called into question later in the segment factorization.

Algorithm complexity

Proposition 1: The time complexity of Locate_ATR_H(t, u) is linear in the length of t.

Proof: The complexity of Locate_ATR_H(t, u) comes from the main loop in which *Extension_Max* is applied to each PTR denoted r_i. The more complex procedure is the joining of two consecutive zones, especially the computation of the factorization of each w_i, which is linear in the length of w_i. The junction between two consecutive zones is only computed once, because: if it is possible, the zone is joined to the PTR on its right, which is removed from the list, and the index *next* is incremented; otherwise a new extension process starts with the following PTR as a new zone.

Then we have:

$$Complexity(\text{Locate\_ATR\_H}(t, u)) \leq \sum_{i=1}^{k} |w_i| < |t|$$

Discussion

Our algorithms have been applied to chromosomes II (807 188 bp), III (315 357 bp), VIII (562 638 bp) and XI

Table I. Global results of the experiments on order-1 Markov sequences. Each pseudo-random sequence is 10 times longer and has the same dimer composition as the corresponding chromosome. The percentage of compressed windows, the maximum and average gains in bits are reported for each motif length

Motif length	% compressed windows	Maximum gain	Average gain
1	0.35	12	2
2	0	0	0
3	0	0	0

(666 448 bp) of yeast. All those sequences represent ~17% of yeast DNA. A main reason justifies this choice: yeast is the only eukaryotic organism (the human is also a eukaryote) for which we know complete chromosomes sequences.

Before testing, each sequence is cut up in adjacent windows of 500 bp [because we are interested in comparing our results with those in Ollivier *et al.* (1995) which also copes with 500 bp long windows]. Each window undergoes the three algorithms, one for each motif length. The compression algorithm is run with the most frequently occurring subword in the window as the motif parameter, except for the mononucleotide algorithm in which each possible motif is tried. Only compressed windows are reported.

For dimer and trimer motifs, we also tried to rerun the algorithm on already compressed windows with another motif as parameter. Except for a very few, the algorithms do not manage to compress these windows a second time. This reveals that in >90% of cases, a window contains at most ATR of one motif (results not reported here).

ATR which bridge two windows are not reported, but their left (respectively right) part which falls into the window on the left (respectively on the right) is reported if it is long enough (i.e. long enough to be compressible).

To validate our results, we also performed the same tests on randomly generated sequences that follow an order-1 Markov model. For each chromosome, we generated a 10 times longer sequence with the same dimer composition. Table I shows, for each motif length: the percentage of compressed windows, the average and the maximum compression gains. Only ATR of mononucleotidic motifs give rise to compression with low gains. The huge differences compared with the tests on real chromosomes enlighten the statistical significance of our method.

Table II shows for each algorithm in all chromosomes: (i) in column Comp. the number of compressed windows; (ii) in column % comp the ratio of the number in the previous column over the total number of compressed windows by the three algorithm; (iii) in column % total the ratio over the total number of windows. It is interesting to note that the presence of ATR is quantitatively uniform on all chromosomes. There are quantitative and qualitative differences between the monomer-ATR zones and di- or trimer-ATR zones: the ratio over the total number of windows reaches 6% for monomers and is around 1 or 2% for dimers and trimers. Moreover, the gains are higher, on average, for compression using dimers and trimers than using monomer ATR zones. For trimer motifs in all chromosomes, 60% of the compressed windows reveal a compression rate > 2%. Complete results and deeper analysis are available in Rivals (1996). All algorithms are written in C and are available on the World Wide Web (URL: http://www.lifl.fr/rivals/Doc/RTA/). On a Sun-Sparc 5 workstation, the whole experiment takes 5 min and 48 s for yeast chromosome XI.

References

Bell.T.C., Clearly, J.G. and Witten.I.H. (1990) *Text Compression*. Prentice Hall.

Benson.G. and Waterman.M.S. (1994) A method for fast database search for all k-nucleotide repeats. In *Symposium on Multiple Alignment*. USA IEEE Computer Society Press, Seattle.

Cover.T.M. and Thomas.J.A. (1991) *Information Theory*. Wiley Series in Telecommunications. A. Wiley-Interscience.

Crochemore.M. and Rytter.W. (1994) *Text Algorithms*. Oxford University Press.

Delahaye.J.-P. (1994) *Information, complexité et hasard*. Hermès, Paris.

Fischetti.V., Landau.G., Schmidt.J. and Sellers.P. (1992) Identifying periodic occurrences of a template with applications to protein structure. In *3rd Annual Symposium of Combinatorial Pattern Matching*, pp. 111–120.

Grumbach.S. and Tahi.F. (1994) A new challenge for compression algorithms: genetic sequences. *J. Inf. Process. Manage.*, **30**, 875–886.

Han.J., Hsu.C., Zhu.Z., Longshore.J.W. and Finley.W.H. (1994) Over-representation of the disease associated (CAG) and (CGG) repeats in the human genome. *Nucleic Acids Res.*, **22**, 1735–1740.

Karlin.S., Blaisdell.B.E., Sapolsky.R., Cardon.L. and Burge.C. (1993) Assessments of DNA inhomogeneities in yeast chromosome III. *Nucleic Acids Res.*, **21**, 703–711.

Li.M. and Vitanyi.P.M.B. (1993) *Introduction to Kolmogorov Complexity*. Springer-Verlag.

Milosavljević.A. and Jurka.J. (1993a) Discovering simple DNA sequences by the algorithmic significance method. *Comput. Applic. Biosci.*, **9**, 407–411.

Milosavljević.A. and Jurka.J. (1993b) Discovery by minimal length encoding: a case study in molecular evolution. *Machine Learn.*, **12**, 69–87.

Ollivier.E., Delorme.M.-O. and Hénaut.A. (1995) DosDNA occurs along yeast chromosomes, regardless of functional significance of the sequence. *C.R. Acad. Sci.*, **318**, 599–608.

Table II. Number of compressed windows in the four yeast chromosomes

Chromosome	No. of windows	Motifs of length 1			Motifs of length 2			Motifs of length 3		
		Comp.	% comp	% total	Comp.	% comp	% total	Comp.	% comp	% total
2	1614	107	76.4	6.29	11	7.8	0.68	22	15.6	1.36
3	630	42	67.7	6.66	11	16.6	1.74	9	14.5	1.42
8	1124	76	69.7	6.76	16	14.6	1.42	17	15.6	1.51
11	1332	84	70.0	6.30	14	11.6	1.05	22	18.3	1.65

RissanenJ. (1995) Minimum description length principle. In Kotz.S. and Johnson.N.L. (eds). *Encyclopedia of Statistical Sciences*, *5*. Wiley, New York, pp. 523–527.

Rivals.É. (1996) Algorithmes de compression et applications à l'analyse de séquences génétiques. PhD Thesis, LIFL, Université des Sciences et Techniques de Lille, France.

Rivals.É, Delgrange.O., Delahaye.J.-P. and Dauchet.M. (1995) A first step towards chromosome analysis by compression algorithms. In Bourbakis.N.G. (ed), *First International IEEE Symposium on Intelligence in Neural and Biological Systems*. IEEE Computer Society Press, pp. 233–239.

Wells.R.D. and Sinden.R.R. (1993) Genome rearrangement and stability, chapter Defined ordered sequence DNA, DNA structure, and DNA-directed mutation. In Davies.K.E. and Warren.S.T. (eds). *Genome Analysis*, *7*. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, NY.

Ziv.J. and Lempel.A. (1978) Compression of individual sequence via variable length coding. *IEEE Trans. Inf. Theory*, **24**, 530–536.

Received on March 27, 1996; revised on July 29, 1996; accepted on August 22, 1996

Index