THÈSE DE DOCTORAT
en Informatique

Comparaison de structures secondaires d'ARN.

Julien Allali

Numéro officiel
Université de Marne-la-Vallée

Thèse de doctorat

Spécialité : Informatique

présentée par

Julien Allali

pour l'obtention du titre de

Docteur en Informatique

sur le sujet

Comparaison de structures secondaires d'ARN.

directeurs de thèse

Maxime Crochemore

Marie-France Sagot

soutenue le 23 Décembre 2004 devant le jury composé de :

Alain Denise (Rapporteur)

Christine Gaspin (Rapporteur)

Robert Giegerich (Rapporteur)

Dominique Perrin (Examinateur)

Claude Thermes (Examinateur)

à Claire et Louis

Remerciements

Un grand merci à Marie-France Sagot d'avoir accepté il y a trois ans de diriger ma thèse. Merci pour son accueil toujours chaleureux, ses nombreux conseils et son aide. Merci pour avoir établi une relation d'égal à égal entre nous me permettant un plein épanouissement en tant que chercheur. Enfin, merci pour son amitié.

Merci à Maxime Crochemore d'avoir accepté de diriger ma thèse. Merci de m'avoir donné goût à l'algorithmique. Merci pour sa gentillesse et ses nombreux conseils.

Merci à Christine Gaspin d'avoir accepté de rédiger un rapport sur ma thèse. Merci à elle ainsi qu'aux membres du *BIA* pour leur accueil et pour l'interêt qu'ils ont manifesté envers mes travaux à chacune de mes visites.

Merci à Robert Giegerich d'avoir rapporté cette thèse et testé avec succès mon logiciel *migal*. Merci pour ses nombreux conseils.

Merci à Alain Denise pour avoir accepté de rapporter ma thèse. Merci aussi pour son accueil chaleureux au séminaire du LRI.

Merci à Dominique Perrin d'avoir accepté d'être examinateur dans mon jury de thèse. Merci pour sa gentillesse tout au long de ces années à l'Université de Marne la Vallée.

Merci à Claude Thermes d'avoir accepté d'être examinateur dans mon jury de thèse. Merci à lui et Yves d'Aubenton-Carafa pour l'interêt qu'ils ont manifesté envers mes travaux lors de mes visites au Centre de Génétique Moléculaire de Gif-Sur-Yvette. Leur aide m'a été précieuse dans ma compréhension des ARN. J'espère continuer notre collaboration en leur fournissant bientôt un outil de qualité.

Merci à Jean Berstel pour ses conseils ainsi que pour la confiance placée en moi et en mes capacités d'enseignant.

Merci à Mathieu Raffinot qui m'a orienté vers la recherche et l'algorithmique pour la génomique. Il n'a jamais été avare de conseils et il sait toujours réapparaitre au bon moment.

Merci à tous les membres du laboratoire de l'Institut Gaspard Monge. Ils ont toujours été gentils et amicaux, je suis convaincu que l'ambiance qu'ils créent tous les jours au laboratoire m'a été d'une grande aide durant ces trois années. Merci à Remi, Julien et Christophe pour les discussions passionnantes aux pauses coca. Merci à Line pour sa bonne humeur permanente et sa dévotion pour les cadeaux de naissance.

Merci aux membres de l'équipe Baobab ainsi qu'à Marina, Raquel et Sébastien pour leur accueil toujours chaleureux à Lyon.

Merci aux imageux : Vence, BuD et Pascal, leur présence est toujours un plaisir : "N'hésitez pas à prendre le RER".

Merci à mes compagnons de thèse : Chloé, Grom, Bud, Pascal, Pirro, Pierre et Benoît ; j'espère que nos chemins continueront à se croiser pour longtemps.

Merci à Skaya pour son amitié et ses nombreux coups de pouce techniques : il reste une bible pour beaucoup d'entre nous. Merci à lui et Bud pour leurs nombreux conseils dans les Warrens ou le système :)

Un grand merci à Laurent Marsan : tu m'as aidé à de nombreuses reprises tant au niveau professionnel que personnel. Méfie-toi quand même du 9-0, un jour il arrivera !

Merci à Remi pour son soutien et ses conseils durant ces 3 années, ainsi que pour son amitié.

Merci à Christophe pour son dynamisme, son franc caractère et son amitié.

Merci aux béta-testeurs de ma librairie sur l'arbre des k-facteurs : Alexandra Carvalho (Lisbonne), Patricia Thebault (Toulouse) et Pierre Peterlongo (Paris).

Merci à tous les développeurs de logiciels libres, outils qui m'ont été indispensables tout au long de ma thèse.

Merci à ma compagne Claire qui m'a soutenu tout au long de cette épreuve : ton soutien constant et ta confiance en moi ont été indispensables à la réalisation de ce travail. Merci à mon fils Louis, qui m'a apporté tant de joie durant ce dernier mois de thèse.

Enfin, merci à toute ma famille, en particulier mes parents et frères pour leur soutien ainsi que la fierté qu'ils ont maintes fois manifestée à mon égard.

Résumé

L'ARN, acide ribonucléique, est un des composants fondamentaux de la cellule. Les ARNs sont constitués d'une séquence orientée de nucléotides notés A,C,G et U. Une telle séquence se replie dans l'espace en formant des liaisons entre les nucléotides deux à deux. La fonction des ARNs au sein de la cellule est liée à la conformation spatiale que ces ARNs adoptent. Ainsi, il est essentiel de pouvoir comparer deux ARNs du point de vue de leur conformation afin de déterminer, par exemple, s'ils ont la même fonction.

On distingue trois niveaux dans la structure d'un ARN. La structure primaire correspond à la séquence de nucléotides, la structure secondaire est constituée de la liste de certaines liaisons formées entre les nucléotides (celles correspondant aux hélices), tandis que la structure tertiaire consiste en la description exacte de la forme tridimensionnelle de la molécule (les coordonnées de chaque nucléotide).

Bien que la structure tertiaire soit celle qui décrit le mieux la forme spatiale d'un ARN, il est admis que deux ARNs ayant une fonction moléculaire similaire ont des structure secondaires proches. On peut distinguer divers éléments de structure secondaire tels que les hélices, les boucles multiples, les boucles terminales, les boucles internes et les renflements.

Essentiellement trois formalismes ont été proposés à ce jour afin de modéliser la structure secondaire d'un ARN. Le premier, celui des séquences annotées par des arcs, permet de représenter la séquence de l'ARN, les arcs correspondant alors aux liaisons entre les nucléotides (les lettres de la séquence) qui s'apparient. Les 2-intervalles, qui sont une généralisation des séquences annotées, sont formés par deux intervalles disjoints. La structure secondaire peut alors être vue comme une famille de 2-intervalles. Enfin, une troisième représentation, celle des arbres enracinés ordonnés, offre de nombreuses possibilités afin de représenter la structure secondaire d'un ARN, du niveau nucléotidique au niveau du réseau des boucles multiples.

L'un des inconvénients de toutes ces approches réside dans le fait qu'elles modélisent la structure secondaire d'un ARN selon un point de vue particulier (nucléotides, hélices etc.). Nous proposons dans ce travail une nouvelle modélisation, appelée RNA-MiGaL, constituée de quatre arbres liés entre eux, représentant la structure à différents niveaux de précision. Ainsi, le plus haut niveau correspond au réseau de boucles multiples considéré comme étant le squelette de la molécule. Le dernier niveau détaille quant à lui les nucléotides de la séquence.

Pour comparer de telles structures, nous utilisons la notion de distance d'édition entre deux arbres. Cependant, au vu de certaines limitations de celle-ci pour comparer des arbres représentant la structure secondaire à un haut niveau d'abstraction, nous avons introduit une nouvelle distance d'édition qui prend en compte deux nouvelles opérations d'édition : la fusion de noeuds et la fusion d'arcs.

À l'aide de cette nouvelle distance, nous proposons un algorithme permettant de comparer deux RNA-MiGaLs. Celui-ci est implémenté au sein d'un outil permettant la comparaison de deux structures secondaires d'ARN.

Abstract

RNAs are one of the fundamental elements of a cell. Generally, RNAs are defined as oriented sequences of nucleotides (denoted by A,C,G and U). Inside a cell, RNAs do not have a linear shape but fold in space. The molecular function of an RNA strongly depends on this tri-dimensional folding. Hence, the comparison of the tri-dimensional structure of two RNAs is essential to determine whether the RNAs share the same function.

The structure of an RNA is generally divided into three parts. The first is the primary structure which corresponds to the sequence of nucleotides. The secondary structure is composed of the list of links between nucleotides that represent helices. Finally, the tertiary structure corresponds to the exact tri-dimensional folding of the RNA.

Although the tertiary structure is the most accurate definition of the spatial structure adopted by an RNA, it is well-known that two RNAs sharing the same function will also have closely related secondary structures. A few other structural elements can be distinguished in an RNA secondary structure. These are the helices, multiloops, hairpin loops, internal loops and bulges.

Up until now, essentially three data structures have been proposed to represent an RNA secondary structure : arc-annotated sequences, 2-intervals and rooted oriented trees. Arc-annotated sequences are sequences with arcs between nucleotides of the sequence that form a pair in the structure. 2-intervals generalise arc-annotated sequences and correspond to two disjoint subsets. An RNA secondary structure is then defined as a family of 2-intervals. Finally, rooted ordered trees can represent an RNA secondary structure at various levels, from the nucleotides up to the network of multiloops.

One of the drawbacks of all these approaches is that they model the secondary structure of an RNA from a specific point of view (nucleotides, helices etc.). We decided to introduce a new model called RNA-MiGaL, made

of four trees related among them. Each of these trees represents the structure of an RNA at a particular level of detail : the upper level models the network of multiloops that is considered as the skeleton of the secondary structure, while the lower level represents nucleotides.

We use the tree edit distance to compare two RNA-MiGaLs. However, due to some limitations of the classical edit distance to compare trees representing RNA secondary structures, we introduced two new edit operations named "node fusion" and "edge fusion", thus providing a new edit distance.

Using this distance, we developed an algorithm to compare two RNA-MiGaLs. The algorithm has been implemented in a package which allows RNA secondary structures to be compared in various ways.

Table des matières

Introduction **7**

1 Contexte Biologique **13**
1.1 Les ARN dans la cellule 13
1.2 Quelques exemples d'ARN 14
1.3 Repliement et structure des ARN 16
1.4 Comparaison des ARN 19
1.4.1 Repliement d'ARN. 20
1.4.2 Inférence de modèles. 21
1.4.3 Classification 21
1.4.4 Phylogénie. 21

2 État de l'art sur la modélisation d'ARN et les algorithmes de comparaison d'ARN **23**
2.1 Séquence annotée par des arcs 24
2.1.1 Introduction et notations 24
2.1.2 Problèmes et Algorithmes 27
2.1.3 2-intervalles : une généralisation des séquences annotées 49
2.1.4 Conclusion 52
2.2 Arbres enracinés et ordonnés 52
2.2.1 Plusieurs arbres possibles 52
2.2.2 Comparaisons d'arbres enracinés et ordonnés 57
2.2.3 Conclusion 74

3 Modélisation et algorithmes pour la comparaison des ARN **77**
3.1 Deux nouvelles opérations d'édition 78
3.1.1 L'édition d'arbres pour les ARN : résultats pratiques . 78
3.1.2 Deux nouvelles opérations d'édition 88

3.1.3 Algorithme 95
3.1.4 Résultats pratiques 99
3.2 MiGaL : "Multiple Graph Layer" 103
3.2.1 Description du modèle 104
3.2.2 RNA-MiGaL 104
3.2.3 Algorithme d'édition à travers MiGaL 107
3.2.4 Comparaison de RNA-MiGaL 111
3.2.5 Résultats pratiques 118

Conclusion **131**

A L'arbre des k-facteurs **135**
A.1 The at most k-deep factor tree 137
A.1.1 Introduction 137
A.1.2 Suffix trees 138
A.1.3 Ukkonen's algorithm 140
A.1.4 Factor trees 146
A.1.5 Coding and experiments 151
A.1.6 Conclusion 164

Bibliographie **165**

Table des figures

1.1 L'ARN 14
1.2 La traduction de l'ARN 16
1.3 Structure d'une hélice 17
1.4 Hélices d'un ARNt 17
1.5 Éléments de structure secondaire 18
1.6 Pseudo-nœud 19
1.7 Arbre phylogénique 22

2.1 Séquences annotées par des arcs 25
2.2 Types de séquences annotées 26
2.3 Séquences annotées et structures secondaires 26
2.4 Séquence annotée d'un ARNt 27
2.5 Problème LAPCS 29
2.6 Réduction de clique à *LAPCS(croisés,croisés)* 32
2.7 Complexités de LAPCS 34
2.8 Exemple de APS 35
2.9 Complexités de APS 36
2.10 Edition de séquence 38
2.11 Algorithme de calcul de la distance d'édition 41
2.12 Alignement de deux séquences 42
2.13 Alignement de séquences annotées 43
2.14 Opérations d'éditions sur les séquences annotées 44
2.15 Complexité de *EDIT(croisés, sans arc)* 47
2.16 Relations entre 2-intervalles 50
2.17 2-intervalles et structures secondaires 50
2.18 Famille de 2-intervalles 50
2.19 Accessibilité des bases d'une structure secondaire 53
2.20 Arbre d'une structure secondaire 54

2.21 Différents codage d'une structure secondaire par les arbres . . 56
2.22 Équivalence des séquences annotées et des arbres 57
2.23 Opération de substitution 59
2.24 Opérations de délétion et insertion 60
2.25 Distance d'édition entre deux arbres 63
2.26 Racines gauches d'un arbre 65
2.27 Ordre de calcul des distances entre sous-arbres 65
2.28 Algorithme du calcul de la distance d'édition entre arbres . . . 67
2.29 Décompositions en sous-arbres lors de l'édition 69
2.30 Association contrainte entre deux arbres 71
2.31 Association contrainte entre deux arbres 72
2.32 Alignement de deux arbres 73
2.33 Représentation parenthésée d'un ARN 75

3.1 Deux ARN de type RNAse P 79
3.2 Codage de deux ARN par des arbres 80
3.3 Deux ARN de type RNAse P 81
3.4 Codage de deux ARN par des arbres 82
3.5 Résultat de l'édition de deux arbres 83
3.6 Résultat de l'édition de deux arbres 83
3.7 Résultat de la comparaison de deux ARN 84
3.8 Résultat de la comparaison de deux ARN 85
3.9 Résultat de l'édition sur deux ARN 87
3.10 Fusion de nœuds, fusion d'arcs 89
3.11 Fusion de nœuds et délétion 93
3.12 Fusion de nœuds suivie d'une fusion d'arcs 94
3.13 Algorithme d'édition avec les opérations de fusion 96
3.14 Résultat de l'édition avec fusion sur deux arbres 100
3.15 Résultat de l'édition avec fusion sur deux ARN 101
3.16 Résultat de l'édition avec fusion sur deux ARN 102
3.17 Résultat de l'édition avec fusion sur deux arbres 102
3.18 MiGaL 105
3.19 Ordre et parenté à travers RNA-MiGaL 107
3.20 RNA-MiGaL 108
3.21 Comparaison de deux MiGaLs 110
3.22 Erreur lors de l'édition séparée 115
3.23 Ancres pour l'édition séparée 116
3.24 Fonctions de coûts 120

3.25 Codage alternatif pour le niveau 3 de RNA-MiGaL 124
3.26 Deux introns de Groupe I 125
3.27 Comparaison des RNA-MiGaLs au niveau 0 126
3.28 Comparaison des RNA-MiGaLs au niveau 1 127
3.29 Comparaison des RNA-MiGaLs au niveau 2 128
3.30 Comparaison des RNA-MiGaLs au niveau 3 129

A.1 Suffix trie, implicit suffix tree and suffix tree 139
A.2 Naive algorithm building suffix tree 140
A.3 Fast string insertion 143
A.4 Ukkonen algorithm 144
A.5 k-factor tree 147
A.6 k-factor tree construction algorithm 150
A.7 A suffix tree exemple 152
A.8 Coding the suffix tree 155
A.9 Insertion in a list of occurrences 158
A.10 Practical results of the factor tree 162

Introduction

L'informatique génomique désigne l'ensemble des procédés informatiques permettant le traitement de données biologiques. Cette discipline de l'informatique revêt un caractère original de par les relations qui se sont créées entre biologistes et informaticiens au cours des 30 dernières années.

Les données collectées par les biologistes sont de plus en plus nombreuses et leur traitement ne peut plus être effectué manuellement de manière systématique. L'ordinateur est devenu un outil indispensable pour la recherche d'informations dans ces données, ou encore pour diriger des expériences pratiques souvent longues et coûteuses.

Pour l'informatique, la génétique est un nouveau domaine riche en problèmes. Au début de la bioinformatique, la génétique a apporté un ensemble de questions relatives à l'ADN dont le séquençage commençait à se systématiser. Ces problèmes sont très divers, de la recherche d'une séquence particulière, par exemple un gène dans un génome, à la recherche d'un motif commun à plusieurs séquences. L'algorithmique du texte, domaine déjà établi de l'informatique, a trouvé là un nouveau champ d'applications apportant ses spécificités : la taille des données à traiter, les types de séquences cherchées, l'alphabet des séquences à considérer.

Pour chaque problème biologique, le travail commence par une analyse de ce problème par l'informaticien et le biologiste afin d'en construire une instance formelle qui doit être à la fois générale mais aussi contenir les contraintes pratiques liées à ce problème.

Imaginons qu'un biologiste recherche une séquence particulière dans un génome, celle-ci pouvant être présente de façon inexacte. Pour l'informaticien, il va falloir généraliser ce problème à la recherche d'un motif quelconque dans un texte avec éventuellement des erreurs. Il faut en outre caractériser ces erreurs ainsi que la nature du motif et du texte. L'informaticien propose alors une instance formelle du problème, qui peut être la recherche d'un motif court de taille fixe k dans un texte de taille n sur un alphabet à 4 lettres avec au plus e erreurs correspondant à des mutations (changement d'une lettre dans le motif). Si ce formalisme correspond bien au problème biologique initial, alors l'informaticien peut commencer à chercher une solution performante à ce problème. Dans cette recherche, il est important de tenir compte des caractéristiques du problème (court motif de taille fixe, alphabet de faible taille) pour en tirer profit.

Une fois une solution trouvée, celle-ci doit être testée par le biologiste afin de vérifier que l'algorithme résout bien la question initiale. Si ce n'est pas le cas, alors il faut revoir le formalisme. Il se peut, par exemple, que le nombre d'erreurs dépende de la taille du motif, ou bien ces erreurs peuvent aussi être des délétions ou insertions.

On voit ainsi que la bioinformatique demande une réelle collaboration entre biologistes et informaticiens tant pour la compréhension du problème biologique et de son formalisme que pour la validation de celui-ci sur des cas concrets.

C'est ainsi qu'a débuté cette thèse, par des discussions et des rencontres avec des biologistes pour analyser le problème de la comparaison de structures secondaires des ARN. Dans un premier temps, nous avons eu l'idée d'un nouveau type de modélisation des structures secondaires des ARN en utilisant différents niveaux de précision dans la représentation de cette structure. Cette idée fut soumise à des experts en ARN qui ont validé cette approche vis-à-vis de la nature des structures de ces ARN pour plusieurs raisons : d'une part, il est bien connu qu'au sein d'une même famille (ARN ayant une même fonction), la "forme" des ARN est conservée tandis que leur séquence peut varier de manière importante d'un ARN à l'autre. D'autre part, le réseau de boucles multiples (niveau le plus abstrait de notre modèle) constitue le squelette de la structure secondaire et est très souvent caractéristique des ARN ayant une même fonction.

L'étape suivante fut la recherche d'un algorithme permettant la comparaison de ce nouveau type d'objets. Cette recherche nous a amené à analyser en profondeur les algorithmes existants pour effectuer la comparaison d'arbres. En outre, l'algorithme d'édition pour la comparaison d'arbres représentant les ARN à un haut niveau d'abstraction a montré certaines limitations. Pour pallier celle-ci, nous avons défini une nouvelle distance pour comparer ces arbres, composants de notre modèle général, puis nous avons établi un algorithme pour la comparaison de deux ARN représentés par notre modèle.

La dernière étape de validation est encore en cours, mais les premiers résultats sont déjà prometteurs. Nous espérons ainsi reprendre la collaboration initiale avec les biologistes pour finir de valider notre modèle en pratique, l'améliorer et le compléter.

Le thèse est divisée en 3 parties, portant respectivement sur le contexte biologique de notre travail, l'état de l'art sur les modèles et algorithmes de comparaison des structures secondaires d'ARN et, enfin, la présentation de nos résultats dans ce domaine.

Dans la première partie, nous présentons le contexte biologique de nos travaux. En effet, pour répondre au mieux au problème de la comparaison d'ARN, il est nécessaire de bien comprendre ce que sont les ARN, leur structure et leur rôle au sein de la cellule. Après avoir décrit la nature moléculaire des ARN, nous expliquerons comment ils se replient et adoptent une conformation spatiale particulière fortement liée à l'activité de l'ARN. Nous finissons par la présentation de quatre problèmes sur les ARN nécessitant un algorithme efficace pour leur comparaison : le repliement de séquence, l'inférence de modèle, la classification automatique et la construction d'arbre phylogénique.

La deuxième partie présente l'état de l'art des modèles et algorithmes déjà proposés pour la représentation et la comparaison des structures secondaires d'ARN. Nous commencerons par les séquences annotées par des arcs, formalisme introduit par Evans [23] pour la modélisation des ARN consistant en des séquences auxquelles on ajoute des arcs reliant deux symboles de la séquence. Nous analyserons trois problèmes sur ces séquences :

– La recherche de la plus longue sous-séquence commune avec conservation des arcs.
– La recherche de motif dans une séquence annotée.
– L'édition de deux séquences annotées.

Nous finirons cette partie en présentant les 2-intervalles, introduits par Vialette [88], qui peuvent être vus comme une généralisation des séquences annotées.

En 1984, Zuker et Sankoff [100] ont été les premiers à introduire les arbres pour la représentation des structures secondaires d'ARN. Plus tard, Shapiro [78] présentera une autre façon de représenter ces structures par des arbres. Nous montrerons qu'en fait il existe de nombreuses façons de coder les structures secondaires par des arbres en fonction du type d'informations que l'on souhaite modéliser. Quel que soit le modèle choisi, l'arbre résultant est un arbre enraciné et ordonné, c'est-à-dire que l'ordre entre les fils d'un nœud compte. L'une des voies pour la comparaison de ces arbres est l'utilisation de la distance d'édition. L'édition d'arbre introduite en 1977 par Selkow [77] est une extension naturelle de la définition de l'édition entre séquences introduite en 1966 par Levenshtein [56]. Dans ce problème, on dispose d'un ensemble d'opérations de base appelées opérations d'édition pour transformer un arbre. À chacune de ces opérations est associé un coût. On définit alors la distance d'édition entre deux arbres comme le minimum des coûts des suites d'opérations transformant le premier arbre en le deuxième.

En 1989, Zhang et Shasha [98] fournissent un algorithme de type programmation dynamique plus simple et plus performant que celui de Tai. Nous analyserons en détail cet algorithme, une partie de nos travaux étant basée dessus. Nous terminerons cette partie en présentant la distance d'alignement d'arbres enracinés et ordonnés introduit par Jiang dans [49] qui, contrairement au cas des séquences, n'est pas équivalente à la distance d'édition.

La troisième partie expose les résultats de nos travaux de thèse. Dans un premier temps, nous faisons une analyse détaillée de certains résultats pratiques obtenus avec le calcul de distance d'édition entre deux arbres ordonnés. En outre, nous montrerons trois problèmes posés par l'utilisation de cette distance dans le contexte de la comparaison de structures secondaires d'ARN. Pour résoudre deux de ces trois problèmes, nous avons introduit de nouvelles opérations d'édition : la fusion de nœuds et la fusion d'arcs. Nous présentons ensuite un algorithme permettant le calcul de la distance d'édition munie de ces nouvelles opérations suivi des résultats obtenus en pratique. Dans un deuxième temps, nous définissons le type MiGaL, correspondant à un empilement de graphes reliés entre eux par des applications d'abstraction. L'application de MiGaL aux structures secondaires conduit au type RNA-MiGaL correspondant à 4 arbres enracinés, ordonnés représentant 4 vues de la structure secondaire à un niveau de détail différent. Ensuite, nous définissons un algorithme général pour la comparaison de telles structures ainsi qu'une déclinaison spécifique utilisant l'algorithme d'édition d'arbre précédemment défini. Nous concluons cette partie par quelques résultats pratiques obtenus avec notre nouveau modèle.

Enfin, en conclusion, nous aborderons les extensions possibles de nos travaux. Du point de vue biologique, nous tentons d'amener les premières idées pouvant permettre la généralisation de notre modèle aux structures tertiaires d'ARN. Cette démarche nous force à introduire des graphes avec cycles dans notre modèle multi-niveau et n'est pas sans conséquence sur les algorithmes permettant alors la comparaison de ces nouveaux modèles. Du point de vue algorithmique, nous essayerons d'examiner les différentes voies qui s'offrent à nous pour le passage au problème de la comparaison multiple de structures secondaires. L'intérêt principal est non plus de fournir une valeur d'appréciation de la similarité entre deux structures comme c'est le cas dans la comparaison deux à deux, mais plutôt d'être capable d'inférer un modèle commun à plusieurs ARN.

Chapitre 1

Contexte Biologique

Nous allons décrire le contexte biologique dans lequel se situent nos travaux. Dans un premier temps, nous détaillerons le rôle des ARN dans la cellule. Afin d'illustrer notre propos, nous donnerons quelques exemples d'ARN connus. Le repliement et la structure des ARN seront ensuite abordés. Pour finir, nous présenterons quelques problèmes posés par les biologistes faisant intervenir la comparaison de structures d'ARN.

1.1 Les ARN dans la cellule

L'ARN, acide ribonucléique, est un des composants fondamentaux de la cellule. Les ARN sont des molécules formées par des nucléotides. Ces **nucléotides** résultent de la combinaison d'une base hétérocyclique azotée, Adénine, Guanine, Uracile ou Cytosine, avec un ribose et un groupement acide phosphorique. Il est usuel de faire référence à ces nucléotides par les lettres A,C,G et U correspondantes aux différentes **bases**.

La figure 1.1 représente les 4 nucléotides ainsi qu'une chaîne d'ARN (formée par ces nucléotides). On peut voir qu'une des extrémités de la chaîne se termine par un groupement phosphate, c'est l'extrémité 5' ; l'autre extrémité se termine par un sucre, c'est l'extrémité 3'. Les ARN sont ainsi des molécules orientées : la séquence débute du coté 5' et se termine du coté 3'.

Les ARN sont le produit de la transcription de l'ADN. Cette transcription est effectuée par l'ARN polymérase (3 types chez les eucaryotes, un seul chez les procaryotes). Celle-ci se fixe à l'ADN sur une région qualifiée de promotrice. L'ARN polymérase parcourt alors le brin d'ADN dans le sens 3' vers 5'

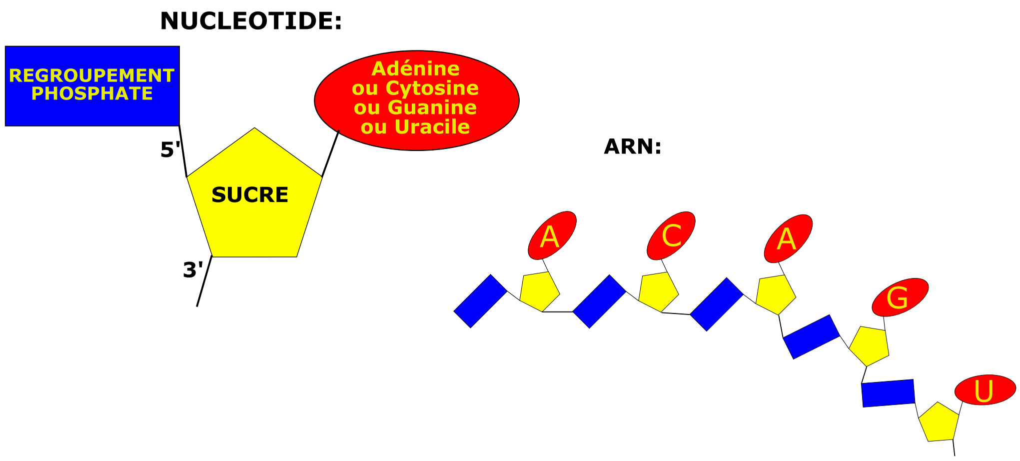


FIG. 1.1 – Sur la gauche, la représentation d'un nucléotide ; à droite celle d'un ARN.

et contruit une séquence complémentaire à l'ADN parcouru. Lorsque l'ARN polymérase rencontre un signal spécifique sur le brin d'ADN, la transcription de l'ADN est terminée et une séquence complémentaire de la région lue est ainsi produite. Chez les procaryotes, cette séquence est produite dans le cytoplasme et peut être directement utilisée pour fabriquer des protéines. Chez les eucaryotes, cette séquence est appelée le transcrit primaire ou ARN nucléaire hétérogène et va subir de profondes modifications à travers le processus de maturation. La maturation du transcrit peut consister en la suppression de certaines parties dites non codantes ou même en la partition du transcrit en plusieurs ARN.

L'ARN ainsi produit peut remplir de nombreux rôles au sein de la cellule en fonction du gène dont il est issu. Nous allons maintenant voir des exemples de divers types d'ARN.

1.2 Quelques exemples d'ARN

- Les ARN les plus connus sont les **ARN messagers**, notés **ARNm**. Ces ARN sont les porteurs de l'information génétique (ADN) codant pour une protéine. C'est à travers le procédé de traduction que les protéines sont synthétisées à partir d'un ARNm.

Jusqu'à très récemment, il a été supposé que les ARNm n'étaient pas structurés et n'avaient pas d'autre rôle que de véhiculer le code d'une protéine. Cependant il a été montré [81] [8] [94] que certains ARNm jouent un rôle important dans certains mécanismes de régulation et qu'ils seraient plus structurés qu'initialement supposé. Par exemple, les "riboswitches" [81][30] sont formés dans les ARNm et sont impliqués dans la régulation de gènes chez les bactéries.

– Les **ARN ribosomaux**, notés **ARNr**, sont les composants de base des ribosomes. Ces ribosomes sont formés de 2 ARNr (nommées respectivement grande sous unité et petite sous unité) ainsi que de protéines. Les ribosomes sont les acteurs principaux de la traduction d'un ARNm en protéine.
– Les **ARN de transfert**, ou **ARNt**, jouent également un rôle lors de la traduction, en partenariat avec les ribosomes. Les ARNt font le lien entre les nucléotides d'un ARNm et les acides aminés (composants de base des protéines). Ce sont eux qui apportent les acides aminés nécessaires à la synthèse des protéines et ainsi font le lien entre le monde des ARN et celui des protéines.

La figure 1.2 présente de manière très simplifiée le mécanisme de traduction et le rôle joué par les ARNm, ARNr et ARNt.

Comme nous venons de le voir, les ARNr, les ARNt ainsi que certains ARNm exercent une fontion moléculaire spécifique au sein de la cellule. Ces fonctions sont liées à des conformations spatiales bien précises. On dit que les ARN sont structurés : ils adoptent dans l'espace une conformation déterminée et spécifique à leur activité. Ainsi, des motifs de structures ont pu être établis pour certaines classes d'ARN [96][37][15]. Nous reviendrons sur ces notions plus tard.

Depuis les années 90, de nombreux autres types d'ARN ont été découverts. Par exemple, les snARN ("small nuclear RNA") sont des ARN de petite taille intervenant dans le processus de maturation du transcrit primaire, les snoARN ("small nucleolar RNA") jouent un rôle dans la modification de bases de plusieurs familles d'ARN (ARNr, snARN, ...), les microARN inhibent l'expression d'un gène. Pour l'ensemble de ces ARN, la conformation spatiale adoptée par l'ARN est également directement liée à son activité. Nous allons maintenant voir en quoi consiste cette conformation spatiale.

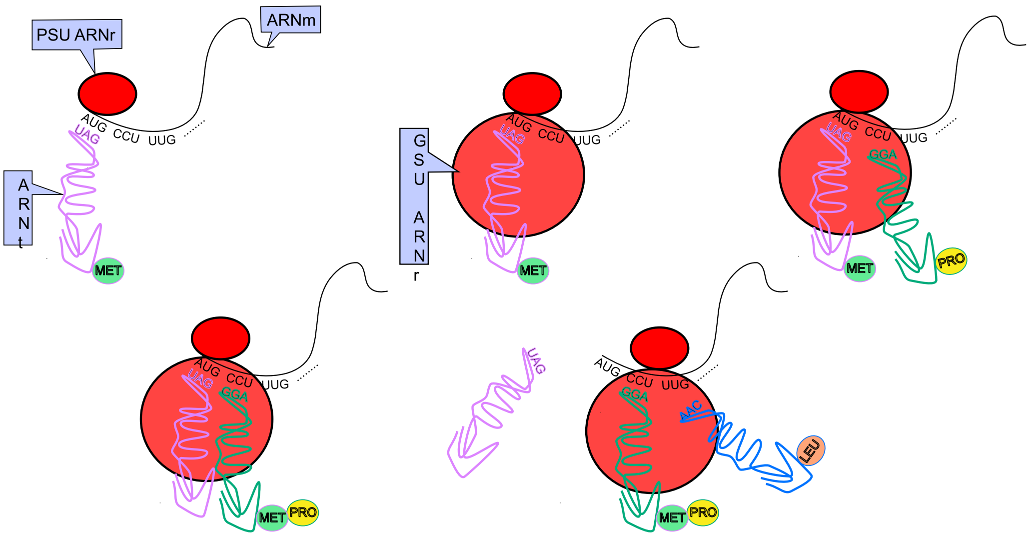


FIG. 1.2 – Les différentes étapes de la traduction.

1.3 Repliement et structure des ARN

Les ARN sont des séquences mono-brin de nucléotides (A, C, G et U). Dans la cellule, ils ne restent pas sous une forme linéaire. En effet, un ARN se replie sur lui-même et des liaisons fortes se créent entre certains de ses nucléotides. Plusieurs liaisons peuvent être rencontrées [73][6] :

- Les liaisons Watson-Crick ou liaisons canoniques sont les liaisons majoritaires. Une liaison de ce type se fait entre un A et un U, ou bien entre un C et un G.
- Les liaisons de type Wobble mises en évidence en 1966 par Crick sont les liaisons entre un G et un U. Ces liaisons sont les liaisons non-canoniques les plus courantes.
- D'autres liaisons plus rares telles que les liaisons G–A [25] et les liaisons C–U [70].

Ces liaisons se forment par blocs tout au long de l'ARN. Un bloc, ou suite de liaisons nucléotidiques, forme une hélice dans l'espace (en trois dimension). La figure 1.3 montre une hélice formée par une telle suite. La figure 1.4 montre les hélices qui composent un ARNt.

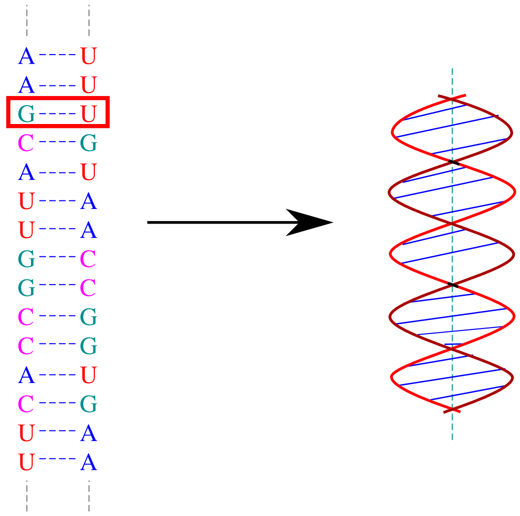


FIG. 1.3 – À gauche, on voit une suite de liaisons dont un wobble en rouge. À droite, l'hélice formée par cette suite de liaisons.

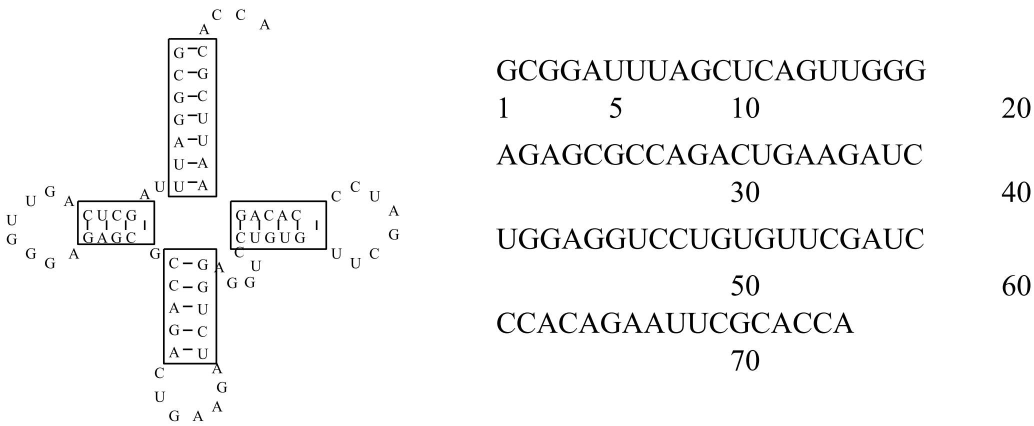


FIG. 1.4 – À gauche sont représentées les quatres hélices qui constituent la structure secondaire de l'ARNt dont la séquence est à droite.

L'ensemble des liaisons nucléotidiques, ou **appariements de bases**, qui composent les hélices d'un ARN constitue sa structure secondaire. Une fois les hélices déterminées, on peut distinguer les éléments de structure secondaire suivants (ceux-ci sont représentés sur la figure 1.5) :

- Une hélice est une suite continue de liaisons entre nucléotides.
- Une boucle terminale est la suite de bases non appariées formant une boucle à l'extrémité d'une hélice.
- Une boucle multiple désigne le point de rencontre d'au moins 3 hélices.
- Une boucle interne est formée par deux suites de bases reliant deux hélices. Une boucle interne peut être vue comme un cas particulier d'une boucle multiple.
- Un renflement est une boucle interne dont l'une des suites de bases est de longueur nulle.
- Une tige dénote une suite d'hélice(s)/boucle(s) interne(s)/renflement(s).

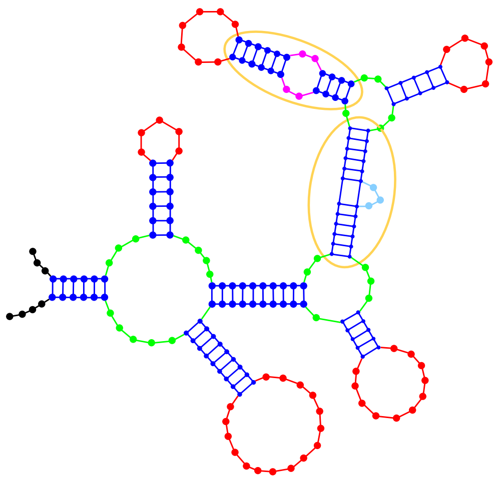


Fig. 1.5 – Les points représentent les bases de l'ARN. Les hélices sont indiquées en bleu, les boucles terminales en rouge, les boucles multiples en vert, une boucle interne en rose et un renflement en bleu ciel. Cet ARN possède 8 tiges dont 6 ne sont formées que d'une hélice, les deux autres étant entourées en jaune.

La structure primaire d'un ARN correspond à sa séquence de nucléotides.

On définit les pseudo-nœuds [91] comme étant une suite de liaisons nucléotidiques entre des nucléotides *libres* (non appariées) dans la structure secondaire (voir figure 1.6). Lorsque l'on prend en compte les pseudo-nœuds ainsi que d'autres intéractions formées au sein de l'ARN telles que par exemple les liaisons triples (une troisième base se lie avec une liaison canonique) et les liaisons hélice-hélice [14], on parle de structure tertiaire. À ce niveau, la structure spatiale de l'ARN est parfaitement définie.

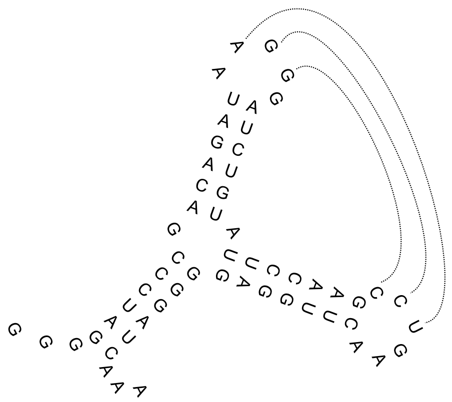


Fig. 1.6 – Exemple d'un pseudo-nœud : les liaisons qui forment le pseudo-nœud sont en pointillé.

Il est bien évident que la structure tertiaire est la structure la plus précise permettant d'étudier la fonction d'un ARN, celle-ci étant liée à la conformation spatiale adoptée. Cependant, il est admis que deux ARN ayant une structure secondaire proche auront une fonction moléculaire similaire. C'est pourquoi l'ensemble des approches visant à étudier la fonction des ARN réalisées à ce jour repose sur la structure secondaire avec éventuellement la prise en compte des pseudo-nœuds.

1.4 Comparaison des ARN

La comparaison de structures secondaires d'ARN trouve de nombreuses applications en biologie. En effet, il est couramment admis que si deux ARN ont une structure spatiale proche, alors ils ont la même fonction biologique.

Nous allons voir des exemples pratiques d'utilisation de la comparaison de la structure secondaire des ARN.

1.4.1 Repliement d'ARN.

Actuellement il existe deux façons de déterminer la structure secondaire d'un ARN.

La première est expérimentale à l'aide de méthodes de cristallographie par diffraction aux rayons X ou de résonance magnétique nucléaire [71]. Ces méthodes sont longues et coûteuses.

La deuxième se fait à l'aide d'algorithmes utilisant la structure primaire d'un ARN. Un premier type d'algorithme utilise une approche basée sur la thermodynamique [64][65][74]. On détermine la structure secondaire de l'ARN qui possède l'énergie libre minimum selon un modèle théorique de calcul d'énergie. C'est donc la structure théorique la plus stable. Bien que cette approche donne de bons résultats, surtout sur les petits ARN, il arrive souvent que la structure prédite ne soit pas la bonne [63]. En effet, un ARN n'adopte pas toujours la structure dont l'énergie totale est minimale mais plutôt une conformation ayant une énergie proche de la conformation la plus stable. C'est pourquoi les programmes basés sur cette approche proposent non pas de générer la meilleure structure secondaire mais plutôt un certain nombre de structures possibles dont l'énergie est proche de l'énergie minimum. Il faut alors choisir parmi ces structures celle qui est "correcte". Si l'on dispose de la structure secondaire d'un ARN dont la fonction est identique à l'ARN dont on cherche le repliement, on peut alors comparer cette structure à chacune des structures possibles afin d'en extraire la plus proche.

Une autre type d'algorithme repose uniquement sur une analyse comparative d'un ensemble de séquences d'ARN dont on fait l'hypothèse qu'ils ont la même fonction [40][33][16][26][39].

Bien que nombreuses, il est possible de déterminer un ensemble de structures secondaires possibles pour chacune de ces séquences. Une fois ces structures calculées, on peut comparer ensemble ces structures afin de trouver pour chaque séquence la structure présentant la meilleure similarité avec les structures des autres séquences d'ARN.

1.4.2 Inférence de modèles.

Une autre application de la comparaison de structures est la génération d'un modèle pour une famille d'ARN connus [20][95]. Par modèle, on entend une description la plus précise possible d'une structure secondaire à laquelle la majorité des ARN de cette famille se conforme.

Pour calculer ce modèle, on dispose d'un ensemble de structures secondaires d'ARN ayant une même fonction biologique. Il faut alors comparer ces structures entre elles et déterminer avec le plus de précision possible les parties communes.

De tels modèles existent déjà pour certaines familles d'ARN [96][15], cependant ceux-ci ont été construits le plus souvent *manuellement* et la génération automatique "fiable" de modèles reste un problème encore largement ouvert.

1.4.3 Classification

Le problème de la classification ("clustering") est de partitionner un ensemble de structures secondaires selon leur similarité. Pour cela, on dispose d'un ensemble de structures dont on ne connaît pas *a priori* la fonction. Dans un premier temps, on compare chacune de ces structures deux à deux. Puis, on constitue des groupes de telle façon à ce que toutes les structures secondaires au sein d'un même groupe soient "proches" [79].

L'une des difficultés est de définir la notion de "proche" et d'établir le nombre de groupes "idéal". En effet, une solution triviale à ce problème revient à créer autant de groupes qu'il y a de séquences. La solution opposée consiste à n'avoir qu'un seul groupe pour l'ensemble des séquences. Ainsi, on voit bien qu'il faut mettre en place des critères à la fois sur le nombre de groupes et la similarité des structures au sein de chaque groupe. L'outil utilisé pour la comparaison des structures tient une place importante dans cette démarche car il sert à établir la similarité entre les ARN.

1.4.4 Phylogénie.

La phylogénie consiste en l'étude de l'évolution de divers organismes afin de déterminer leurs liens de parenté (souvent représenté par un arbre)[38][62][46]. Dans la figure 1.7, on peut voir des arbres phylogéniques pour 5 espèces. La racine de ces arbres représente l'ancêtre hypothétique commun à ces 5 espèces. Un nœud interne représente l'ancêtre commun aux espèces descendantes de ce nœud. Dans notre exemple, l'arbre de gauche nous dit que les espèces C et D ont un ancêtre commun, lui même issu d'une espèce qui est aussi ancêtre de E. L'arbre de droite donne un autre schéma possible de l'évolution de ces espèces.

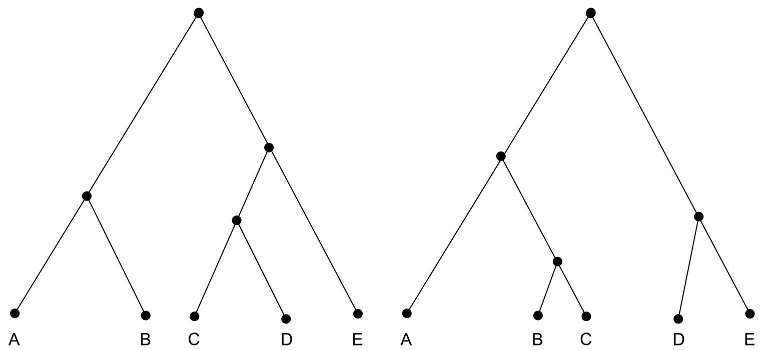


FIG. 1.7 – Exemple de deux arbres phylogéniques des 5 espèces : A,B,C,D et E. Chaque arbre présente une histoire différente de l'évolution.

Ainsi, l'une des problématiques de la phylogénie est d'établir l'arbre phylogénétique d'un ensemble d'espèces. Pour cela, on doit observer les similarités entre ces espèces afin de trouver une histoire évolutive possible reflétant ces similarités. Par exemple, on peut utiliser le gène codant pour une même protéine de plusieurs espèces. Une autre possibilité est d'utiliser des structures d'ARN pour étudier l'évolution d'une famille d'ARN [72][53]. Pour déterminer la proximité de ces structures, nous avons besoin d'algorithmes capables d'en effectuer la comparaison.

Comme nous venons de le voir, la comparaison de structures d'ARN présente de nombreuses applications et est un aspect important dans l'étude des ARN. Nous allons maintenant examiner les différentes possibilités existantes pour comparer deux ARN.

Chapitre 2

État de l'art sur la modélisation d'ARN et les algorithmes de comparaison d'ARN

Depuis une quinzaine d'années de nombreux modèles et formalismes ont été proposés pour représenter la structure des ARN dans le but de les comparer. Dans cette partie, nous allons exposer ces différents formalismes ainsi que les algorithmes qui leur sont associés.

Le premier modèle sera les séquences annotées par des arcs. Ce modèle est très proche de la structure secondaire car il consiste en une séquence augmentée par des arcs entre les symboles de celles-ci qui peuvent être vus comme les liaisons entre les nucléotides. Nous verrons trois problèmes formalisés à partir de ce modèle : la recherche de la plus longue sous séquence annotée commune à deux séquences annotées, la recherche d'un motif dans une séquence annotée par des arcs et l'édition de deux séquences annotées. Ces problèmes sont, dans le cas général, NP-Complets. Nous examinerons des restrictions compatibles avec la modélisation des ARN permettant d'obtenir des algorithmes polynomiaux. Nous terminerons sur les 2-intervalles. Ceux-ci peuvent être vus comme une généralisation des séquences annotées par des arcs. Un 2-intervalle est l'union disjointe de deux intervalles sur les entiers ou les réels. Les ARN sont alors modélisés par un ensemble de 2-intervalles (un 2-intervalle représentant une hélice). Le problème étudié dans ce cas sera celui de la recherche de motif dans cet ensemble. De même que pour les séquences annotées, ce problème a été prouvé comme étant NP-Complet dans le cas général. Cependant, des restrictions sur le motif cherché permettent d'obtenir des algorithmes efficaces.

Enfin, le dernier chapitre montre comment utiliser les arbres pour modéliser les ARN. Nous verrons qu'il existe de nombreux types d'arbres pour cette modélisation en fonction du type d'information que l'on souhaite représenter. Nous finirons sur le problème d'édition entre deux arbres ainsi que celui de leur alignement.

2.1 Séquence annotée par des arcs

Evans dans [23] et [24] introduit la structure de *séquence annotée par des arcs*. La fin de ce chapitre est consacrée aux 2-intervalles qui peuvent être vu comme une généralisation des séquences annotées.

Dans un premier temps, nous allons donner la définition d'une telle séquence ainsi que certains cas particuliers de séquences annotées. Puis nous aborderons trois problèmes algorithmiques : la recherche de la plus longue sous-séquence commune avec conservation des arcs, la recherche de motif dans une sous-séquence annotée et l'édition de deux séquences annotées.

2.1.1 Introduction et notations

Dans l'ensemble de ce chapitre nous adoptons les notations suivantes : on désigne par t une séquence de $|t|$ symboles pris sur un alphabet Σ. On notera $t[i]$ le $i^{\text{ème}}$ caractère de t. Ainsi $t = t[1]t[2]\ldots t[|t|]$. La cardinalité d'un ensemble E est notée $|E|$.

La définition générale d'une séquence annotée est la suivante :

Définition 1. *Séquence annotée par des arcs :*
Un séquence annotée par des arcs $S(t, A)$ est définie par une chaîne t sur un alphabet Σ ainsi qu'un ensemble A de couples de positions (i, j) sur t ($i, j \in [1; |t|]$) tels que $\forall (i, j) \in A,\ i < j$.

La figure 2.1 montre deux exemples de séquences annotées par des arcs.

On peut fixer un certain nombre de contraintes (limitations) sur une séquence annotée telles que :

1. Un symbole $c \in \Sigma$ ne peut être mis en relation via un arc qu'avec un autre symbole $c' \in \Sigma$.

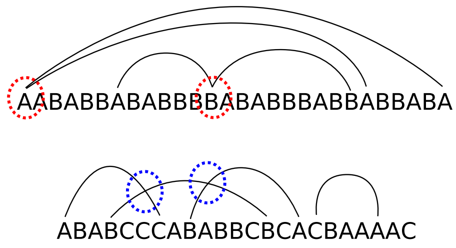


FIG. 2.1 – Deux exemples de séquences annotées par des arcs.

2. Les arcs de la séquence ne peuvent se croiser (cas représenté par des cercles bleus dans la figure 2.1) :
   $\forall (i, j)$ *et* $(k, l) \in A$ tels que $i < k$ alors $j \leq k$ ou $j \geq l$.
3. Chaque élément de t ne peut avoir au plus qu'un seul arc (cas représenté par des cercles rouges sur la figure 2.1) :
   $\forall i, l \in [1; |t|]$, il existe au plus un couple $(i, l) \in A$.
   Cela signifie que deux arcs ne peuvent partager une même extrémité.
4. Deux arcs ne peuvent être inclus l'un dans l'autre :
   $\forall (i, j)$ *et* $(k, l) \in A$ alors soit $i < j \leq k < l$, soit $k < l \leq i < j$.
5. A est vide.

Muni de ces contraintes, nous pouvons définir différents types de séquences annotées par des arcs (voir la figure 2.2) :

général : pas de limitation.
croisés : contrainte 3.
imbriqués : contraintes 2 et 3.
successifs : contraintes 2, 3 et 4
sans arc : contrainte 5.

On remarquera la relation d'inclusion existante entre ces groupes. En effet, les séquences de type *sans arc* définissent un sous-ensemble des séquences de type *successifs*... qui sont un sous-ensemble des séquences *général*. Voyons maintenant comment modéliser les ARN avec de telles séquences.

Comme l'a fait remarquer Lin dans [59], lorsqu'on représente la structure **secondaire** des ARN par des séquences annotées par des arcs, alors deux arcs ne partagent pas une même extrémité (restriction 3). La figure 2.3 montre

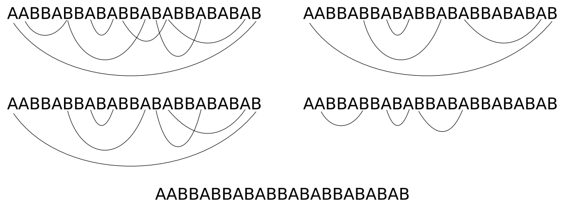


FIG. 2.2 – Exemple des différents types de séquences annotées (de gauche à droite et de haut en bas) : *général*, *imbriqués*, *croisés*, *successifs* et *sans arc.*

un ARN représenté par une séquence annotée par des arcs.

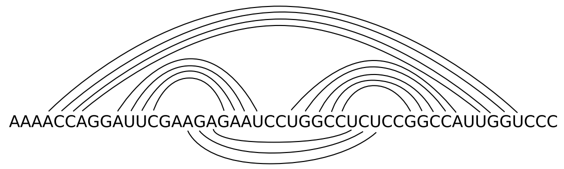


FIG. 2.3 – Une séquence annotée par des arcs représentant un ARN. Les arcs dessinés sous la séquence correspondent à un pseudo-nœud et croisent d'autres arcs.

Proposition 1. *Modélisation de la structure secondaire des ARN sans pseudo-nœuds :*
Toute structure secondaire d'ARN qui ne contient pas de pseudo-nœuds est modélisable par une séquence annotée par des arcs de type imbriqués.

Démonstration. Immédiat (voir figure 2.4) de par la définition d'une structure secondaire sans pseudo-nœuds. □

Proposition 2. *Modélisation de la structure secondaire des ARN avec pseudo-nœuds :*
Toute structure secondaire d'ARN prenant en compte les pseudo-nœuds est modélisable par une séquence annotée par des arcs de type croisés.

Démonstration. Immédiat (voir figure 2.4) de par la définition d'une structure secondaire avec pseudo-nœuds. □

Notons aussi que la structure primaire (séquence) des ARN est modélisable par le type *sans arc*. La structure tertiaire des ARN implique des liaisons telles que les pseudo-nœuds ou encore les liaisons triples (liaison d'une base avec un liaison canonique). Ainsi les structures tertiaires des ARN sont modélisables par les séquences annotées par des arcs de type *général*.

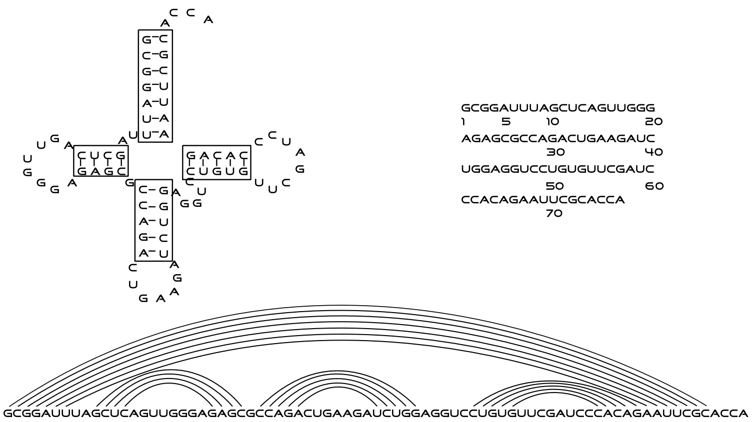


FIG. 2.4 – Séquence annotée représentant un ARN de transfert.

2.1.2 Problèmes et Algorithmes

Nous allons maintenant voir trois problèmes sur les séquences annotées par des arcs, à savoir la recherche de la plus longue sous séquence commune à deux séquences annotées, la recherche d'un motif dans une séquence annotée et l'édition de deux séquences annotées par des arcs.

LAPCS : recherche d'une sous-séquence commune à deux séquences.

Comme nous l'avons vu, un des enjeux de l'étude des ARN est l'inférence d'une sous-structure commune à deux (ou plusieurs) ARN. Dans le cadre d'une modélisation par des séquences annotées par des arcs, ceci se traduit par le problème de la recherche de la plus longue sous-structure commune à deux séquences annotées, aussi appelé LAPCS (pour "*longest arc preserving common subsequence*"). Ce problème repose sur la notion d'*association* entre deux séquences annotées avec préservation des arcs.

Définition 2 (Association). *Soient deux séquences annotées par des arcs* $S_1(t_1, A_1)$ *et* $S_2(t_2, A_2)$*. On définit une association* $M(S_1, S_2)$ *comme étant un ensemble de couples* (i, j) *avec* $i \in [1; |t_1|]$ *et* $j \in [1; |t_2|]$ *tels que :*
$\forall(i_1, j_1) \in M$ *et* $\forall(i_2, j_2) \in M$ *:*
- *Si* $i_1 = i_2$ *alors* $j_1 = j_2$ *(association un-un).*
- *Si* $i_1 < i_2$ *alors* $j_1 < j_2$ *(préservation de l'ordre).*
- $(i_1, i_2) \in A_1$ *si et seulement si* $(j_1, j_2) \in A_2$ *(préservation des arcs).*
- $t_1[i_1] = t_2[j_1]$ *(préservation des symboles).*

De là, nous pouvons définir une sous séquence commune à deux séquences annotées avec préservation des arcs :

Définition 3 (Sous séquence commune). *Soient deux séquences annotées par des arcs* $S_1(t_1, A_1)$ *et* $S_2(t_2, A_2)$*. On dira que* $L(t, A)$*, séquence annotée par des arcs, est une sous séquence commune à* S_1 *et* S_2 *avec préservation des arcs s'il existe une association (suite de couples)* $M = \{m_1, \ldots, m_{|t|}\}$ *telle que :*
- $\forall m_v = (i, j) \in AS,\ t[v] = t_1[i] = t_2[j]$.
- $\forall m_u = (i_1, j_1), m_v = (i_2, j_2) \in M,\ (u, v) \in A$ *si et seulement si* $(i_1, i_2) \in A_1$ *(et donc* $(j_1, j_2) \in A_2$*).*

Il ne reste plus qu'à définir la plus longue sous séquence commune à deux séquences avec préservation des arcs :

Définition 4 (Problème LAPCS(longest arc-preserving common subsequence)). *Soient deux séquences annotées par des arcs* $S_1(t_1, P_1)$ *et* $S_2(t_2, P_2)$*. On appelle LAPCS, la sous séquence commune à* S_1 *et* S_2 *de longueur maximale.*

On notera *LAPCS(Type1, Type2)* le problème de la recherche de la plus longue sous séquence commune à deux séquences annotées, la première étant de type *Type1* et la deuxième de type *Type2*, parmi les types *général, croisés, imbriqués, successifs, sans arc* définis précédemment.

Comme le montre la table 2.5, les différents problèmes *LAPCS(Type1, Type2)* sont inclus les uns dans les autres du fait de la relation d'inclusion qui existe entre les différents types de séquences annotées.

Le problème *LAPCS(sans arcs, sans arcs)* correspond à la recherche de la plus longue sous séquence commune à deux séquences, problème bien connu en algorithmique du texte.

LAPCS	général	croisés	imbriqués	successifs	sans arcs
général	$\bullet$				
	$\cup$				
croisés	$\bullet$ $\supset$	$\bullet$			
	$\cup$	$\cup$			
imbriqués	$\bullet$ $\supset$	$\bullet$ $\supset$	$\bullet$		
	$\cup$	$\cup$	$\cup$		
successifs	$\bullet$ $\supset$	$\bullet$ $\supset$	$\bullet$ $\supset$	$\bullet$	
	$\cup$	$\cup$	$\cup$	$\cup$	
sans arcs	$\bullet$ $\supset$	$\bullet$ $\supset$	$\bullet$ $\supset$	$\bullet$ $\supset$	$\bullet$

FIG. 2.5 – Relation entre les différentes instances du problème *LAPCS*.

Proposition 3 (*LAPCS(sans arcs, sans arcs)* [41]). *La recherche de la plus longue sous séquence commune à deux séquences t_1 et t_2 sur un alphabet fixe se fait en $O(|t_1||t_2|)$.*

Démonstration. L'algorithme résolvant ce problème a été proposé en 1975 par Hirschberg et repose sur une technique de programmation dynamique. Pour plus de détails on se référera à [41]. □

Dans [23], Evans montre que *LAPCS(général, sans arcs)* est NP-Complet. Nous pouvons tout d'abord faire la proposition suivante :

Proposition 4 (*LAPCS(Type1, Type2)*). *LAPCS(Type1, Type2) est dans NP.*

Puis Evans montre que *LAPCS(général, sans arcs)* est NP-dur par réduction à partir du problème des *ensembles indépendants* :

Définition 5 (Problème des ensembles indépendants (*Independent Set*)). *Soit un graphe $G(V, E)$, défini par un ensemble de sommets V et d'arcs E. On dit qu'un sous ensemble V' de sommets de G est indépendant, si pour chaque couple de sommets $(u, v) \in V'$, il n'existe pas d'arc entre u et v dans E. Le problème des* ensembles indépendants *est le suivant : Soit un graphe $G(V, E)$ et un entier k, existe-t-il un sous ensemble indépendant de G de taille k ?*

Proposition 5 (Réduction). *Le problème des ensembles indépendants peut être réduit en temps polynomial au problème LAPCS(général, sans arcs).*

Démonstration. Nous ne donnons pas la réduction complète, mais seulement le codage utilisé, pour la preuve complète voir [23].

Pour effectuer la réduction, on part d'un graphe $G(V, E)$ et on construit deux séquences annotées $S_1(t_1, P_1)$ et $S_2(t_2, P_2)$. Pour S_1, on a $t_1 = a^{|V|}$ et $P_1 = E$ et pour S_2, $t_2 = a^k$ et $P_2 = \emptyset$. On montre ainsi que G possède un ensemble indépendant de taille k si et seulement si il existe une association de taille k entre S_1 et S_2. □

On conclut par le théorème suivant :

Théorème 1 (Complexité de LAPCS(général, sans arcs)). *LAPCS(général, sans arcs) est NP-Complet.*

Démonstration. D'après les propositions 4 et 5, le problème est dans NP et est NP-dur. □

Corollaire 1 (Complexité de LAPCS(général, Type2)). *LAPCS(général, Type2) est NP-Complet quel que soit le type de* Type2.

Démonstration. Quelque soit le type de Type2, LAPCS(général, Type2) a pour sous problème LAPCS(général, sans arcs) (voir la table d'inclusion 2.5) et est dans NP (proposition 4). □

Puis Evans montre que le problème de recherche de clique dans un graphe peut être réduit en temps polynomial à *LAPCS(croisés, croisés)*. Voici la définition du problème de recherche de clique dans un graphe :

Définition 6 (Clique dans un graphe (décision)). *Soit $G(V, E)$ un graphe non orienté connexe et k un entier. Existe-t-il un sous graphe complet de G à k sommets ?*

Proposition 6 (Réduction). *Le problème de recherche de clique dans un graphe peut être réduit en temps polynomial au problème LAPCS(croisés, croisés).*

Démonstration. Voici le codage utilisé : On se donne un graphe $G(V, E)$ composé de n sommets et un entier k. On construit alors les séquences annotées $S_1(t_1, P_1)$ et $S_2(t_2, P_2)$ de la manière suivante :

- $t_1 = (ba^n b)^n$: t_1 est composé de n blocs de la forme $(ba^n b)$. Ainsi la longueur de t_1 est $n^2 + 2n$.

- $P_1 = \{((u-1)(n+2)+1, u(n+2))|u \in V\} \bigcup$
  $\{((u-1)(n+2)+v+1, (v-1)(n+2)+u+1|(u,v) \in E\}$
  La première partie définit un arc entre les deux "b" aux extrémités de chaque bloc (ba^nb) de t_1. La deuxième partie, pour chaque arc (u,v) de E, crée un arc entre le $v^{\text{ème}}$ "a" du bloc u et le $u^{\text{ème}}$ "a" du bloc v.
- $t_2 = (ba^kb)^k$: t_2 est composé de k blocs de la forme (ba^kb). Ainsi la longueur de t_2 est k^2+2k.
- $P_2 = \{((u-1)(k+2)+1, u(k+2))|u \in [1;k]\} \bigcup$
  $\{((u-1)(k+2)+v+1, (v-1)(k+2)+u+1)|u \in [1;k[, v \in]u;k]\}$
  La première partie de l'expression construit un arc entre les deux "b" aux extrémités de chaque bloc (ba^kb) de t_2. Dans la deuxième partie, pour chaque bloc i, on crée un arc pour tous les $j^{\text{ème}}$ "a", $j > i$, vers le $(k-j)^{\text{ème}}$ "a" du bloc j.

En fait, S_2 correspond à la séquence S_1 que l'on obtiendrait à partir d'un graphe complet de taille k.

Nous avons donné ici le codage utilisé pour la réduction, on se référera à [23] pour la preuve complète. □

La figure 2.6 montre l'exemple d'un graphe et des séquences annotées construites selon le codage décrit ci-dessus.

Il en suit le théorème suivant :

Théorème 2 (Complexité de *LAPCS(croisés, croisés)*). *Le problème LAPCS(croisés, croisés) est NP-Complet.*

Démonstration. LAPCS(croisés, croisés) est dans NP et d'après la proposition 6, le problème est NP-dur. □

Proposition 7 (Réduction). *Le problème des ensembles indépendants peut-être réduit en temps polynomial au problème LAPCS(croisés, sans arc).*

Démonstration. Voici le codage utilisé pour construire une instance du problème LAPCS(croisés, sans arc) à partir d'une instance du problème des ensembles indépendants. (voir [23] pour la preuve complète).

Pour faire cette réduction, on part d'un graphe $G(V,E)$ à n sommets et d'un entier k. On construit alors $S_1(t_1, P_1)$ et $S_2(t_2, P_2)$ comme suit :

- $t_1 = (ba^n)^n$
- $P_1 = \{((u-1)(n+1)+v+1, (v-1)(n+1)+u+1)|(u,v) \in E\}$: pour chaque arc (u,v), on crée un arc entre le $u^{\text{ème}}$ "a" du $v^{\text{ème}}$ bloc (ba^n) et le $v^{\text{ème}}$ "a" du $u^{\text{ème}}$ bloc.

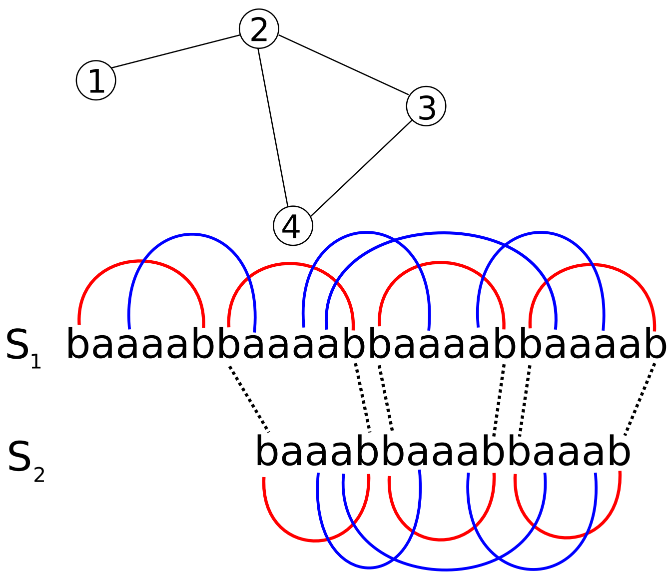


FIG. 2.6 – Exemple de séquences construites à partir d'un graphe avec $k = 3$ selon le codage utilisé pour faire la réduction du problème de recherche de clique à *LAPCS(croisés,croisés)*.

– $t_2 = (b^{n-k+1}a^n)^k n^{n-k}$
– $P_2 = \emptyset$

□

Théorème 3 (Complexité de LAPCS(croisés, {sans arc, successifs, imbriqués})). *Les problèmes LAPCS(croisés, sans arc), LAPCS(croisés, successifs) et LAPCS(croisés, imbriqués) sont NP-Complets.*

Démonstration. Le problème LAPCS(croisés, sans arc) est dans NP et la proposition 6 montre qu'il est NP-dur. De plus, LAPCS(croisés, sans arc) est une restriction de LAPCS(croisés,successifs) et LAPCS(croisés, imbriqués) (voir la table 2.5). Il en suit que LAPCS(croisés,successifs) et LAPCS(croisés, imbriqués) sont NP-Complets. □

Evans donne un algorithme qui résout le problème *LAPCS(croisés, croisés)* pour $S_1(t_1, P_1)$ et $S_2(t_2, P_2)$ en $O(f(k)|t_1||t_2|)$, k étant la *hauteur d'arc* des séquences en entrée et $f(k)$ une fonction exponentielle en k indépendante de $|t_1|$ et $|t_2|$. La *hauteur d'arc* à la position i désigne le nombre d'arcs "passant" par ou se terminant à cette position. Par extension, on désigne la *hauteur d'arc* d'une séquence comme étant le maximum des *hauteurs d'arcs* en chaque position de la séquence. L'algorithme divise le problème initial en k^2 sous-problèmes en construisant k séquences de type *successifs* à partir de S_1 et k autres séquences de type *successifs* à partir de S_2. Puis, il utilise le résultat du calcul de la plus longue sous-séquence commune entre chacune de ces séquences (k^2) pour calculer la plus longue sous séquence commune à S_1 et S_2 avec préservation des arcs.

En outre, cet algorithme résout le problème *LAPCS(successifs, Type2)*, pour Type2 égal à *successifs* ou *sans arcs*, en $O(|t_1||t_2|)$. En effet, dans le cas d'une séquence de type *successifs*, la hauteur d'arc est de 1.

En 2000, Lin *et al.* dans [48] proposent un algorithme de programmation dynamique en $O(|t_1| * (|t_2|^3))$ permettant de répondre au problème *LAPCS(S_1, S_2)* avec S_1 de type *imbriqués* et S_2 de type *successifs* ou *sans arc*. Puis en 2002, ils montrent que *LAPCS(imbriqués, imbriqués)* est NP-Complet (la preuve par réduction est trop complexe pour être résumée ici).

La table 2.7 résume les différentes complexités du problème *LAPCS* en fonction des restrictions posées sur les séquences en entrée.

Comme nous l'avons vu, la structure secondaire des ARN est modélisée par les séquences de types *imbriqués* ou *croisés* selon que l'on prend en compte ou non les pseudo-nœuds. Nous venons de voir que le problème *LAPCS* est

Type	général	croisés	imbriqués	successifs	sans arcs
général	NP-C[23]				
croisés	NP-C[23]	NP-C [23]			
imbriqués	NP-C[23]	NP-C [23]	NP-C [59]		
successifs	NP-C[23]	NP-C [23]	$O(nm^3)$[48]	$O(nm)$[23]	
sans arcs	NP-C[23]	NP-C [23]	$O(nm^3)$[48]	$O(nm)$[23]	$O(nm)$[41]

FIG. 2.7 – Complexités pour le problème *LAPCS*(S_1, S_2) avec S_1 une séquence annotée de longueur n (première colonne) et S_2 une séquence annotée de longueur m (première ligne).

NP-Complet pour cette utilisation. Cependant, pour certains problèmes, il existe des algorithmes d'approximation [35] ainsi que des algorithmes exacts performants sous certaines conditions [3].

La recherche d'un motif dans une séquence annotée.

Ce problème consiste à trouver une occurrence d'un motif donné dans une séquence annotée. On notera ce problème APS ("*arc-preserving subsequence*").

Définition 7 (APS). *Soit une séquence annotée $T(t, A)$ et une séquence annotée $P(p, B)$ que l'on nommera motif. Le problème est de répondre à la question : P est-il une sous séquence de T ? Autrement dit, existe-t-il un ensemble M de couples de positions dans S et P tel que :*
- $\forall j \in [1; |p|]$ $\exists i$ *tel que* $(i, j) \in M$ *(préservation de P).*
- $\forall (i, j) \in M$ *alors* $t[i] = p[j]$ *(conservation des symboles).*
- $\forall (i, j)(i', j') \in M$ *:*
  - *Si* $i < i'$ *alors* $j < j'$ *(conservation de l'ordre).*
  - *Si* $(i, i') \in A$ *alors* $(j, j') \in B$ *(conservation des arcs).*

Une autre façon de poser ce problème est [34] : est-il possible d'obtenir P à partir de T en retirant un nombre arbitraire de symboles de T (lorsqu'un symbole est détruit, alors tous les arcs l'ayant comme extrémité sont détruits aussi).

En fait ce problème est un cas particulier du problème *LAPCS* vu précédemment. En effet, *APS(T,P)* est vrai si *LAPCS(T,P)=P*.

Un exemple de ce problème est représenté dans la figure 2.8.

De même que pour *LAPCS*, on notera *APS(Type1, Type2)* le problème *APS* pour une séquence annotée de type *Type1* et un motif de type *Type2*,

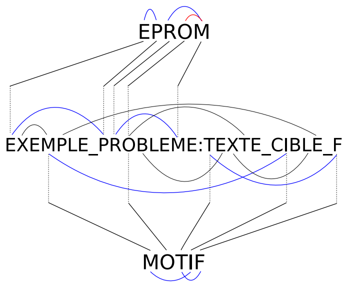


FIG. 2.8 – Exemple de la recherche d'une sous-séquence dans une séquence annotée. La séquence $EPROM$ n'est pas dans le texte (l'arc entre O et M n'est pas préservé). Par contre, la séquence $MOTIF$ est présente dans le texte.

le type *Type2* étant inclut dans le type *Type1*. *Type1* et *Type2* peuvent être *général, croisés, imbriqués, successifs* ou *sans arcs*. Il y a une relation d'inclusion entre chaque problème APS pour les différents types de séquences en entrée, celle-ci est la même que pour $LAPCS$ (voir table 2.5).

En utilisant la même réduction des ensembles indépendants à *LAPCS(général, sans arc)*, on montre que *APS(général, sans arcs)* est NP-Complet. De même, on montre que *APS(croisés, croisés)* est NP-Complet par réduction à partir du problème de calcul de clique.

La NP-Complétude de *APS(croisés, successifs)* a été prouvée en 2002 par Guo dans [35]. La preuve passe par une réduction du problème des ensembles indépendants. Le codage utilisé est similaire à celui employé dans la réduction de clique à *LAPCS(croisés, croisés)*.

Très récemment, Blin et al. [11] ont montrés que *APS(croisés, sans arcs)* est NP-Complet.

Enfin, Gramm [34] donne trois algorithmes de programmation dynamique en $O(nm)$ pour résoudre les problèmes *APS(imbriqués, successifs)*, *APS(imbriqués, sans arcs)* et *APS(imbriqués, imbriqués)*, avec n la taille de la séquence et m la taille du motif.

Le tableau 2.9 résume les complexités en temps du problème APS pour les différents types de S et de P.

Pour clore cette section sur la recherche de motif, nous allons voir les travaux de Bafna *et al.* [7] sur les séquences annotées dans le contexte de

Type	général	croisés	imbriqués	successifs	sans arcs
général	NP-C[23]				
croisés	NP-C[23]	NP-C [23]			
imbriqués	NP-C[23]	NP-C [35]	$O(nm)$[11]		
successifs	NP-C[23]	NP-C [35]	$O(nm)$[11]		
sans arcs	NP-C[23]	NP-C [11]	$O(nm)$[11]	$O(n+m)$	$O(n+m)$

FIG. 2.9 – Complexités pour le problème APS entre une séquence annotée de taille n et un motif de taille m.

la modélisation des ARN. Dans ce cadre, les arcs sont uniquement de type imbriqués.

Le problème posé est celui de la recherche de l'occurrence d'un mot dans un texte avec conservation exacte de la structure secondaire (*i.e.* des arcs) ou bien une conservation partielle de cette structure.

Définition 8 (Occurrence exacte et symétrique [7]). *Soient un ARN $T(t, A)$ et un motif $P(p, B)$, le problème consiste à trouver toutes les positions i (aussi appelées occurrences de P dans T) telles que :*
- $\forall j \in [1; |p|]\ t[i+j] = p[j]$.
- $(k; l) \in B$ *si et seulement si* $(i+k, i+l) \in A$.

Définition 9 (Occurrence exacte non symétrique [7]). *Soient un ARN $T(t, A)$ et un motif $P(p, B)$, le problème consiste à trouver toutes les positions i (aussi appelées occurrences de P dans T) telles que :*
- $\forall j \in [1; |p|]\ t[i+j] = p[j]$.
- $\forall (k; l) \in B$, $(i+k, i+l) \in A$.

Ces deux problèmes se résolvent en temps polynomial.

Plus exactement, le problème de la recherche d'occurrence exacte et symétrique se fait en $O(|t| + |p|)$ par une modification de l'algorithme de Knut-Morris-Pratt [51] de recherche de motif dans un texte.

Le problème de la recherche d'occurrence exacte non symétrique se fait en $O(|t| \log |p|)$. L'algorithme utilisé se déroule en deux temps.

Dans un premier temps, on trouve toutes les positions dans t où p n'apparaît pas (sans tenir compte des arcs), ce qui se fait en $O(|t| + |p|)$.

La deuxième phase consiste à trouver toutes les positions dans t où la structure secondaire (les arcs) ne concorde pas. Cette phase est accomplie en construisant deux nouvelles chaînes t' et p' telles que : pour chaque position

i dans t, si (k, i) ou $(i, k) \in A$ alors $t'[i] = k - i$ sinon $t'[i] = t[i]$. On fait de même pour construire p' sauf que si aucun arc n'a pour extrémité i alors $p'[i]$ est un caractère joker. Puis on utilise un algorithme de recherche avec joker pour trouver toutes les positions où la structure secondaire ne concorde pas, ce qui se fait en $O(|t| \log |p|)$.

Une variante de ces problèmes consiste à autoriser un certain nombre d'erreurs dans l'occurrence du motif. Ainsi pour une occurrence i, on autorise à ce que $t[i \ldots i + |m|]$ et p diffèrent en au plus k positions. Bafna *et al.* donnent deux algorithmes permettant de résoudre le problème de la recherche d'occurrence avec k erreurs, symétrique et non symétrique, en $O(nm^{\frac{2}{3}} \log m)$ et $O(n\sqrt{m} \log m)$, où n est la taille du texte et m celle du motif.

L'utilisation des séquences annotées pour les ARN dans le cas où l'on ne prend pas en compte les pseudo-nœuds est donc tout à fait envisageable en pratique pour la recherche d'une sous-structure commune. Cependant, cela suppose que l'on connaît la sous structure à chercher et que celle-ci se présente de manière exacte dans la séquence dans le cas du problème *APS*.

Les deux derniers problèmes intègrent la notion d'erreurs permettant de prendre en compte les phénomènes de type mutation (une base est remplacée par une autre).

Or, en pratique, on rencontre souvent des différences entre des ARN proches qui ne peuvent être pris en compte par ces approches. Ces différences peuvent être, par exemple, l'absence d'une paire de base dans une hélice ou bien même l'absence d'une hélice complète. Ceci se traduit au niveau des séquences par des "trous". Le problème *APS* permet gérer ces "trous" mais pas les erreurs au niveau des symboles des séquences.

C'est pourquoi, afin de mieux prendre en compte ces réalités biologiques, le problème d'édition de deux séquences annotées par des arcs a été introduit.

Édition de deux séquences annotées par des arcs

Nous allons dans un premier temps présenter le problème de l'édition pour les chaînes de caractères. Puis nous montrerons comment étendre cette notion aux séquences annotées.

Édition de deux chaînes de caractères
La notion d'édition [56][76][36] entre deux chaînes de caractères introduit un formalisme permettant de passer d'une chaîne à une autre au moyen de 3 opérations élémentaires :

Définition 10 (Opérations d'édition). *Soit une chaîne t sur un alphabet Σ. On note ϵ le mot vide. On définit trois opérations d'édition :*

1. *La* **substitution** *: Changement d'une lettre 'a' de t à une position donnée par une autre lettre 'b', notée $\sigma(a, b)$.*
2. *La* **délétion** *: suppression d'une lettre de t, notée $\sigma(a, \epsilon)$.*
3. *L'***insertion** *: insertion d'une lettre dans t, notée $\sigma(\epsilon, a)$.*

Munis de ces trois opérations, nous pouvons définir l'édition d'une chaîne en une autre de la manière suivante :

Définition 11 (Édition). *Soient deux chaînes t et t' sur Σ^*. On dit qu'une suite d'opérations d'édition réalise l'édition de t en t' si, en appliquant chacune des opérations de cette suite à t, on obtient la chaîne t'.*

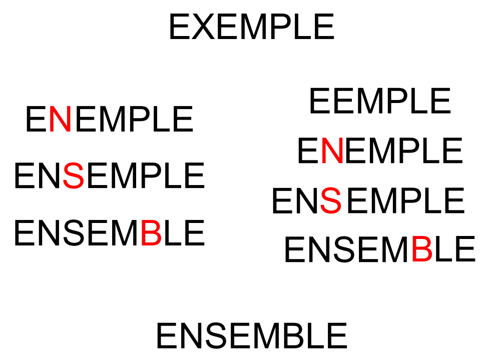


FIG. 2.10 – Deux suites d'opérations d'édition réalisant l'édition de la chaîne "EXEMPLE" en la chaîne "ENSEMBLE". La première suite est $\{\sigma(E, E);\ \sigma(X, N);\ \sigma(\epsilon, S);\ \sigma(E, E);\ \sigma(M, M);\ \sigma(P, B);\ \sigma(L, L);\ \sigma(E, E)\}$ la deuxième est $\{\sigma(E, E);\ \sigma(X, \epsilon);\ \sigma(\epsilon, N);\ \sigma(\epsilon, S);\ \sigma(E, E);\ \sigma(M, M);\ \sigma(P, B);\ \sigma(L, L);\ \sigma(E, E)\}$. Dans le cas des fonctions de coût suivantes : $\sigma(a, a) = 0$, $\sigma(a, b) = 1$, $\sigma(a, \epsilon) = 1$ et $\sigma(\epsilon, a) = 1$, la première suite totalise un coût de 3 (0+1+1+0+0+1+0+0) et la deuxième un coût de 4 (0+1+1+1+0+0+1+0+0). La distance d'édition entre ces deux séquences est de 3.

La figure 2.10 illustre l'édition de deux chaînes. On notera qu'il existe un grand nombre de suites réalisant l'édition d'une chaîne t en une autre chaîne.

C'est pourquoi nous allons affecter un poids à chacune de ces opérations. Ainsi on définit trois fonctions de score *sub*, *ins* et *del* qui, à chaque opération d'édition, associent un score. De plus, si l'on dispose d'une suite d'opérations S, on notera $score(S)$ la somme des coûts des opérations composant S. Maintenant que l'on peut associer un score à une suite d'opérations d'édition, nous pouvons définir la distance d'édition entre deux chaînes :

Définition 12 (Distance d'édition). *Soient deux chaînes t et t', et soit E l'ensemble des suites réalisant l'édition de t en t'. La distance d'édition notée $D(t,t')$ est le minimum des scores de ces suites :*

$$D(t,t') = Min(score(S)|S \in E)$$

Cette distance d'édition est souvent appelée distance de Levenshtein. Pour que D soit une distance, il faut que les fonctions de scores satisfassent à certaines conditions :

Proposition 8 (Conditions sur les fonctions de score[18]). *D est une distance sur Σ^* si et seulement si $sub(a,b)$ (substitution) est une distance sur Σ et $del(a) = ins(a) > 0$ (délétion et insertion) pour tout $a \in \Sigma$.*

Démonstration. L'implication se démontre facilement, $D(a,b) = score(a,b)$ donc $sub(a,b)$ est une distance. De plus $D(a,\epsilon) = D(\epsilon,a) = del(a) = ins(a)$ car D est une distance. Ceci implique que $del(a) = ins(a) > 0$ pour tout $a \in \Sigma$.

Pour montrer la réciproque, nous devons prouver les 4 propriétés que doit vérifier D pour être une distance :

1. **Positivité :** D est positive car c'est la somme de coûts eux mêmes positifs.
2. **Séparation :** Si $D(u,v) = 0$ alors $u = v$ car seul le coût de la substitution peut-être nul et *sub* est une distance sur Σ donc $sub(a,b) = 0$ implique que $a = b$. Pour la même raison si $u = v$ alors $D(u,v) = 0$.
3. **Symétrie :** On doit montrer que $D(u,v) = D(v,u)$. Si $D(u,v)$ correspond à la suite d'opérations d'édition $\{s_1 \dots s_n\}$, alors on peut construire la suite $\{s'_n \dots s'_1\}$ telle que pour tout $i \in [1;n]$
   - si $s_i = sub(a,b)$ alors $s'_i = sub(b,a)$
   - si $s_i = del(a)$ alors $s'_i = ins(a)$
   - si $s_i = ins(a)$ alors $s'_i = del(a)$

Par construction la suite d'opérations $\{s'_n \dots s'_1\}$ réalise l'édition de v en u. Comme le coût de l'insertion est égal à celui de la délétion et que sub est symétrique, le coût associé à $\{s'_n \dots s'_1\}$ est égal à $D(u, v)$. Enfin, le score de $\{s'_n \dots s'_1\}$ est minimal par construction et donc $D(u, v) = D(v, u)$.

4. **Inégalité triangulaire :** Nous devons montrer que, quel que soit le mot $w \in \Sigma^*$, $D(u, w) + D(w, v) \leq D(u, v)$. Supposons qu'il existe un tel mot, alors si l'on construit la suite d'opérations résultant de la concaténation des opérations réalisant l'édition de u en w et de celle réalisant l'édition de w en v, cette suite réalise l'édition de u en v et son coût est inférieur à $D(u, v)$ ce qui contredit la définition de D.

□

La distance d'édition entre deux séquences peut-être calculé par un algorithme de programmation dynamique. Soient deux chaînes t et t' sur un alphabet Σ de longueur n et m. On note $t[i]$ le $i^{\text{ème}}$ caractère de t, et $t[i \dots j]$ le facteur de t composé des caractères $t[i]t[i+1] \dots t[j]$. L'algorithme repose sur la relation suivante :

$$\boxed{D(t, t') = D(t[1 \dots n], t'[1 \dots m]) =}$$

Si $t \neq \epsilon$ et $t' \neq \epsilon$

$$Min \begin{cases} D(t[1 \dots n-1] & , t' &) + Del(t[n]) \\ D(t & , t'[1 \dots m-1] &) + Ins(t'[m]) \\ D(t[1 \dots n-1] & , t'[1 \dots m-1] &) + Sub(t[n], t'[m]) \end{cases} \tag{2.1}$$

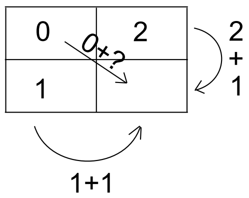
Sinon

$$D(\epsilon, \epsilon) = 0$$

On pourra se référer à [36] pour une preuve de la validité de cet algorithme.

Proposition 9 (Complexité). *La distance d'édition entre deux chaînes de longueurs n et m sur un alphabet Σ peut être calculée à l'aide d'un algorithme de programmation dynamique en $O(nm)$ temps et $O(min\{n, m\})$ espace.*

Démonstration. En effet, comme le montre la figure 2.11, l'algorithme consiste à calculer l'ensemble des valeurs d'une matrice de taille $(n + 1) * (m + 1)$, le calcul d'une de ces valeurs se faisant en temps constant. De plus, on peut remarquer que si l'on effectue le calcul de cette matrice ligne après ligne, on peut réaliser l'ensemble du calcul dans un tableau de la taille d'une ligne plus un élément. On peut utiliser le même procédé en ne conservant qu'une seule colonne plus un élément durant le calcul. Il en résulte la complexité en mémoire annoncée. □



		E	X	E	M	P	L	E
	0	1	2	3	4			
E	1	0	1	2	3			
N	2	1	1	2	3			
S	3	2	2	2	3			
E	4	3	3	2	3			
M	5	4	4	3	3			
B	6	5	5	4				
L	7	6	6	5				
E	8	7	7	6				

FIG. 2.11 – Calcul de la distance d'édition entre deux chaînes selon un algorithme de programmation dynamique.

Une notion liée à celle de l'édition entre deux séquences est l'alignement de deux séquences.

Définition 13 (Alignement [36]). *L'alignement de deux séquences S_1 et S_2 consiste à insérer des blancs (caractère spécial noté '-' n'appartenant pas à Σ) dans S_1 et S_2, éventuellement aux extrémités. On obtient alors deux séquences S'_1 et S'_2 telle que :*
- *$l = |S'_1| = |S'_2|$: S'_1 et S'_2 sont de même longueur.*
- *$\forall i \in [1; l]$, si $S'_1[i] =' -'$ alors $S'_2[i] \neq' -'$.*

Les séquences S'_1 et S'_2 sont un alignement des séquences S_1 et S_2.

La figure 2.12 montre l'alignement des deux séquences "plante" et "depanner".

En fait, on voit tout de suite l'équivalence entre l'édition et un alignement. En effet, si on dispose d'un alignement entre deux séquences, lorsque deux caractères sont alignés (l'un en dessous de l'autre), cela correspond à une substitution, si un caractère de la première séquence est aligné avec un blanc, cela correspond à la délétion de ce caractère, l'inverse étant l'insertion.

position :	1	2	3	4	5	6	7	8	9
S_1			p	l	a	n	t	e	
S'_1	-	-	p	l	a	n	t	e	-
S'_2	d	e	p	-	a	n	n	e	r
S_2	d	e	p		a	n	n	e	r

FIG. 2.12 – Alignement des séquences "plante" et "depanner". On peut construire la séquence d'édition suivante pour passer de S_1 à S_2 : $\{\sigma(\epsilon, d)\,;\sigma(\epsilon, e)\,;\ \sigma(p, p)\,;\sigma(l, \epsilon)\,;\sigma(a, a)\,;\sigma(n, n)\,;\sigma(t, n)\,;\sigma(e, e)\,;\sigma(\epsilon, r)\}$.

En suivant le même raisonnement, on peut facilement construire l'alignement de deux séquences à partir du calcul de leur distance d'édition.

Édition de deux séquences annotées

Plusieurs travaux ont étendu la notion d'édition sur les séquences décrites précédemment aux séquences annotées. Nous allons présenter trois approches de l'édition de deux séquences annotées. Chacune de ces approches introduit de façon différente de nouvelles opérations d'éditions agissant au niveau des arcs de la séquence.

Dans la suite, nous nous plaçons dans le cadre de la comparaison des ARN par la calcul de l'édition entre deux séquences annotées. Ainsi, nous ne considérerons que les séquences annotées de type *croisés* (pour chaque position de la séquence, il y a au plus un arc dont une des extrémités aboutit à cette position) ou un de ses sous types.

Première approche : traitement séparé des symboles et des arcs

Les premiers travaux [7] ont consisté à faire l'édition sur séquences annotées à l'aide des 4 opérations d'édition suivantes : substitution, délétion, insertion de symbole et substitution d'arc. La substitution d'arc est une opération supplémentaire ajoutée lorsque l'édition met en correspondance deux paires de bases qui sont connectées par un arc dans une séquence mais pas dans l'autre. La figure 2.13 illustre ces quatre opérations. Il en résulte un algorithme de programmation dynamique calculant l'alignement de deux séquences annotées en $O(n^2m^2)$ avec n et m la longueur des deux séquences.

Cette approche présente l'inconvénient de traiter séparement les arcs et les symboles de la séquence reliés par ces arcs. Du point de vue de la structure secondaire ceci revient à traiter la liaison entre deux nucléotides sans considérer les deux nucléotides impliqués par cette liaison. Comme nous allons le voir, d'autres opérations d'éditions pour les séquences annotées permettent une gestion plus fines des arcs de la séquence et donc des liaisons nucléotidiques.

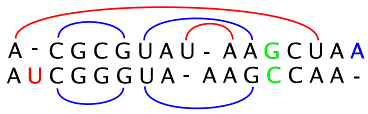


FIG. 2.13 – Exemple d'alignement de deux séquences annotées utilisant la première approche. Les 3 opérations classiques sont représentées par les lettres colorées. Les arcs rouges correspondent à des substitutions d'arcs.

Deuxième approche : bases appariées ou non appariées

Plus tard, en 1999, Zhang *et al.* proposèrent dans [99][66] une définition de l'édition entre deux séquences annotées utilisant des opérations d'éditions opérant sur les symboles de la séquence annotée et des opérations opérant sur les arcs de la séquence annotée.

On dit qu'une position/lettre/caractère est libre si aucun arc n'a pour extrémité cette position. Dans le cas contraire, on dit qu'elle est appariée ou liée.

Comme pour les séquences, les trois opérations d'édition sont la substitution, la délétion et l'insertion que l'on peut appliquer, soit à une lettre libre, soit à une lettre appariée.

On a donc 6 opérations d'édition : dans le cas où une position est libre, les opérations d'édition sont définies de la même manière que les opérations d'édition sur les séquences. Considérons maintenant l'arc $(i, j) \in A$ de la séquence annotée $T(t, A)$. La substitution appliquée à cet arc consiste à remplacer $t[i]$ et $t[j]$, la délétion consiste à retirer (i, j) de A ainsi que $t[i]$ et $t[j]$ de t (délétion des symboles) et l'insertion revient à ajouter deux symboles dans t ainsi que l'arc qui les lie dans A.

À chacune de ces opérations est affecté un coût et donc le problème est de trouver la suite d'opérations d'édition permettant de passer d'une séquence annotée à une autre dont le coût est minimal.

On remarque tout de suite la différence qu'il y a entre la précédente approche et celle-ci : dans le cas présent, les symboles libres et les symboles liés sont considérés comme deux types d'objets bien distincts disposant d'opérations d'édition spécifiques. Ainsi, contrairement à l'approche

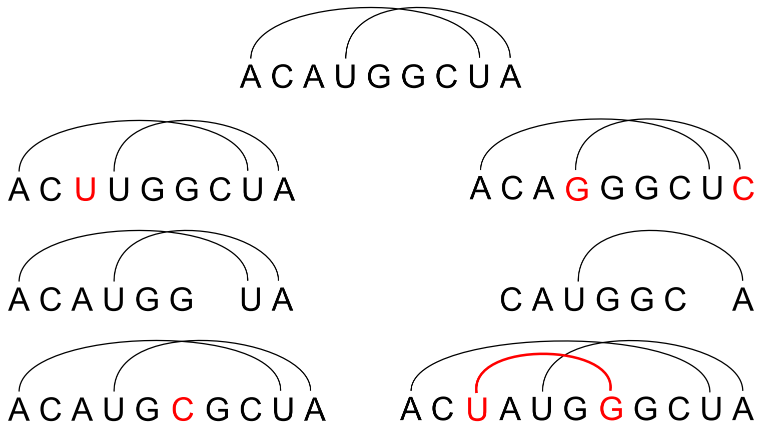


FIG. 2.14 – Les opérations d'édition de la deuxième approche : à gauche la substitution, la délétion et l'insertion sur les bases et à droite la substitution, la délétion et l'insertion sur les arcs.

précédente, un arc ne peut être aligné avec deux bases libres qu'en supprimant cet arc (et donc les symboles aux extrémités de celui-ci) et en insérant deux nouveaux symboles dans la séquence. Du point de vue de la structure secondaire, cela revient à considérer qu'une paire de bases ne peut pas être associée à deux bases libres.

Dans le cas où les deux séquences annotées sont de type *croisés*, le problème du calcul de leur distance d'édition est NP-Complet [99]. En effet, on montre que le problème 3-SAT peut être réduit en temps polynomial au calcul de la distance d'édition entre deux séquences annotées de type *croisés*.

Définition 14 (3-SAT). *Soient n clauses $C_1 \dots C_n$, chacune composée de 3 littéraux. Le problème est : Existe-t-il des valeurs pour ces variables telles que les clauses soient satisfaites ?*

Proposition 10 (Réduction). *Le problème 3-SAT peut être réduit en temps polynomial au problème du calcul de la distance d'édition entre deux séquences annotées de type* croisés.

Démonstration. Pour la preuve, on se référera à [99]. □

Dans [17], Collins *et al.* proposent un algorithme polynomial permettant de calculer une distance d'édition "contrainte" entre deux séquences annotées de type *croisés*. La donnée du problème est deux séquences annotées ainsi qu'un ensemble E de couples de positions sur les deux séquences. De plus E doit être tel que si l'on supprime dans les deux séquences les arcs mis en relation par E alors au plus une des deux séquences reste de type *croisés*. L'algorithme permet alors de calculer l'édition des deux séquences telle que pour chaque couple (i, j) de E, la $i^{\text{ème}}$ lettre de la première séquence est associée à la $j^{\text{ème}}$ lettre de la deuxième séquence.

Dans le cas où les deux séquences sont de type *imbriqués*, celles-ci peuvent être vues comme des arbres. Nous verrons dans le chapitre consacré à la modélisation utilisant les arbres que les opérations d'édition définies ici sont une transposition des opérations d'édition définies sur les arbres. C'est pourquoi nous ne donnons pas ici l'algorithme d'édition qui est le même que celui pour les arbres.

Enfin, si l'une des séquences est de type *croisés* et l'autre de type *imbriqués*, alors on peut calculer leur distance d'édition à l'aide d'un algorithme de programmation dynamique. La complexité en temps de l'algorithme est $O(m^2 n \log n)$ avec m et n la taille des séquences et $m < n$ [99].

Cette approche de l'édition de séquences annotées est directement inspirée de l'édition d'arbre que nous verrons dans le prochain chapitre. En effet, il y a une totale équivalence entre les séquences annotées de type *imbriqués* et les arbres ordonnés. Les arbres ne pouvant prendre en compte des interactions tertiaires (pseudo-nœuds), Zhang *et al.* [99] ont reformulé le problème de l'édition d'arbre avec les séquences annotées. Nous verrons que la définition des opérations d'édition donnée ici est équivalente aux opérations d'édition sur les arbres. Ceci apparaît clairement du fait que les opérations définies sur les séquences traitent un arc et les symboles à ces extrémités comme une seule entité : les opérations sur les arcs agissent simultanément sur l'arc et ses symboles. En effet, dans le cas des arbres, les arcs et les symboles à leurs extrémités forment une seul entité : un nœud. Enfin, l'algorithme calculant l'édition entre une séquence de type *croisés* et une séquence de type *imbriqués* utilise une formule de récurrence très proche de celle que nous présenterons pour le calcul de la distance entre deux arbres.

Concluons sur le fait que le type *croisés* nous permet de prendre en compte uniquement la structure secondaire ainsi que les pseudo-nœuds dans la modélisation et non l'ensemble des interactions tertiaires possibles.

Voici une dernière approche où les opérations d'édition permettent plus de souplesse dans le traitement des arcs.

Troisième approche : opérations dédiées aux arcs

Cette approche de l'édition de deux séquences annotées a été proposée par Lin *et al.* en 2001 dans [60] et [47]. L'édition de deux séquences annotées repose alors sur 7 opérations d'édition.

Ainsi, dans [60], Lin *et al.* définissent les 4 opérations suivantes sur les arcs :

- La substitution d'arc consiste à changer les 2 symboles aux extrémités d'un arc. Le coût de cette opération est noté w_{am}.
- L'altération d'arc correspond à la délétion d'un des symboles aux extrémités d'un arc. Il en résulte que ce symbole est aligné avec un blanc '-'. Le coût de cette opération est w_a.
- La délétion d'arc : les deux symboles incidents à un arc sont supprimés. Son coût est w_r.
- La cassure d'arc : un arc est supprimé, les symboles incidents à celui-ci deviennent libres. Le coût de l'opération est w_b.

De plus, on dispose des opérations d'édition classiques, à savoir la substitution de coût w_m, la délétion de coût w_d et l'insertion dont le coût est aussi w_d.

Bien qu'elles n'aient pas été définies dans [60], de même que l'insertion est l'opération inverse de la délétion, l'altération d'arc, la délétion d'arc et la cassure d'arc possèdent leurs opérations inverses dont le coût est le même afin de conserver la propriété de symétrie. Ainsi cette définition de l'édition de séquence annotée utilise onze opérations d'édition.

De même que pour les problèmes APS et LAPCS, on désignera par *EDIT(Type1,Type2)* l'édition entre une séquence de type *Type1* et une séquence de type *Type2*. Ainsi, dans le cadre de la comparaison d'ARN, nous devons étudier le problème pour *Type1* et *Type2* égaux à *croisés*, *imbriqués*, *successifs* et *sans arcs*. Signalons que l'édition classique décrite au début de cette section répond au problème *EDIT(sans arcs,sans arcs)*.

Lin *et al.* ont montré que le problème de *coupe maximale dans un graphe cubique* peut être réduit en temps polynomial au problème EDIT(croisés, sans arc).

Définition 15 (Coupe maximale dans un graphe cubique). *Soient un graphe $G(V, E)$, une partition de V en deux ensembles V_1 et V_2. Une* coupe *de G est un sous-ensemble d'arcs de E tel que chaque arc de cet ensemble a une extrémité dans V_1 et l'autre dans V_2.*

Le problème de la coupe maximale dans un graphe cubique est : étant donné un graphe cubique $G(V, E)$ (chaque sommet est de degré trois), trou-

ver une partition de V telle que la coupe *qui lui est associée soit de taille maximale.*

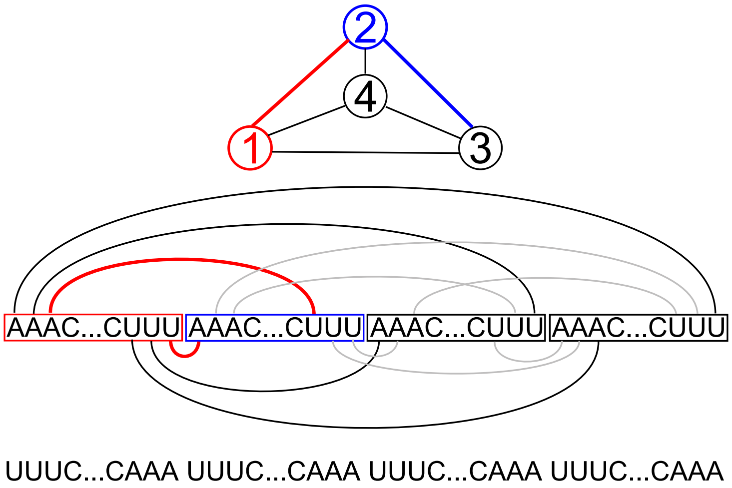


FIG. 2.15 – Séquence construite à partir d'un graphe cubique selon le codage utilisé dans la réduction du problème de *coupe maximale dans un graphe cubique* à *EDIT(croisés, sans arc)*.

Proposition 11 (Réduction). *Le problème de* coupe maximale dans un graphe cubique *peut être réduit en temps polynomial au problème EDIT(croisés, sans arc).*

Démonstration. Voici le codage utilisé, le reste de la preuve est dans [60].

On dispose d'un graphe $G(V, E)$ avec $n = |V|$. On définit deux séquences annotées $S_1(t_1, P_1)$ et $S_2(t_2, P_2)$ de la façon suivante :

- $t_1 = (AAAUUU(C)^x)^{n-1}AAAUUU$, t_1 est composé de n blocs de la forme $AAAUUU$ séparés par x lettres 'C', la définition de x est donnée plus bas.
- $t_2 = (UUUAAA(C)^x)^{n-1}UUUAAA$, t_2 est composé de n blocs de la forme $UUUAAA$ séparés par x lettres 'C'.
- P_1 : Pour chaque couple (i, j), on crée un arc entre un des 'A' du bloc i et un des 'U' du bloc j ainsi qu'entre un des 'U' du bloc i et un des 'A' du bloc j (voir figure 2.15).

– $P_2 = \emptyset$.

x est une valeur dépendant des fonctions de coûts utilisées égale à $max\{2nw_d, \frac{min(w_b,w_a,w_r)}{w_d}\}$. Ces séries de 'C' servent à s'assurer que l'édition des deux séquences n'associera pas des bases de deux blocs s'il n'y a pas d'arc entre les sommets correspondants à ces blocs. □

On en déduit,

Théorème 4 (Complexité de EDIT(croisés, Type2)). *Les problèmes EDIT(croisés, Type2) sont NP-Complets quel que soit le type Type2 de la deuxième séquence.*

Démonstration. Il est clair que le problème EDIT(croisés, sans arcs) est dans NP, et d'après la proposition précédente, ce problème est NP-dur car *coupe maximale dans un graphe cubique* est NP-dur et il peut être réduit à *EDIT(croisés, Type2)* en temps polynomial.

De plus, EDIT(croisés, sans arcs) est une restriction de EDIT(croisés, Type2) pour tout type Type2. □

Un algorithme de programmation dynamique pour le calcul de *EDIT(imbriqués, sans arc)* a été proposé dans [60], sa complexité est $O(nm^3)$ avec n la taille de la séquence de type *imbriqués* et m la longueur de la séquence *sans arc*. Cependant, cette distance d'édition est rarement utile dans le cas de la comparaison d'ARN. En effet, cela revient à comparer un ARN et sa structure secondaire avec la séquence primaire d'un autre ARN.

Le cas *EDIT(imbriqués,imbriqués)*, correspondant à la comparaison de deux structures secondaires, est NP-Complet. La preuve, trop compliquée pour être résumée ici, pourra être trouvée dans [10].

Enfin, on notera aussi un algorithme de programmation dynamique permettant de faire le calcul de *EDIT(croisés, imbriqués)* en temps polynomial pour certaines fonctions de coût. En effet, Lin *et al.* montrent que si $2w_a = w_b + w_r$ alors toutes les opérations sur les arcs précédemment décrites seront réalisées avec l'opération de cassure d'arc. L'algorithme qui en découle est similaire à celui permettant le calcul de *EDIT(imbriqués, sans arc).*

Cette troisième approche, bien que plus précise que les précédentes dans le traitement des arcs, ne permet pas de résoudre efficacement le problème de la comparaison d'ARN.

Conclusion sur l'édition

Parmi ces trois approches, il semblerait que la deuxième soit la plus pertinente pour la comparaison des ARN. En effet, contrairement à la premiere approche, elle considère les "paires de bases" (symboles liés par des arcs) et les "bases non appariées" (symboles sans arc) comme deux entités bien distincts, ce qui semble pertinent vis à vis des structures secondaires d'ARN. La troisième approche définie des opérations plus complexes sur les arcs permettant entre autre le passage d'une "paire de bases" à une "base non appariées" cependant l'utilisation de cette édition pour la comparaison de structures secondaires nous mène à un problème NP-Complet.

Nous reviendrons sur cette distance dans le chapitre sur les arbres. La troisième approche, bien que plus fine dans la gestion des arcs, n'est pas utilisable pour la comparaison des ARN, le problème étant NP-Complet.

2.1.3 2-intervalles : une généralisation des séquences annotées

Dans [88], Vialette introduit les 2-intervalles pour la modélisation des ARN. Dans un premier temps, nous donnerons la définition d'un ensemble de deux intervalles et décrirons son utilisation pour la modélisation de la structure secondaire des ARN. Puis nous étudierons le problème de la recherche de motif sur les 2-intervalles.

Présentation des 2-intervalles

Un 2-intervalle est un couple d'intervalles disjoints. Deux 2-intervalles $D(I_1, I_2)$ et $D'(J_1, J_2)$ sont disjoints ou *comparables* si $(I_1 \bigcup I_2) \bigcap (J_1 \bigcup J_2) = \emptyset$. Comme le montre la figure 2.16, on peut définir trois relations binaires possibles entre deux 2-intervalles comparables : $<$, $\sqsubset$ et $\sqcap\sqcup$ correspondant aux cas (A), (B) et (C) de la figure. Pour un sous ensemble R de $<, \sqsubset, \sqcap\sqcup$, on dit que deux 2-intervalles sont R-comparables s'ils le sont par une des relations de R.

Il apparaît assez naturellement que les 2-intervalles permettent de modéliser les hélices d'une structure secondaire d'ARN. La figure 2.17 représente l'ensemble des deux intervalles associés à un ARN de transfert.

On définit alors une famille de 2-intervalles comme étant un ensemble de 2-intervalles. On peut remarquer qu'une séquence annotée par des arcs peut-être vue comme un cas particulier de famille de 2-intervalles dans laquelle tous les intervalles sont disjoints. Une des façons de construire une famille

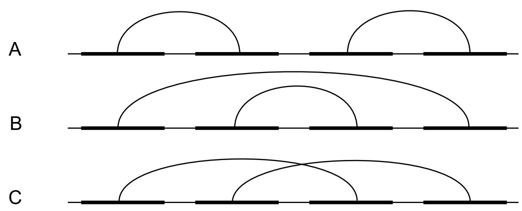


FIG. 2.16 – Relations possibles entre deux 2-intervalles comparables.

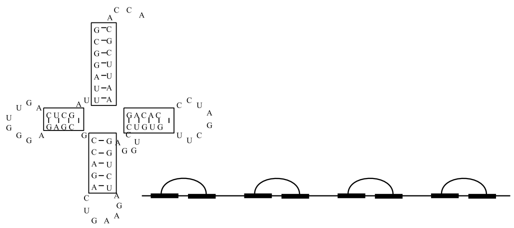


FIG. 2.17 – Ensemble de 2-intervalles modélisant la structure secondaire d'un ARN de transfert.

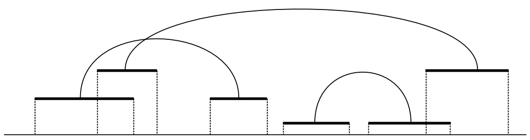


FIG. 2.18 – Exemple d'une famille de 2-intervalles.

de 2-intervalles à partir d'une séquence annotée par des arcs consiste à associer une longueur d'intervalle différente pour chaque lettre de l'alphabet de la séquence annotée. Puis on construit la suite d'intervalles disjoints correspondant à la séquence, un intervalle par lettre. Enfin, pour chaque arc de la séquence, on crée un 2-intervalle.

Recherche de sous famille maximale et recherche de motif.

Deux problèmes ont été étudiés sur les 2-intervalles.

Le premier problème consiste à trouver la plus grande sous famille F' d'une famille F de 2-intervalles telle que tous les 2-intervalles de F' sont deux à deux R-comparables pour une ensemble R de relations. Vialette a montré dans [88],[89] et [90] que ce problème est NP-complet lorsque R est $\{<, \sqsubset, \sqcap\sqcup\}$ ou $\{\sqsubset, \sqcap\sqcup\}$. Dans le cas où R est $\{<, \sqsubset\}$,$\{<\}$, $\{\sqsubset\}$ et $\{\sqcap\sqcup\}$ il fournit un algorithme polynomial pour résoudre le problème. La complexité du problème lorsque $R = \{<, \sqcap\sqcup\}$ est encore un problème ouvert à ce jour.

Le deuxième problème est la recherche d'un motif dans une famille de deux intervalles. Il correspond au problème biologique suivant : on dispose du modèle d'une famille d'ARN, c'est-à-dire d'une description des hélices communes à l'ensemble des ARN de cette famille, et de la séquence d'un ARN. La question est de savoir s'il existe parmi l'ensemble des hélices pouvant être formées sur cet ARN, un sous ensemble compatible avec le modèle. Formellement, le motif $\mathcal{M}$ est décrit par une famille de 2-intervalles disjoints $M_1, \ldots, M_k$ R-comparables deux à deux. Par analogie aux problèmes sur les séquences, on appellera texte, noté $\mathcal{T}$, une famille de 2-intervalles $T_1, \ldots, T_l$ quelconques. On définit alors une occurrence du motif dans le texte par un ensemble $\mathcal{O}$ de couples de 2-intervalles du motif et du texte tel que :

- $\forall D \in \mathcal{M}, \exists (D, F) \in \mathcal{O}$.
- $\forall (D_1, E_1), (D_2, E_2) \in \mathcal{O}$, D_1 et D_2 sont α-comparable ssi E_1 et E_2 sont α-comparable, $\alpha \in R$.
- La sous-famille de $\mathcal{T}$ formée par les 2-intervalles intervenant dans $\mathcal{O}$ est composée d'intervalles disjoints (implicite selon la deuxième condition).

Ce problème est NP-Complet dans le cas où R est $\{<, \sqsubset, \sqcup\sqcap\}$ ou $\{\sqsubset, \sqcup\sqcap\}$. Vialette [88][89][90] donne des algorithmes polynomiaux pour résoudre le problème lorsque R est $\{<, \sqsubset\}$, $\{<\}$, $\{\sqsubset\}$ ou $\{\sqcup\sqcap\}$. La recherche de motifs pour $R = \{<, \sqcup\sqcap\}$ est un problème ouvert.

2.1.4 Conclusion

Comme nous avons pu le voir, les séquences annotées permettent de correctement modéliser les ARN et leur structure secondaire, voir même tertiaire. Cependant, malgré des restrictions sur le modèle, de nombreux problèmes sur ces séquences sont NP-Complets.

En fait, nous verrons dans le chapitre sur les arbres des algorithmes efficaces permettant de comparer les ARN. Ces algorithmes, tels que le calcul de la distance d'édition, peuvent tout à fait être adaptés aux séquences annotées de types *imbriqués*. Nous avons choisi de les présenter dans le contexte des arbres car cette représentation nous paraît plus facile à manipuler.

2.2 Arbres enracinés et ordonnés

Ce chapitre est consacré à l'utilisation des arbres pour la modélisation des structures secondaires d'ARN.

La première partie de ce chapitre donne la définition de cette modélisation. La deuxième partie sera quant à elle consacrée aux algorithmes sur ces arbres. Nous nous concentrerons exclusivement sur les algorithmes de comparaison d'arbres développés dans le contexte de la bioinformatique, plus particulièrement de l'analyse des ARN.

Nous considérons ici que la structure secondaire des ARN ne prend pas en compte les pseudo-nœuds.

2.2.1 Plusieurs arbres possibles

Lorsque l'on regarde le dessin de la structure secondaire d'un ARN (figure 1.5 par exemple), il paraît naturel de représenter cette structure par un arbre. Pour construire un arbre à partir d'une structure secondaire, nous pouvons utiliser la notion d'accessibilité introduite par Zuker et Sankoff.

Définition 16 (Accessibilité [100]). *Soit une structure secondaire d'ARN dont les bases sont référencées par leurs indices d'apparitions le long de la séquence de l'ARN. Une paire de bases (ou appariement) est alors représentée par le couple de positions correspondantes aux bases de cette paire.*

Soit un appariement (i, j) $(i < j)$ de la structure secondaire, une base r est dite accessible depuis l'appariement (i, j), s'il n'existe pas de paire (k, l)

telle que $i < k < r < l < j$*. Un appariement* (r, s) *est accessible depuis* (i, j) *si* r *et* s *sont accessibles depuis* (i, j)*.*

La figure 2.19 illustre cette notion.

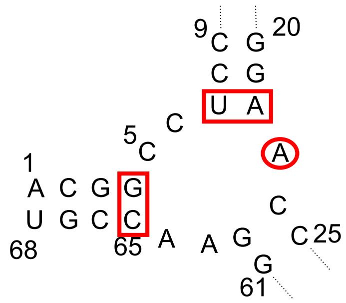


FIG. 2.19 – Exemple d'accessibilité : Les bases 5, 6, 23, 63 et 64 et les paires de base $(7, 22)$, $(24, 62)$ sont accessibles depuis $(4, 65)$. Seule la paire $(2, 67)$ est accessible depuis $(1, 68)$. La base 23 n'est pas accessible depuis $(7, 22)$ mais l'est depuis $(4, 65)$.

Ainsi, il est facile de donner la définition d'un arbre représentant la structure secondaire d'un ARN [100] :

Définition 17 (Arbre d'une structure secondaire d'ARN). *On dispose en entrée de la structure secondaire d'un ARN. L'arbre* T *modélisant cette structure est construit comme suit :*

- *À chaque base non appariée, on associe une feuille de* T *et à chaque paire de bases, on associe un nœud interne de* T*.*
- *Chaque nœud interne de* T *représentant la paire de bases* (i, j) *a pour fils l'ensemble des nœuds représentant les bases non appariées et les paires de bases accessibles depuis* (i, j)*. Ceux-ci sont ordonnés selon leur ordre d'apparition dans la séquence de l'ARN.*
- *Les forêts ainsi obtenues sont connectées au nœud* R*, enracine de* T*, et ne correspondant à aucun élément de la structure secondaire.*

T est un arbre enraciné, ordonné, c'est-à-dire que l'ordre entre les fils compte. Le fait que T soit enraciné implique que ces arcs sont orientés. Les étapes de cette construction sont illustrées dans la figure 2.20.

Il existe bien d'autres façons de coder une structure secondaire par un arbre. Ainsi, Fontana *et al.* introduisent dans [28] l'arbre homéomorphe irréductible, noté *HIT* (*"Homeomorphically irreductible tree"*). Cet arbre est une version compacte de l'arbre construit par Zuker et Sankoff.

On peut aisément voir qu'une hélice dans la structure secondaire se traduit par une suite de nœuds internes dont tous les nœuds sauf le dernier n'ont qu'un seul fils. De même, une séquence de bases non appariées se traduit par une suite de feuilles. Ainsi, on peut "compacter" l'arbre initial en remplaçant toute suite de feuilles par une seule feuille, et toute suite de nœuds internes par un seul nœud. Cependant, pour pouvoir conserver l'équivalence entre les deux structures (*i.e.* construire un arbre à partir de l'autre), on code dans chaque nœud et chaque feuille du *HIT* le nombre de nœuds correspondant dans l'arbre d'origine. Notons que cette structure est utilisée pour faire la comparaison de structures secondaires d'ARN dans la plateforme *VIENNA* [42].

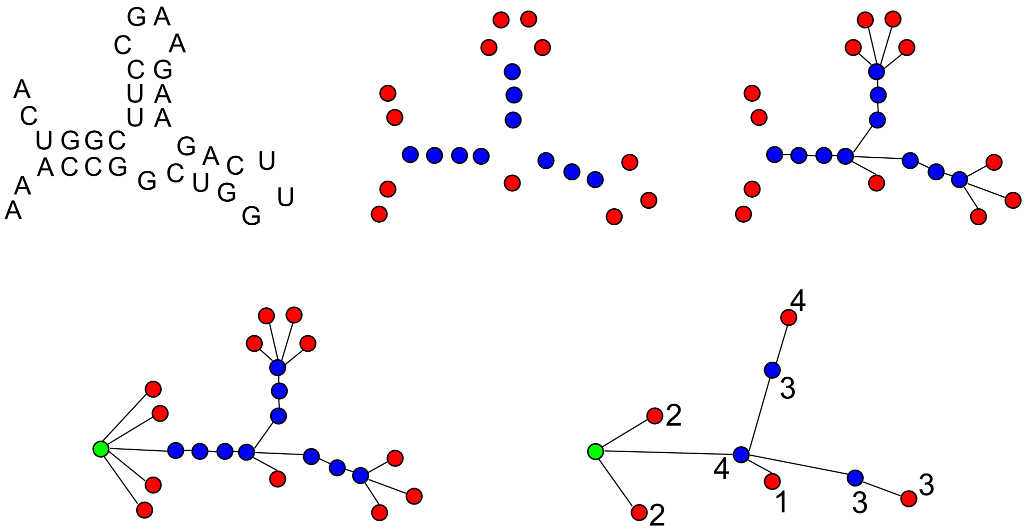


FIG. 2.20 – Étape de la construction de l'arbre codant la structure secondaire d'un ARN. En bas, à gauche, l'arbre construit selon Zuker et Sankoff [100] et à droite, l'arbre homéomorphe irréductible utilisé par Fontana *et al* [28].

Bien qu'intéressants, ces deux arbres posent néanmoins un problème : on perd l'information des symboles de la séquence lors de la construction pour ne garder que l'information portant sur les liaisons de la structure secondaire.

Ainsi, on préférera souvent utiliser un arbre étiqueté pour représenter la structure secondaire d'un ARN (2.21 (3)). Cet arbre, que nous appellerons arbre des liaisons nucléotidiques, est le même que celui de Zuker et Sankoff mais des étiquettes sont adjointes aux nœuds ainsi qu'aux feuilles. À chaque nœud de l'arbre est associée une étiquette sur $\{A, C, G, U\}^2$, et à chaque feuille un symbole sur $\{A, C, G, U\}$ correspondant au symbole présent dans la structure primaire à la position codée par cette feuille.

On peut alors étendre la notion de *HIT* aux arbres des liaisons nucléotidiques. Dans ce cas, le *HIT* est un arbre enraciné, ordonné, étiqueté dont les nœuds internes ont pour étiquettes des couples sur $\{A, C, G, U\}^*$, c'est-à-dire deux séquences nucléotidiques. Les feuilles sont étiquetées sur $\{A, C, G, U\}^*$. L'arbre (4) de la figure 2.21 en est un exemple.

D'autres types d'arbres ont été introduits dans le but de mieux modéliser la structure secondaire. L'unité à modéliser qui est considérée dans ce cas ne correspond pas aux nucléotides mais aux éléments de structures secondaires. Ainsi, Shapiro dans [78] propose de modéliser la structure secondaire des ARN par un arbre étiqueté sur $\{M, B, I, H, R\}$, ces lettres représentant, respectivement, les boucles multiples, les renflements, les boucles internes, les boucles terminales et la racine de l'arbre. L'arbre (5) est un exemple de ce codage. Cette modélisation permet de bien prendre en compte l'agencement des éléments structuraux. Comme nous l'avons déjà mentionné, deux ARN ayant une fonction commune auront une conformation spatiale commune. Cette conformation est dictée par ces éléments structuraux plus que par les nucléotides eux mêmes. Nous verrons en effet dans le prochain chapitre qu'il arrive souvent que deux structures ayant une même fonction possèdent les mêmes éléments structuraux mais ont des séquences nucléotidiques très différentes.

En suivant ce raisonnement, on peut alors imaginer d'autres types de représentations comme celles présentées dans la figure 2.21. De même que précédemment, dans chacun de ces arbres, on peut stocker un nombre varié d'informations. Par exemple, l'arbre (6) est un arbre dont les nœuds codent pour les boucles terminales et les boucles multiples d'un ARN. Les arcs de cet arbre codent alors pour les tiges de la structure secondaire. On peut donc étiqueter les arcs par la longueur des tiges, ou bien même par leur composition en $\{A, C, G, U\}$.

Dans l'ensemble de ces représentations, un certain nombre de propriétés

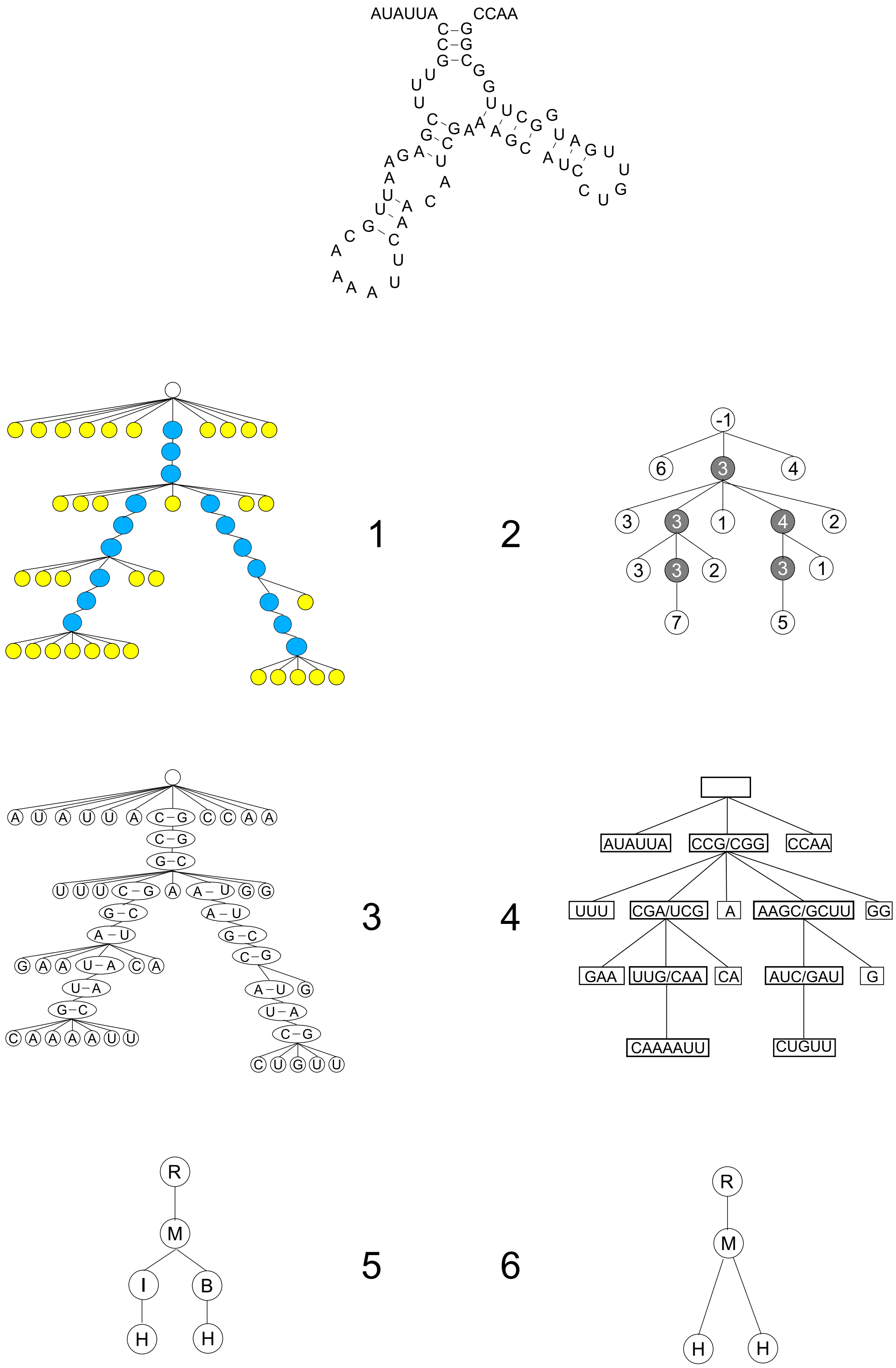


FIG. 2.21 – Exemples des types d'arbres pouvant être utilisés afin de représenter la structure secondaire d'un ARN.

subsistent : il s'agit toujours d'arbres enracinés, ordonnés et étiquetés. Nous allons donc maintenant voir des algorithmes permettant la comparaison de tels arbres.

Avant cela, pour conclure sur cette partie de modélisation d'ARN par des arbres, nous noterons la symétrie qu'il existe entre les séquences annotées de type arcs *imbriqués* et les arbres ordonnés. En effet, toute séquence annotée de type arcs *imbriqués* peut s'écrire sous la forme d'une forêt d'arbres ordonnés. Pour cela, à chaque lettre libre (aucun arc ne l'a pour extrémité) de la séquence et à chaque couple de lettres liées par un arc, on associe un nœud de l'arbre. Cette transformation est représentée dans la figure 2.22.

De même, on peut aisément définir une séquence annotée à partir d'un arbre ordonné.

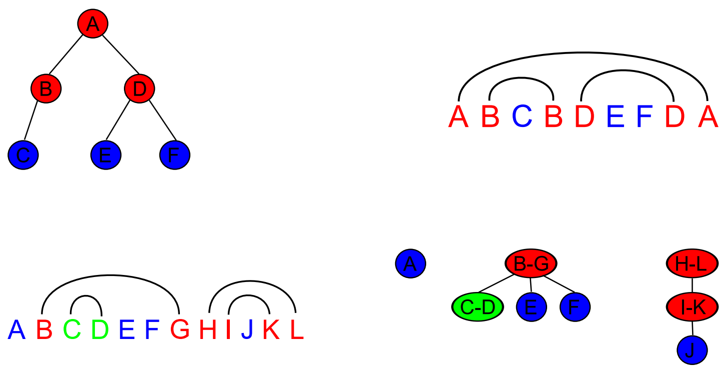


FIG. 2.22 – Exemple de séquences annotées par des arcs et d'arbres équivalents.

On peut cependant noter une différence majeure entre ces deux structures. Le nœud interne, qui ne forme qu'une entité dans l'arbre, représente deux entités dans la séquence annotée, deux symboles de la séquence ainsi qu'un arc. Cette différence est surtout remarquable dans la définition des opérations d'édition portant sur ces deux objets. Nous allons voir ceci dans la suite consacrée aux algorithmes pour la comparaison d'arbres.

2.2.2 Comparaisons d'arbres enracinés et ordonnés

Les algorithmes ci-dessous, élaborés entre autres pour la comparaison de la structure secondaire des ARN, reprennent la notion d'édition que nous avons vue au chapitre précédent. Celle-ci a été étendue aux arbres par Tai en [83]. L'algorithme le plus connu pour calculer cette distance d'édition entre arbres est celui de Zhang et Shasha [98]. Nous le présenterons dans une première partie. Puis nous verrons la notion d'alignement entre deux arbres qui, contrairement à l'alignement de séquences, n'est pas équivalent à l'édition.

On désignera par T un arbre ordonné composé de $|T|$ nœuds. On note t_i le nœud numéroté i lors d'un parcours postfixe de l'arbre. Les fils du nœud t_i seront désignés par la suite $c_i = \{t_{i_1}, t_{i_2} \ldots\}$.

Édition de 2 arbres

L'édition d'arbres est définie de la même façon que l'édition sur les chaînes de caractères (*c.f.* chapitre précédent). On dispose de deux arbres T et T' ainsi que d'un ensemble d'opérations d'édition. À chacune de ces opérations, on associe un coût. La distance d'édition entre T et T' est alors le minimum des coûts des suites d'opérations qui, appliquées à T, produisent T'.

On définit les 3 opérations d'éditions suivantes sur l'arbre T :

- **la substitution** : Cette opération consiste à changer l'étiquette d'un nœud (figure 2.23).
- **la délétion** : La délétion, présentée dans la figure 2.24, consiste en la suppression d'un nœud t_i de l'arbre. Les fils de t_i sont rattachés au père de t_i, à la place de t_i, avec conservation de leur ordre.
- **L'insertion** : Cette opération est l'inverse de la délétion, une suite consécutive de fils d'un nœud est remplacée par un nouveau nœud. Ce nouveau nœud a alors pour fils la suite des nœuds précédemment détachée.

Notons tout d'abord la différence avec les séquences annotées de type *imbriqués*. Rappelons que les nœuds internes de l'arbre correspondent aux arcs de la séquence annotée ainsi qu'aux symboles aux extrémités de l'arc. Les feuilles quant à elles, correspondent aux symboles libres de la séquence (aucun arc n'a ce symbole pour extrémité). On peut alors re-écrire les trois opérations d'édition sur les arbres pour les séquences annotées :

- la substitution se traduit par **3** opérations :
  1. le changement des étiquettes des 2 extrémités d'un arc en même temps.
  2. le changement d'un symbole de la séquence s'il n'est pas lié.

3. si un arc relie deux lettres consécutives, on supprime cet arc et on remplace ces deux lettres par une seule lettre (substitution d'un nœud interne en une feuille).

– La délétion se traduit par 2 opérations :

4. la délétion d'un arc et des lettres à ses extrémités.
5. la délétion d'un symbole libre de la séquence.

– L'insertion se traduit par 2 opérations :

6. l'insertion dans la séquence de deux caractères ainsi que d'un arc les reliant.
7. l'insertion d'un unique caractère dans la séquence.

On remarque ainsi que l'édition d'arbre ordonné se rapproche de l'édition de séquence annotée présentée dans le chapitre précédent et introduite par Zhang *et al.* dans [99][66]. Cependant, ces deux éditions différent du fait que l'opération **3** n'avait pas été définie dans les opérations d'édition possibles sur les séquences annotées. Pour finir, notons que la représentation arborescente présente l'avantage de simplifier la définition des opérations d'édition, celle-ci s'effectuant ici à partir de trois opérations comme le montrent les deux figures 2.23 et 2.24 ci-dessous.

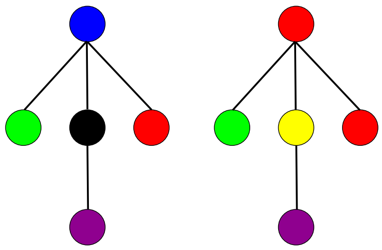


FIG. 2.23 – Re-étiquetage du nœud noir en jaune et de la racine bleue en rouge.

De même que pour l'édition de séquences, on notera $\sigma(a, b)$ l'opération de substitution de a par b. La délétion d'un nœud a sera alors notée $\sigma(a, \epsilon)$ et l'insertion d'un nœud b par $\sigma(\epsilon, b)$. On se muni d'une fonction de coût *score* qui à chaque opération $\sigma(a, b)$ affecte un score positif noté $score(a, b)$. De plus, on impose que *score* soit une distance, c'est à dire :

– $score(a, b) \geq 0$.

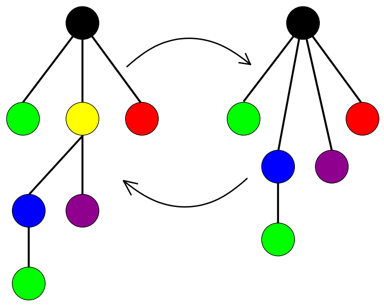


FIG. 2.24 – La délétion du nœud jaune dans l'arbre gauche ou l'insertion du nœud jaune dans l'arbre droit.

– $score(a, b) = 0$ si et seulement si $a = b$.
– $score(a, b) = score(b, a)$.
– $score(a, b) \leq score(a, c) + score(c, b)$.

Définition 18 (Distance d'édition pour les arbres enracinés ordonnés étiquetés). *Soient deux arbres enracinés ordonnés et étiquetés T et T'. Soit S l'ensemble des suites d'opérations d'édition qui, appliquées à T, produisent T' (on dit qu'une telle suite réalise l'édition de T en T'). Alors, la distance d'édition entre T et T', notée $D(T, T')$ est définie par :*

$$D(T, T') = Min\{score(s)|s \in S\}$$

On note que D est une distance car *score* est une distance.

Le premier algorithme permettant le calcul de cette distance d'édition fut proposé par Tai [83]. Plus tard, Zhang et Shasha dans [98] ont fourni un algorithme de programmation dynamique calculant cette distance et qui est plus performant que celui de Tai.

Dans un premier temps, nous présenterons cet algorithme, puis nous verrons en détail sa complexité. Enfin, nous montrerons comment il peut être optimisé.

De même que pour les séquences annotées par des arcs, nous allons introduire la notion d'association entre deux arbres.

Définition 19 (Association). *Soient deux arbres T et T' ordonnés, enracinés et étiquetés. Une association $A(T, T')$ entre T et T' est un ensemble de couples (i, j) représentant la mise en correspondance du nœud t_i avec le nœud t'_j. L'association A doit, de plus, satisfaire aux conditions suivantes :*

- $\forall (i,j) \in A$, $1 \leq i \leq |T|$ *et* $1 \leq j \leq |T'|$.
- $\forall (i_1, j_1), (i_2, j_2) \in A$
  - $i_1 = i_2$ *si et seulement si* $j_1 = j_2$.
  - $i_1 < i_2$ *si et seulement si* $j_1 < j_2$.
  - t_{i_1} *est ancêtre de* t_{i_2} *si et seulement si* t'_{j_1} *est ancêtre de* t'_{j_2}.

En d'autres termes, un nœud de T ne peut être mis en relation qu'avec un seul nœud de T' et l'ordre des fils doit être conservé par l'association, de même que la relation d'ancestralité.

On définit le score d'une association de la façon suivante :

$$\begin{aligned} score(A) = \sum_{(i,j)\in A} score(t_i, t'_j) \ &+ \sum_{1\leq i\leq |T|, (i,j)\notin A} score(t_i, \epsilon) \\ &+ \sum_{1\leq j\leq |T'|, (i,j)\notin A} score(\epsilon, t'_j) \end{aligned}$$

On voit ainsi que la notion d'association et celle d'édition sont fortement liées :

Proposition 12 ([98]). *Soit* S *une suite d'opérations* $s_1 \dots s_k$ *réalisant l'édition de* T *en* T'*, deux arbres enracinés ordonnés. Il existe une association* $A(T,T')$ *telle que* $score(A) \leq score(S)$*. De même, pour toute association* $A(T,T')$*, il existe une suite d'opérations* S *telle que* $score(A) = score(S)$*.*

Démonstration. La preuve de la première partie de cette proposition se fait par récurrence sur k. La relation est montrée pour le cas initial $k = 1$ correspondant à une unique opération d'édition. Supposons maintenant que l'on dispose d'une suite d'opérations $S = s_1 \dots s_{k-1}$ et d'une association A telle que $score(A) \leq score(S)$. Si l'on ajoute l'opération s_k à S pour construire la suite S', alors on construit l'association A' définie comme la composée de A avec l'association B correspondante à s_k, c'est-à-dire :

$$A' = A \circ B = \{(i,j) | \forall (i,j)\ \exists k \text{ tel que } (i,k) \in A \text{ et } (k,j) \in B\}$$

On montre [98] que $score(A') \leq score(A) + score(B)$. Il en résulte que $score(A') \leq score(A) + score(B) \leq score(S) + score(s_k) \leq score(S')$.

La deuxième partie de cette proposition se montre facilement : à partir de l'association A, on construit la suite d'opérations telle que tout nœud de T n'ayant pas de correspondant dans T' est détruit, tout nœud de T' n'ayant pas de correspondant dans T est inséré, et pour chaque couple (i,j) de A, on ajoute une substitution de t_i en t'_j. □

Ainsi, si l'on note E l'ensemble des associations possibles entre T et T', $D(T,T') = Min\{score(A)|A \in E\}$. La notion d'association est un formalisme intéressant. Nous verrons à la fin de cette section qu'il permet d'exprimer simplement des restrictions au problème d'édition.

L'algorithme de Zhang et Shasha utilise la notion de descendant le plus à gauche. Ainsi, on note $dpg(t_i)$ l'indice du descendant le plus à gauche du nœud t_i. La notation $t_i \ldots t_j$ représente la forêt composée des nœuds $t_i, t_{i+1} \ldots t_j$. L'arbre T s'écrit ainsi $t_1 \ldots t_{|T|}$. Afin de simplifier les formules qui suivent, nous noterons i le nœud t_i lorsque le contexte le permet. Remarquons que $t_i \ldots t_j = i \ldots j$ représente un arbre dans le cas où $i = dpg(j)$. L'algorithme repose sur les deux relations de récurrences suivantes :

$$\boxed{distance(i_1 \ldots i_2, j_1 \ldots j_2) =}$$

Si $((i_1 == dpg(i_2))\ et\ (j_1 == dpg(j_2)))$

$$MIN \begin{cases} distance(& i_1 \ldots i_2 - 1 & , j_1 \ldots j_2 &) + score(i_2, \epsilon) \\ distance(& i_1 \ldots i_2 & , j_1 \ldots j_2 - 1 &) + score(\epsilon, j_2) \\ distance(& i_1 \ldots i_2 - 1 & , j_1 \ldots j_2 - 1 &) + score(i_2, j_2) \end{cases} \tag{2.2}$$

Sinon

$$MIN \begin{cases} distance(& i_1 \ldots i_2 - 1 & , j_1 \ldots j_2) &) + score(i_2, \epsilon) \\ distance(& i_1 \ldots i_2) & , j_1 \ldots j_2 - 1 &) + score(\epsilon, j_2) \\ distance(& i_1 \ldots dpg(i_2) - 1 & , j_1 \ldots dpg(j_2) - 1 &) \\ \quad +distance(& dpg(i_2) \ldots i_2 & , dpg(j_2) \ldots j_2 &) \end{cases} \tag{2.3}$$

La figure 2.25 montre une représentation graphique de ces équations. Ainsi, on distingue deux parties : la première partie (2.2) calcule la distance entre deux arbres. En effet, celle-ci n'est appliquée que si les deux intervalles $i_1 \ldots i_2$ et $j_1 \ldots j_2$ décrivent des arbres ($i_1 = dpg(i_2)$ et $j_1 = dpg(j_2)$). Dans ce cas, la distance entre les deux arbres est le coût minimal entre :

- la délétion de la racine de l'arbre gauche,
- l'insertion de la racine de l'arbre droit,
- la substitution de la racine de l'arbre gauche par celle de l'arbre droit.

La deuxième partie (2.3) consiste en le calcul de la distance entre deux forêts. Cette distance est égale au minimum entre :

- la délétion de la racine de l'arbre droit de la forêt gauche,
- l'insertion de la racine de l'arbre droit de la forêt droite,
- la distance entre les deux arbres droits des deux forêts plus la distance entre les deux forêts restantes.

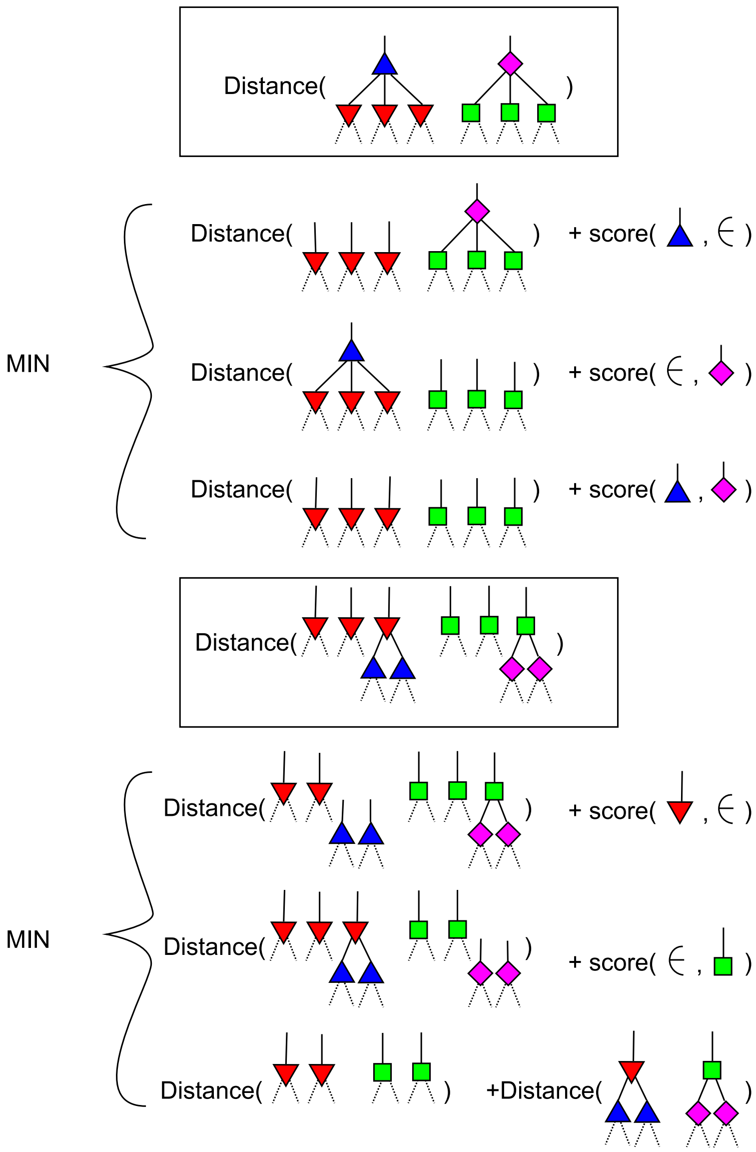


FIG. 2.25 – L'algorithme de calcul de la distance d'édition entre deux arbres de Zhang et Shasha : la première partie indique la distance entre deux arbres et la deuxième celle entre deux forêts.

Une implémentation naïve de cet algorithme mènerait à un temps d'exécution exponentiel en la taille des arbres en entrée dû au fait que grand nombre de calculs seraient faits plusieurs fois. Afin d'obtenir un temps d'exécution polynomial de cet algorithme nous devons alors conserver en mémoire un certain nombre de résultats intermédiaires.

En examinant la deuxième partie de la formule de récurrence, il apparaît qu'il nous faut mémoriser $distance(dpg(i) \dots i, dpg(j) \dots j)$ pour tous les i et j du fait de la dépendance impliquée par la formule. Ceci revient à mémoriser la distance entre tous les sous-arbres de T et ceux de T'. Cela implique aussi d'ordonner les calculs de telle façon que lorsque l'on calcule la distance entre deux sous-arbres i et j de T et T', les distances entre tous les sous-arbres de i et j soient déjà calculées. C'est pourquoi nous devons commencer par considérer les arbres au niveau des feuilles, puis remonter vers la racine.

De plus, pour chaque nœud i et j, la première partie de l'algorithme nous indique qu'il n'est pas nécessaire de calculer séparément la distance entre les sous-arbres i' et j' tels que i' est sur le chemin entre i et $dpg(i)$, et j' sur le chemin reliant j à $dpg(j)$. En effet, ces calculs seront effectués lors du calcul de la distance entre le sous-arbre i et le sous-arbre j.

Ainsi, pour un arbre T, on note RG l'ensemble des "racines gauches" défini

comme suit :

$$RG = \{k | 1 \leq k \leq |T| \text{ et } \nexists k' > k \text{ tq } dpg(k') = dpg(k)\}$$

Les deux observations ci-dessus, ainsi que l'ensemble des "racines gauches", sont illustrées dans la figure 2.26. Comme le montre la figure, l'ensemble des "racines gauches" représente l'ensemble des sous-arbres dont le calcul de la distance doit être fait de façon séparée. Il en découle l'algorithme suivant pour le calcul de la distance entre les deux arbres T et T'.

Comme nous l'avions dit précédemment, la distance entre chaque sous arbre de T et T' est conservée dans une matrice dont la taille est $|T| * |T'|$. L'algorithme calculant la distance d'édition, effectué selon les formules de récurrences 2.2 et 2.3, est présenté dans la figure 2.28. Ce calcul requiert une table supplémentaire afin de conserver la distance entre les forêts. Cette table est temporaire. En effet, les distances qui y sont conservées ne sont plus nécessaires une fois la distance entre les deux sous-arbres calculée. On se reportera à [98] pour la preuve de la validité de cet algorithme.

Dans [79], Zhang et Shapiro utilisent cet algorithme pour faire de la comparaison multiple de structures secondaires. Entre autres, ils utilisent des

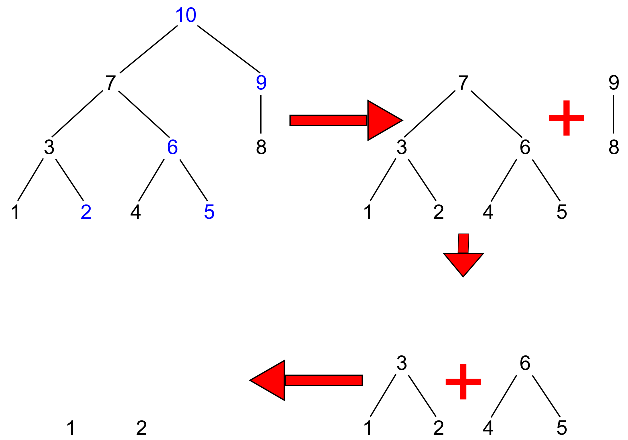


FIG. 2.26 – L'ensemble des "racines gauches" de l'arbre est $\{2, 5, 6, 9, 10\}$. Cette figure montre le découpage du calcul en sous-arbres induit par l'algorithme de Zhang et Shasha. On voit ainsi que le calcul de la distance entre l'arbre enraciné au nœud 10 et un autre arbre utilise le calcul des distances des sous-arbres enracinés aux nœuds $9, 6$ et 2 et induit le calcul de la distance des sous-arbres dont les racines sont sur le chemin entre le nœud 1 et le nœud 10 (1, 3 et 7).

Distance(T, T')

1. Calculer pour chaque nœud de T le *dpg*
2. Calculer pour chaque nœud de T' le *dpg*
3. Calculer l'ensemble RG des racines gauches de T
4. Calculer l'ensemble RG' des racines gauches de T'
5. pour chaque nœud i de RG dans l'ordre croissant
6.   pour chaque nœud j de RG' dans l'ordre croissant
7.   calculer $distance(dpg(i) \ldots i, dpg(j) \ldots j)$

FIG. 2.27 – Algorithme de Zhang et Shasha : ordonnancement des calculs de distance entre sous-arbres.

arbres dont les nœuds représentent les éléments structuraux. Leur approche consiste à calculer la distance d'édition entre toutes les structures deux à deux, puis à utiliser ces distances pour construire un arbre phylogénétique à l'aide de l'algorithme développé dans [43].

Complexité de l'algorithme de Zhang et Shasha

Le calcul de la complexité en temps de l'algorithme du calcul de la distance d'édition de Zhang et Shasha [98] consiste à déterminer pour chaque nœud de l'arbre, dans combien de calculs de distance d'arbres ce nœud intervient. Ceci revient à calculer pour chaque nœud i, le nombre de ces ancêtres qui sont des racines gauches. Cette valeur sera notée $\delta(i) = |anc(i) \bigcap RG(T)|$, où $anc(i)$ désigne l'ensemble des ancêtres de i. Nous pouvons alors définir $\delta(T)$ comme étant le maximum des $\delta(i)$ pour chaque nœud i de l'arbre.

On note $haut(i)$ la hauteur du nœud i ($haut(i) = |anc(i)|$), par extension $haut(T)$ sera la hauteur de l'arbre ($haut(T) = \max haut(i)$). Nous pouvons voir que $|RG(T)| \leq |feuilles(T)|$, avec $|feuilles(T)|$ le nombre de feuilles de l'arbre.

De par la définition de $\delta(i)$, on a alors $\delta(i) \leq \min(haut(i), |RG(T)|)$ et $\delta(T) \leq \min(haut(T), |RG(T)|)$.

Théorème 5 (Complexité). *La complexité en temps de l'algorithme de Zhang et Shasha pour le calcul de la distance d'édition entre deux arbres T et T' est $O(|T| * |T'| * \min(haut(T), |RG(T)|) * \min(haut(T'), |RG(T')|)$. Cet algorithme a une complexité en espace de $O(|T| * |T'|)$.*

Démonstration. La complexité en espace correspond à la place mémoire requise par les deux matrices $Arbre$ et $Foret$.

La complexité en temps de l'algorithme dépend du coût des comparaisons de sous-arbres enracinés aux racines gauches de chacun des arbres. Ainsi, si l'on note $T(i)$ le sous-arbre de T ayant pour racine le nœud i, la complexité de l'algorithme est

$$\sum_{i \in RG(T)} |T(i)| * \sum_{j \in RG(T')} |T'(j)|$$

Or l'expression $\sum_{i \in RG(T)} |T(i)|$ revient à compter, pour chacun des nœuds de l'arbre, dans combien de sous-arbres enracinés à une racine gauche il appartient. Ainsi $\sum_{i \in RG(T)} |T(i)| = \sum_{i=1}^{i=|T|} \delta(i)$.

Comme $\delta(i) \leq \delta(T) \leq min(haut(T), |RG(T)|)$, il en résulte la complexité annoncée. $\square$

Distance($dpg(i) \ldots i$,$dpg(j) \ldots j$)

1. $Arbre$: matrice des distances entre sous-arbres de taille $|T| * |T'|$
2. $Foret$: matrice de taille $i * j$
3. $Foret[0][0] = 0$
4. pour k allant de $dpg(i)$ a i
5. $Foret[k][0] = Foret[k-1][0] + score(t_k, \epsilon)$
6. pour l allant de $dpg(j)$ a j
7. $Foret[l][0] = Foret[l-1][0] + score(\epsilon, t_l)$
8. pour k allant de $dpg(i)$ a i
9.   pour l allant de $dpg(j)$ a j
10.   Si $dpg(k) = dpg(i)$ et $dpg(l) = dpg(j)$ alors
11. $Foret[k][l] = min\{$
12. $Foret[k-1][l] + score(t_k, \epsilon)$
13. $Foret[k][l-1] + score(\epsilon, t_l)$
14. $Foret[k-1][l-1] + score(t_k, t_l)\}$
15. $Arbre[k][l] = Foret[k][l]$
16.   Sinon
17. $Foret[k][l] = min\{$
18. $Foret[k-1][l] + score(t_k, \epsilon)$
19. $Foret[k][l-1] + score(\epsilon, t_l)$
20. $Foret[dpg(k)-1][dpg(l)-1] + Arbre[k][l]\}$
21.   Fin Si
22.   Fin Pour
23. Fin Pour
24. Destruction de la matrice $Foret$
25. Le resultat est dans Arbre[i][j]

FIG. 2.28 – Algorithme de Zhang et Shasha, calcul de la distance entre deux arbres. $Foret[k][l]$ représente la distance entre les forêts $dpg(i) \ldots k$ et $dpg(j) \ldots l$. On notera à la ligne 20 que le calcul suppose que $Arbre[k][l]$ est déjà calculé, ce qui est assuré par l'ordre des calculs présenté dans la figure 2.27.

Dans le pire des cas, la complexité en temps est $O(n^4)$, avec n la taille des deux arbres comparés. Ce cas apparaît lorsque les deux arbres sont des peignes.

Récemment, Dulucq et Tichit [21] ont montré la complexité en moyenne de cet algorithme.

Théorème 6 (Complexité en moyenne). *La complexité en moyenne de l'algorithme de Zhang et Shasha pour le calcul de la distance d'édition entre deux arbres T et T' est $O(|T|^{3/2}|T'|^{3/2})$.*

Il est intéressant de noter que, dans la deuxième partie de la formule de récurrence 2.3, l'algorithme divise toujours le problème entre l'arbre droit de la forêt et la forêt restante. Il n'est pas difficile de transformer l'algorithme afin que le problème soit divisé selon l'arbre gauche de la forêt plutôt que le droit. On utilisera alors les sous-arbres enracinés à gauche à la place de ceux enracinés à droite. Bien que cela ne change pas la complexité, on peut voir que la décomposition induite par ce choix ne mène pas forcément au même nombre de sous-arbres. Si l'on regarde l'arbre de la figure 2.29 et les décompositions qui lui sont associées, on voit que dans le cas d'une décomposition suivant les sous-arbres enracinés aux racines gauches, l'algorithme construit 12 sous forêts contre 8 lorsque l'on utilise les sous-arbres enracinés aux racines droites.

Dans [50], Klein propose une décomposition différente de celle utilisée par Zhang et Shasha et qui permet d'obtenir un algorithme en $O(n^3 \ln(n))$ dans le pire des cas pour l'édition de deux arbres de taille n. Pour cela, lors de l'appel récursif du calcul de la distance entre deux forêts, Klein choisit l'arbre dont le poids (nombre de nœuds) est le plus élevé.

Récemment, Dulucq et Touzet ont proposé dans [22] une analyse de ce problème de décomposition. Ils fournissent les critères permettant d'établir la décomposition optimale pour un arbre donné. Cependant, une question ouverte est de trouver un algorithme permettant de calculer cette décomposition optimale et dont le coût de calcul serait moins élevé que le gain généré par cette décomposition.

Autres problèmes liés à l'édition d'arbres.

Un certain nombre de "variantes" de la distance d'édition ont été proposées. Nous allons brièvement en faire la liste, puis nous verrons plus en détails le problème de l'alignement de deux arbres.

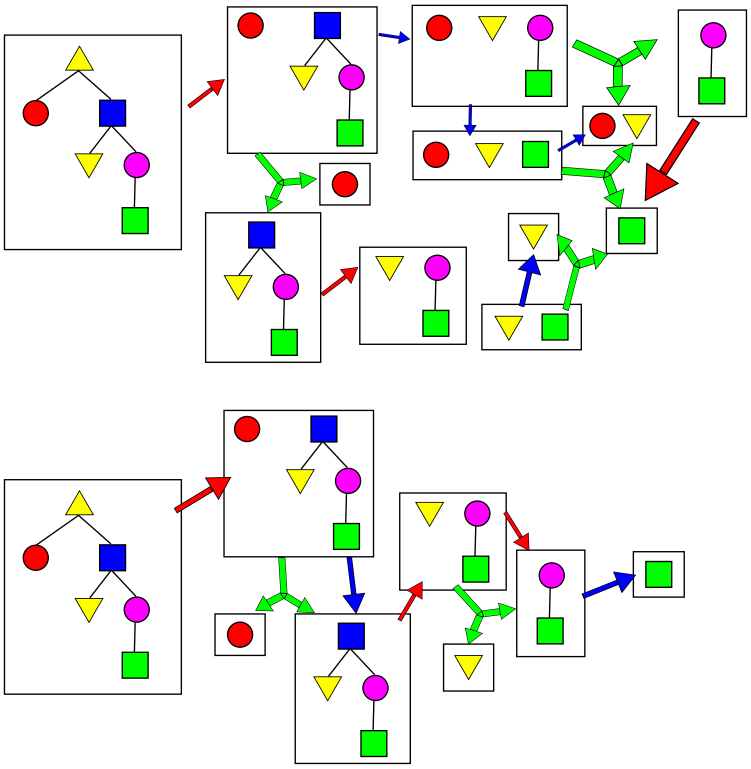


FIG. 2.29 – Décompositions produites par l'algorithme de Zhang et Shasha. Les flèches rouges indiquent une décomposition résultant de la partie 2.2 de la formule. Les flèches bleues indiquent la délétion de la racine de l'arbre gauche/droit de la forêt. Les flèches vertes indiquent la séparation du calcul d'une forêt en arbre gauche/droit et forêt restante. En haut, la décomposition suivant l'utilisation des sous-arbres enracinés aux racines gauches. En bas, la décomposition suivant l'utilisation des sous-arbres enracinés aux racines droites.

Les plus grandes sous-structures communes à deux arbres dont la distance d'édition est au plus k.

Wang *et al.* [92] posent un problème lié à l'édition d'arbre qui est intéressant dans le cadre de la comparaison des ARN.

Le problème repose sur la notion de sous-structure. Soit T un arbre enraciné ordonné, on définit une sous-structure de T comme étant un sous-arbre de T dont on a éventuellement supprimé un certain nombre de sous-arbres.

Soient alors deux arbres T et T' enracinés et ordonnés ainsi qu'un entier positif k, l'objectif est de trouver deux sous-structures S et S' de T et T' telles que :

1. la distance entre ces sous-structures soit inférieure à k,
2. Les sous-structures S et S' sont des sous-structures maximales dont la distance est inférieur à k.

Ce problème peut s'exprimer sous la forme de la recherche d'une association sous certaines contraintes. Pour cela, nous devons modifier la définition du coût d'une association. Soit $A(T, T')$ une association, on définit N et N' les ensembles de nœuds de T et T' qui n'interviennent pas dans A : $N = \{u|\ \nexists v \text{ tq } (u, v) \in A\}$ et $N' = \{v|\ \nexists u \text{ tq } (u, v) \in A\}$.

On note $anc(i)$ l'ensemble des ancêtres du nœud i et $desc(i)$ l'ensemble des descendants de i.

On construit le sous-ensemble I de N tel que pour chaque nœud n de I au moins un des descendants de n et un des ancêtres de n sont impliqués dans A :

$$\begin{aligned} I = \{n|n \in N; \exists x, y \in T' & \\ & \exists p \in anc(n) \text{ tq } (p, x) \in A \text{ et} \\ & \qquad \exists q \in desc(n) \text{ tq } (q, y) \in A\} \end{aligned}$$

De la même façon, on définit l'ensemble I'. On peut maintenant établir la nouvelle fonction de score suivante :

$$score_{ss}(A) = \sum_{(i,j)\in A} score(i, j) + \sum_{i\in I} score(i, \epsilon) + \sum_{j\in I'} score(\epsilon, j)$$

Le problème consiste alors à déterminer l'association de cardinalité maximale dont le $score_{ss}$ est inférieur à une valeur d.

Wang *et al.* fournissent un algorithme de programmation dynamique proche de l'algorithme de Zhang et Shasha permettant d'effectuer ce calcul en $O(|T| * |T'|)$.

Sous-arbres isomorphes, de taille maximale contenant les racines.

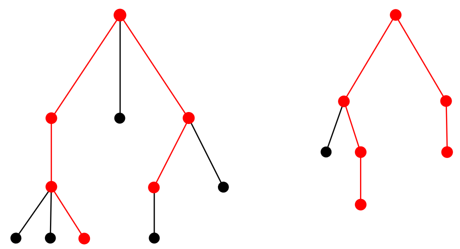


FIG. 2.30 – En rouge, le sous-arbre isomorphe, de taille maximale contenant les racines.

Ce problème proposé par Selkow [77] et résolu en $O(|T| * |T'|)$ par Yang [97] consiste à détruire les nœuds des deux arbres à partir des feuilles. Formellement, on cherche les plus grandes sous-structures S de T et S' de T' telles que :

1. Le sous-arbre de T induit par S est connexe (de même pour le sous-arbre de T' induit par S').
2. L'arbre défini par S a pour racine la racine de T (de même pour S' et T').
3. Il n'existe pas de sous-structures U et U' répondant aux deux critères précédents tels que $|U| + |U'| > |S| + |S'|$ et $D(U, U') < D(S, S')$.

Du point de vue de l'association, cela signifie que l'association A doit satisfaire à la condition suivante :

– Pour tous les couples de nœuds (i, j) de A tels que i et j ne soient pas les racines de T et T', le père de i est mis en relation avec le père de j dans A.

Bien que l'on cette contrainte soit très souvent verifiée lors de la comparaison d'ARN, il arrive parfois que ce ne soit pas le cas. Ainsi, en imposant cette contrainte systèmatiquement pour de la comparaison d'ARN, certains cas ne pourront être pris en compte.

Sous-structure maximale commune à deux arbres.

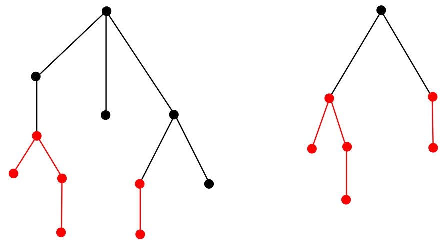


Fig. 2.31 – En rouge, la plus grande sous-structure commune depuis les feuilles.

Dans ce problème, on cherche la sous-structure de taille maximale en détruisant les nœuds depuis la racine vers les feuilles, c'est-à-dire qu'un nœud ne peut être détruit que si ces ancêtres l'ont été. Ainsi, pour chaque nœud i participant à une relation de l'association, l'ensemble des nœuds du sous-arbre enraciné à i doivent participer à l'association. Dans [87], Valiente fournit un algorithme en $O(|T| + |T'|)$ afin de calculer une telle distance. Cette distance ne semble pas non plus très utile dans le cadre de la comparaison d'ARN car elle ne repond pas à une contrainte biologique, c'est pourquoi nous n'en fournirons pas plus de détails.

Alignement de deux arbres

Comme nous l'avions dit précédemment, bien qu'il existe une relation d'équivalence claire entre l'édition et l'alignement de deux séquences, il n'en est pas de même pour les arbres. Le problème de l'alignement de deux arbres fut énoncé en 1994 par Jiang *et al.* dans [49].

Effectuer l'alignement de deux arbres consiste à insérer des nœuds dans chacun de ces arbres jusqu'à ce qu'ils deviennent isomorphes. L'insertion se fait de la même manière que celle définie pour l'édition d'arbre. La figure 2.32 montre l'alignement de deux arbres.

De même que pour la distance d'édition, on peut associer un coût à un alignement, celui-ci est la somme des scores en chacun des nœuds de l'alignement. La distance d'alignement est alors l'alignement dont le coût est minimal. On peut remarquer que l'alignement d'arbre et la distance d'édition

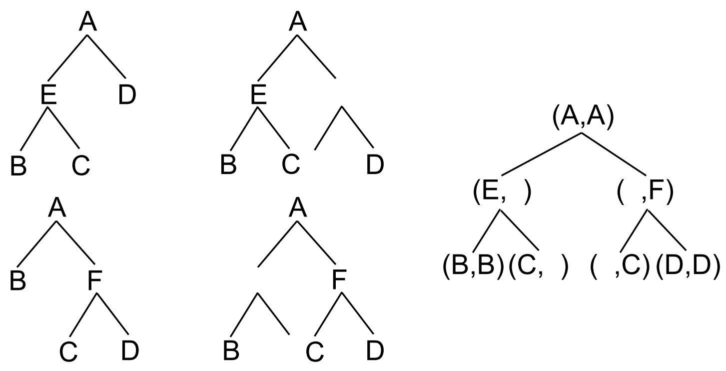


FIG. 2.32 – Exemple d'alignement entre deux arbres [49].

d'arbre ne sont pas équivalents. En effet, si l'on regarde la figure 2.32 et que l'on suppose que le coût d'une substitution est de 1 si les symboles différent, de 0 sinon, et le coût d'une délétion ou d'une insertion est 1, alors l'alignement présenté a un coût de 4 (c'est le coût de l'alignement des deux arbres). Lors d'un calcul d'édition, on peut supprimer le nœud E, puis insérer le nœud F et obtenir ainsi le deuxième arbre. La distance d'édition entre ces deux arbres est alors de 2.

En fait, l'alignement est une édition dans laquelle toutes les opérations d'insertion doivent être faites avant les opérations de délétion. Il en résulte que le coût d'un alignement est supérieur ou égal à la distance d'édition.

L'alignement de deux arbres enracinés ordonnés se calcule à l'aide d'un algorithme de programmation dynamique [49] dont la complexité en temps est $O(|T|*|T'|*(deg(T)*deg(T'))^2)$, $deg(T)$ étant le degré maximal des nœuds de T. Récemment, Jansson et Lingas ont proposé un algorithme permettant de calculer l'alignement en $O(n\log(n)deg^4d^2)$ où n est la taille des arbres en entrée, deg leur degré maximal et d le nombre maximal de nœuds insérés dans les deux arbres.

Höchsmann *et al.* dans [44] ont généralisé cette définition pour le calcul de l'alignement de deux forêts. En autres, ils proposent une nouvelle façon de coder les structures secondaires par des arbres ordonnés. Dans [45], ils utilisent leur modélisation pour effectuer l'alignement multiple de plusieurs ARN. Cet alignement multiple repose sur le calcul de l'alignement des structures deux à deux et leur permet d'établir des modèles pour un ensemble d'ARN.

Alignement multiple de séquences parenthésées.

Pour finir ce chapitre sur la modélisation des ARN par les arbres, nous allons brièvement voir un cas d'alignement multiple de structures d'ARN proposé par Shapiro en 1988 [78].

Shapiro propose de modéliser les ARN par des arbres de haut niveau, c'est-à-dire de construire des arbres dont les nœuds sont étiquetés sur $\{N, I, M, B, H\}$, chaque symbole correspondant à un élément de structure secondaire, I pour les boucles internes, M pour les boucles multiples, B pour les renflements, H pour les boucles terminales et N pour la racine de l'arbre.

Puis à partir de cet arbre, Shapiro construit une chaîne sur l'alphabet $\{N, I, M, B, H\} \bigcup \{(,)\}$. Cette chaîne est construite à l'aide d'un parcours en profondeur dans l'arbre, une parenthèse étant ouverte avant d'aller dans un sous-arbre puis fermée en revenant de ce même sous-arbre. Ainsi, l'arbre représenté dans la figure 2.33 est codé par la chaîne $(N(M(I(H))(B(H))))$. Cette séquence peut être simplifiée en ne disposant des parenthèses que lorsque le sous-arbre a plus d'un fils. Cela revient à ne pas mettre de parenthèses au niveau des tiges. On obtient alors la séquence $(NM(IH)(BH))$.

La figure 2.33 nous montre la représentation parenthésée d'un ARN. Cette représentation est un cas particulier de la représentation annotée par des arcs avec deux restrictions, à savoir : deux arcs ne peuvent avoir une extrémité en commun et ne peuvent se croiser. Ce sont donc des séquences annotées de type arcs *imbriqués*.

Puis Shapiro utilise un algorithme d'alignement multiple, GENALIGN [80], pour comparer un ensemble d'ARN. Cette approche n'est pas satisfaisante car lors de l'alignement multiple, la conservation des parenthèses (arcs) n'est pas maintenue. C'est pourquoi, en 1990 [79], Shapiro et Zhang proposent de modéliser les structures secondaires des ARN par des arbres et d'utiliser l'édition pour les comparer.

2.2.3 Conclusion

Nous venons de voir que les arbres étaient tout à fait adaptés à la modélisation des structures secondaires des ARN. Ils permettent une grande diversité quant aux possibilités de codage de ces structures : on peut aussi bien représenter les paires de bases et les bases non appariées que les tiges, boucles terminales et boucles multiples.

L'édition d'arbres enracinés, ordonnés et étiquetés est une approche per-

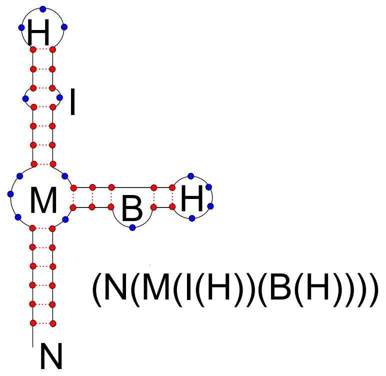


FIG. 2.33 – Un ARN et sa représentation parenthésée.

tinente pour la comparaison de ces arbres car elle permet de prendre en compte de manière réaliste des phénomènes biologiques connus comme la perte (délétion) de certaines bases, ou même d'éléments structuraux complets, ou encore le changement d'une base par une autre (mutation).

Notons pour finir qu'il ne fait aucun doute que le cas des arbres peut être appréhendé comme un cas particulier des séquences annotées de type *imbriqué*. Cependant, comme nous l'avons vu, ce sont des objets beaucoup plus simples à manipuler, que cela soit dans le cas de la distance édition ou même dans celui de la formulation d'associations particulières.

Chapitre 3

Modélisation et algorithmes pour la comparaison des ARN

Cette partie est consacrée aux travaux réalisés durant la thèse.

Dans le premier chapitre, nous exposerons certains résultats pratiques obtenus lors de la comparaison d'ARN en utilisant l'édition d'arbre. En outre, nous montrerons que l'édition d'arbre ne permet pas d'obtenir un résultat optimal pour la comparaison des ARN dans certaines situations. Nous avons ainsi constaté trois problèmes lors de l'utilisation de la distance d'édition d'arbres pour la comparaison des ARN.

Les deux premiers problèmes sont la conséquence du fait que l'association produite par l'édition d'arbres met en relation un nœud du premier arbre avec un nœud du deuxième arbre. Or, il est apparu que, dans le cas des ARN, certaines configurations demandent à pouvoir associer plusieurs nœuds du premier arbre à plusieurs nœuds, éventuellement en nombre différent, du deuxième arbre. Notre solution pour ce problème consiste en l'ajout de quatre nouvelles opérations d'édition : la fusion de nœuds et la fusion d'arcs, ainsi que leurs opérations inverses, la scission de nœuds et la scission d'arcs.

Le dernier problème se pose lorsque les ARN comparés, bien que proches globalement, diffèrent fortement sur une ou plusieurs régions. Alors qu'on souhaiterait que les deux régions soient tout simplement ignorées (détruites/insérées), l'association, résultante du calcul d'édition, met en correspondance de manière dispersée certains éléments de chacune de ces régions. Ce phénomène, que nous avons appelé *dispersion*, a déjà été observé dans le passé mais aucune solution satisfaisante n'a été donnée. Il apparaît qu'une solution possible consiste à intégrer dans le modèle des informations de plus haut niveau, telles que les hélices, lorsque l'on considère les bases de l'ARN ou bien les tiges pour les hélices.

Dans le deuxième chapitre, nous introduirons une modélisation originale des structures secondaires d'ARN. Celle-ci repose sur l'utilisation de plusieurs arbres liés entre eux par des relations d'abstraction. Dans un premier temps, nous décrirons ce modèle, appelé MiGaL. Puis nous montrerons comment l'utiliser pour modéliser la structure secondaire des ARN. Enfin, nous verrons comment faire de l'édition sur de telles structures. En outre, cette nouvelle modélisation intègre une solution à chacun des trois problèmes cités plus haut.

3.1 Deux nouvelles opérations d'édition

Dans un premier temps, nous allons donner des exemples concrets de comparaison de structures secondaires d'ARN en utilisant le calcul de la distance d'édition entre deux arbres défini dans la partie précédente. Après avoir relevé certains problèmes de cette distance, nous présentons quatre nouvelles opérations d'édition, ainsi que les modifications apportées à l'algorithme de Zhang et Shasha pour prendre en compte ces modifications. Puis, suite à une analyse de la complexité de ce nouvel algorithme, nous fournissons des exemples des résultats obtenus en pratique.

3.1.1 L'édition d'arbres pour les ARN : résultats pratiques

Actuellement, l'édition d'arbres présentée dans la partie précédente est l'approche la plus aboutie permettant d'effectuer la comparaison de deux structures secondaires d'ARN. Comme il l'a déjà été observé [98][79], l'un des avantages de l'édition d'arbres enracinés ordonnés est qu'elle peut être appliquée à toutes modélisations de la structure secondaire sous forme d'arbre enraciné ordonné. Voyons ainsi des résultats obtenus sur deux types d'arbres différents.

Codages utilisés

Pour nos exemples, nous avons utilisé deux façons différentes de représenter les structures secondaires avec des arbres. Bien que nous ayons déjà briévement décrit ces codages dans la partie précédente, nous redonnons ici une définition plus détaillée de ces arbres ainsi que de leurs utilisations pour nos exemples.

Le premier type d'arbre consiste à coder une paire de bases par un nœud interne de l'arbre et une base non appariée par une feuille de l'arbre. Dans la figure 3.1, on peut voir deux ARN extraits de la base de données des RNAse P [13]. Les arbres représentant ces deux structures sont présentés dans la figure 3.2. Les nœuds de ces arbres sont étiquetés par une base pour les feuilles et une paire de bases pour les nœuds.

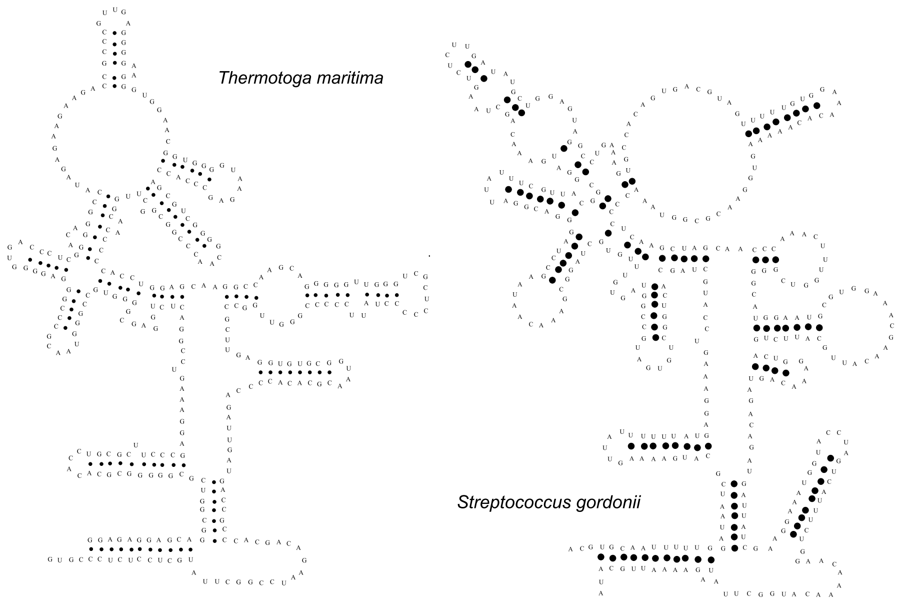


FIG. 3.1 – Deux ARN de type RNAse P. À gauche, la RNAse P de *Thermotoga maritima* et à droite celle de *Streptococcus gordonii.*

Le deuxième type d'arbre modélise les éléments structuraux des ARN. Ainsi, un nœud interne de l'arbre peut représenter une boucle multiple, un renflement ou une boucle interne. Les feuilles codent pour les boucles terminales et les arcs pour les hélices de la structure. La figure 3.3 représente deux ARN de type RNAse P extraits de la base [13]. Les modèles d'arbres correspondants sont illustrés dans la figure 3.4. Dans cette modélisation, des valeurs sont affectées aux nœuds et aux arcs. Celles-ci correspondent, respectivement, aux nombres de bases libres ou paires de bases formant l'élément

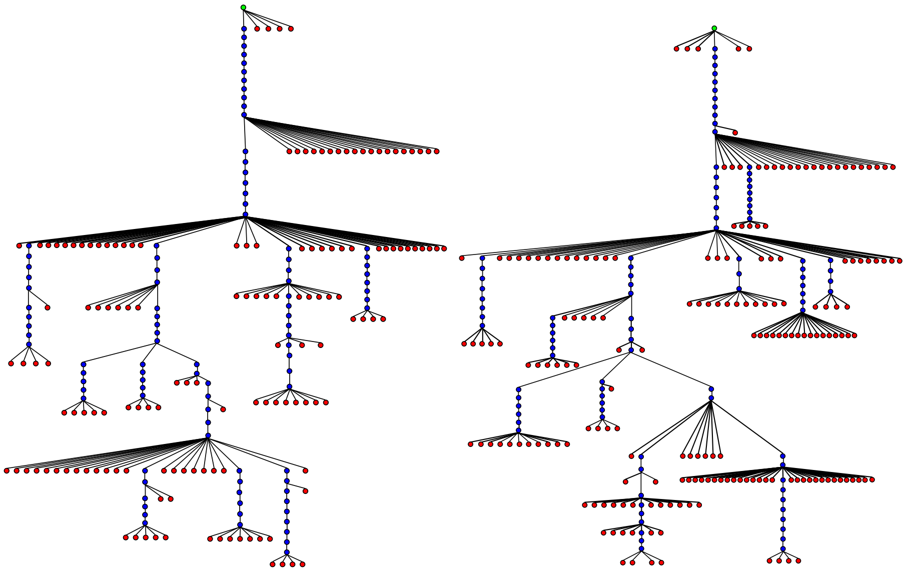


FIG. 3.2 – Les deux arbres modélisant les structures de la figure 3.1.

de structure secondaire.

On comprend vite qu'il existe ainsi une multitude de façons de représenter un ARN par un arbre, en fonction de la granularité voulue ou bien encore des étiquettes. Voyons maintenant le résultat de l'édition "classique" d'arbre sur nos deux exemples.

Calcul de la distance d'édition

La figure 3.5 illustre le résultat obtenu avec le calcul de la distance d'édition d'arbre entre les arbres de la figure 3.2. Les fonctions de score utilisées sont :

- substitution : 1 si différent, 0 sinon.
- insertion : 1.
- délétion : 1.

Pour comparer les deux arbres de la figure 3.4, nous avons utilisé la distance d'édition munie d'une fonction linéaire tenant compte des différentes valeurs des nœuds ainsi que de leurs types respectifs. Le résultat de cette édition est illustrée dans la figure 3.6.

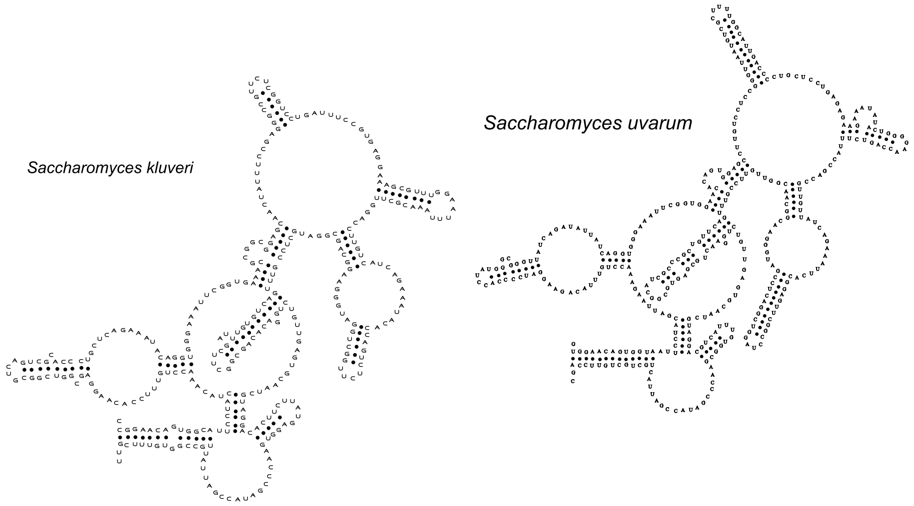


FIG. 3.3 – Deux ARN de type RNAse P. À gauche, la RNAse P de *Saccharomyces kluveri* et à droite celle de *Saccharomyces uvarum*.

Analyse des résultats : trois problèmes

Examinons tout d'abord notre premier exemple. La figure 3.7 nous montre le résultat de l'édition d'arbre sur les ARN.

Sur chacun de ces ARN nous avons entouré deux zones, l'une dans un carré et l'autre dans un cercle. Dans chacune de ces zones, les deux bases encadrées à gauche sont mises en relation avec les bases encadrées dans l'ARN de droite. Ces appariements montrent l'un des problèmes majeurs posé par l'édition en général. Le calcul de la distance repose sur une minimisation de coûts. Ainsi, même si deux zones des éléments comparés sont très différentes, il y aura très probablement des nœuds (bases) identiques dans chacune de ces zones. La définition même de la distance mène à associer ces éléments afin de diminuer le coût global plutôt que de détruire/insérer ces zones.

Ce problème, que nous avons appelé *problème de dispersion* n'est pas spécifique au cas des ARN. Ainsi, si l'on prend deux séquences aléatoires et que l'on effectue le calcul de la distance entre ces deux séquences, il est sûr que des lettres de la première séquence seront mises en correspondance avec des lettres de la deuxième séquence. Dans le cas de séquences aléatoires, ce

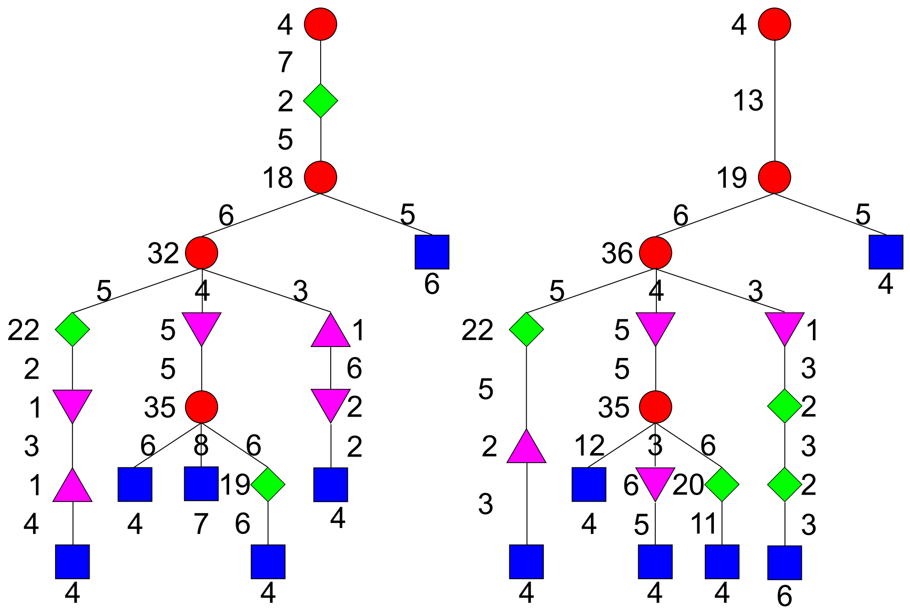


FIG. 3.4 – Représentation arborée des ARN de la figure 3.3. Les cercles codent pour les boucles multiples, les triangles pour les renflements, les losanges pour les boucles internes, les carrés pour les boucles terminales et les arcs pour les hélices. Les valeurs correspondent au nombre de base libres ou paires de bases formant l'élément de structure.

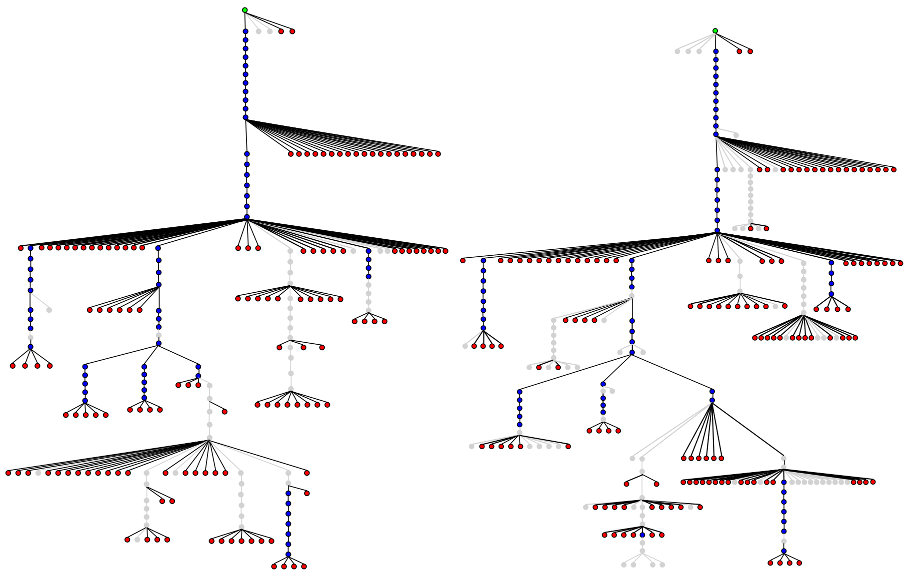


FIG. 3.5 – Résultat de l'édition des deux arbres de la figure 3.2.

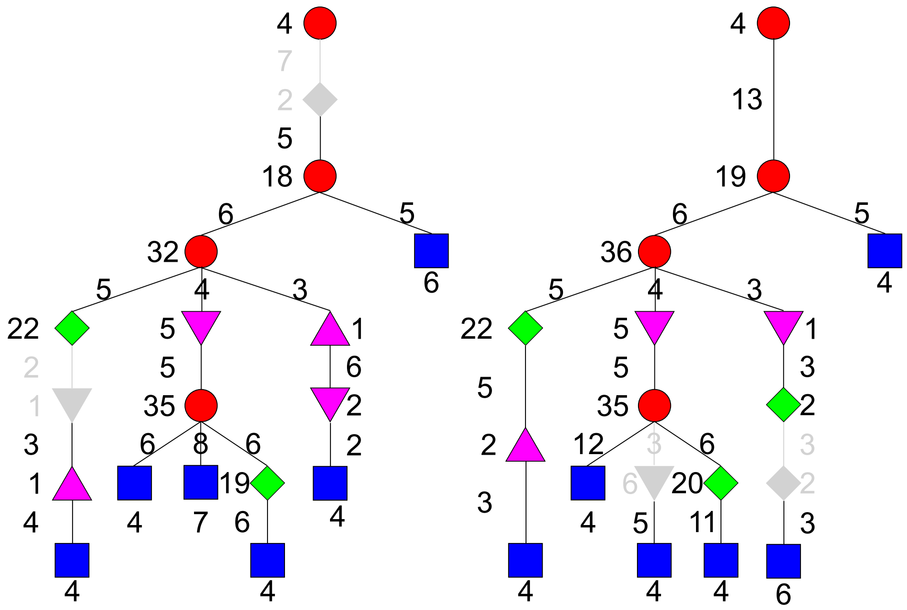


FIG. 3.6 – Résultat de l'édition des deux arbres de la figure 3.4.

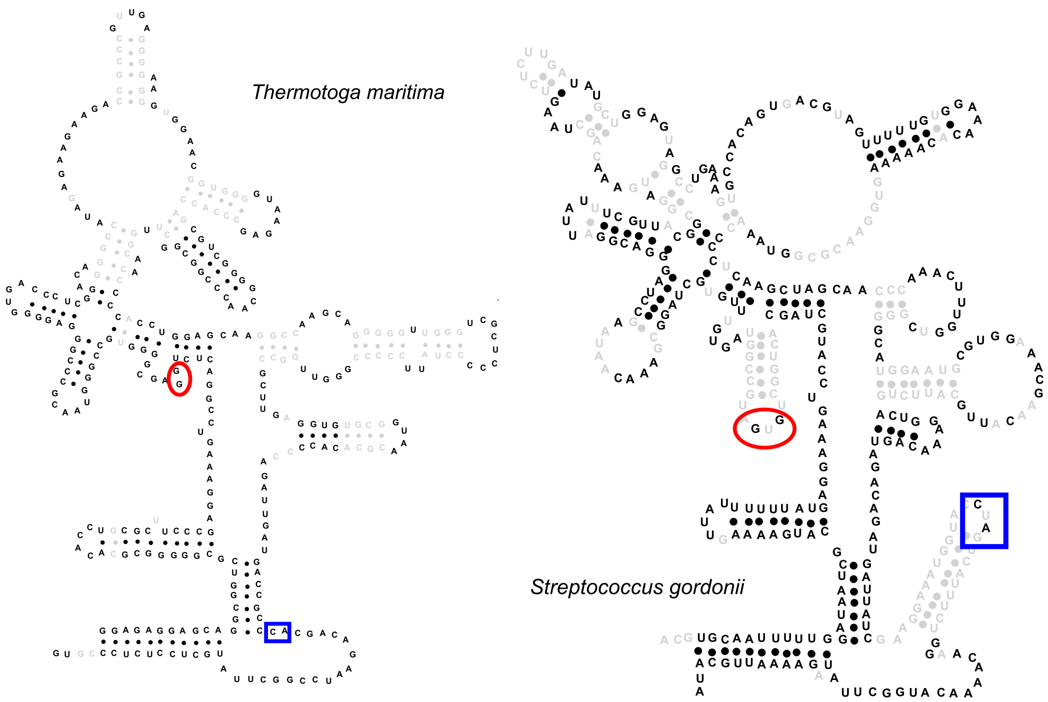


FIG. 3.7 – Résultat de l'édition des deux arbres de la figure 3.5 reporté sur les structures secondaires des ARN.

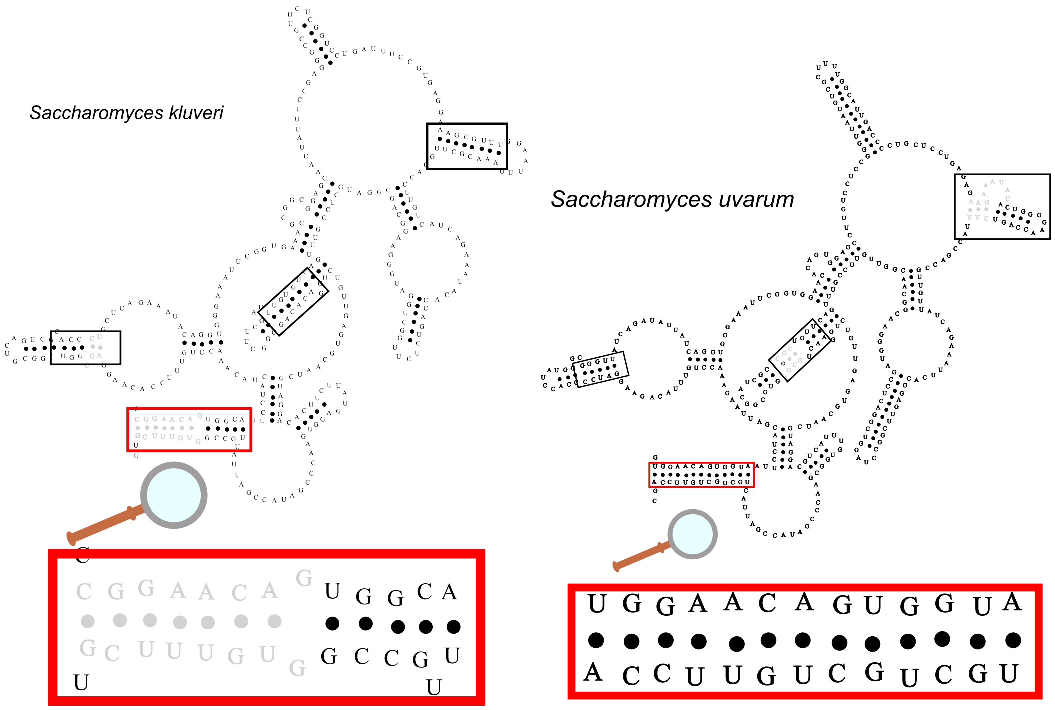


FIG. 3.8 – Résultat de l'édition des deux arbres de la figure 3.6 reporté sur les structures secondaires des ARN.

résultat n'a rien de surprenant, tandis que lors de la comparaison d'ARN, les structures possèdent des similarités qui *doivent* être mises en avant par l'édition d'arbre.

Cependant, les bases libres ne sont pas toutes équivalentes. En effet, elles font partie d'éléments structuraux tels que les boucles multiples, boucles terminales etc. Ainsi, on pourrait fortement réduire, voire éviter ce phénomène, si l'on tenait également compte d'une information de plus haut niveau sur la structure secondaire. Dans notre exemple, les deux zones encadrées mettent en relation des bases qui font partie d'une boucle terminale d'un coté, et d'une boucle multiple de l'autre. On voit bien qu'une information structurale complémentaire serait utile.

La figure 3.8 montre le résultat de l'édition de la figure 3.6 sur les deux ARN de l'illustration 3.3. Le problème qui se pose est signalé par les rectangles dans la figure. Si l'on regarde la boîte en bas : pour l'ARN de gauche nous avons une hélice composée de 7 paires de bases, une boucle interne, puis une autre hélice formée de 5 paires de bases. Pour l'ARN de droite, nous avons uniquement une hélice de 13 paires de bases. Du point de vue des arbres construits à partir de ces ARN, dans le premier cas nous avons un arc puis un nœud et un autre arc tandis que pour le deuxième cas, il y a uniquement un arc. Or l'édition, par définition, ne peut associer qu'un seul élément du premier arbre à un élément du deuxième arbre. Cependant, il est flagrant qu'ici, les deux hélices de l'ARN de gauche correspondent à l'hélice de l'ARN de droite. Les boîtes en haut de la figure illustrent un cas similaire.

Ainsi, l'impossibilité dans certains cas de pouvoir associer plusieurs hélices (arcs) du premier ARN (arbre) à une ou plusieurs hélices (arcs) du deuxième ARN (arbre) est réellement handicapant pour la comparaison des ARN. On peut trouver d'autres types d'arbres modélisant des ARN pour lequel ce problème apparaît. Cela est le cas, par exemple, lorsque les nœuds internes de l'arbre codent pour les boucles multiples, les arcs pour les tiges et les feuilles pour les boucles terminales.

Le troisième problème est similaire au précédent mais porte sur les nœuds. Ce problème est illustré dans la figure 3.9. Le type d'arbre utilisé est le même que celui de la figure 3.4. Dans cette figure, nous avons dessiné les arbres correspondant aux zones entourées des deux ARN. On voit bien que la meilleure association consiste à mettre en correspondance les deux boucles internes (losanges verts) de l'ARN de gauche avec la boucle interne de l'ARN de droite. De plus, on voit aussi un exemple du problème précédent provoqué par la présence dans l'ARN de gauche d'un petit renflement au début de la zone

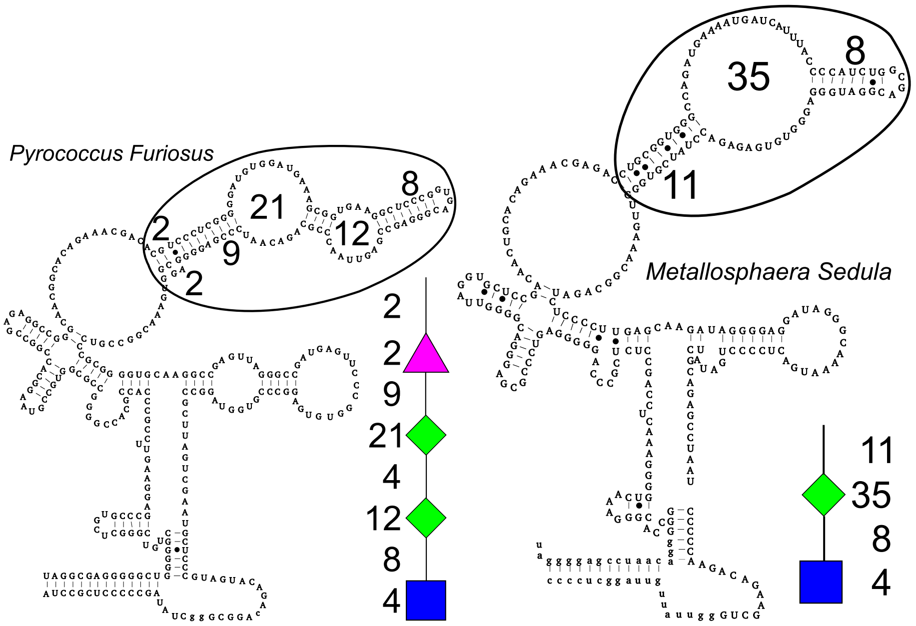


FIG. 3.9 – Édition de deux ARN de type RNAse P extraits de la base [13]. Les arbres utilisés pour modéliser ces ARN codent les éléments de structure secondaire, c'est-à-dire que les nœuds internes codent pour les boucles multiples, boucles internes et renflements, les feuilles pour les boucles terminales et les arcs pour les hélices. Pour chaque ARN, l'arbre en bas à droite représente la partie entourée de l'ARN.

entourée. Une fois encore, l'édition ne peut pas mettre en correspondance un groupe d'éléments d'un arbre avec un groupe d'éléments d'un autre arbre.

Pour fournir une solution aux deux derniers problèmes, nous avons introduit quatre nouvelles opérations d'édition : la fusion de nœuds et la fusion d'arcs ainsi que leurs opérations inverses, la scission de nœuds et la scission d'arcs. Le premier problème, celui de la dispersion, sera quant à lui traité dans le prochain chapitre à l'aide d'une nouvelle modélisation.

3.1.2 Deux nouvelles opérations d'édition

La fusion de nœuds et la fusion d'arcs ont pour objectif de répondre aux problèmes soulevés par la comparaison de structures secondaires d'ARN utilisant des arbres dont les nœuds et arcs représentent des éléments de cette structure.

Si u et v sont deux nœuds tels que v est le fils de u et que a désigne l'arc reliant u et v, alors u est appelé l'*origine* de l'arc a et v sa *destination*. On dit aussi que a *part* de u et *pointe* sur v.

Il est important de noter que nous faisons ici référence à des arbres dont les nœuds et les arcs sont étiquetés. Cependant, un arc et le nœud sur lequel l'arc pointe ne forment qu'un seul objet au sens des opérations d'édition. Ainsi, par exemple, la délétion d'un nœud implique la délétion aussi de l'arc pointant sur ce nœud. De même, la substitution d'un nœud par un autre implique la substitution de l'arc pointant sur le premier nœud par l'arc pointant sur le deuxième nœud.

Définition des nouvelles opérations

La fusion de nœuds est illustrée dans la figure 3.10. Soit T un arbre enraciné, ordonné et étiqueté sur Σ, u un nœud de cet arbre. La suite $\{f_1, \ldots, f_k\}$ dénote les fils de u. La fusion de nœuds de u avec f_i consiste à supprimer f_i de l'arbre. Les fils $\{f'_1, \ldots, f'_l\}$ de f_i sont alors rattachés à u à la place de f_i, u a alors pour fils $\{f_1, \ldots, f_{i-1}, f'_1, \ldots, f'_l, f_{i+1}, \ldots, f_k\}$. Enfin, l'étiquette du nœud u est modifiée selon une fonction $renomme_{nœud} : \Sigma \times \Sigma \to \Sigma$ qui fournit l'étiquette correspondante à la fusion de l'étiquette de u et de f_i.

Au premier abord, cette opération semble équivalente à une délétion suivie d'une substitution. Nous reviendrons sur ce point un peu plus tard lors d'une discussion sur les fonctions de coût.

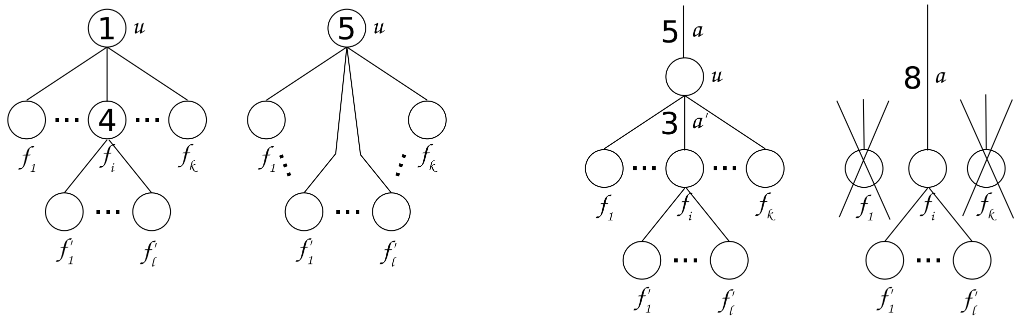


FIG. 3.10 – Exemple de fusion de nœuds (à gauche) et de fusion d'arcs (à droite).

La scission de nœuds est définie comme étant l'opération inverse de la fusion de nœuds (de même que l'insertion est l'opération inverse de la délétion). Celle-ci est très proche de l'opération d'insertion : une suite consécutive s de fils d'un noeud u est remplacée par un nouveau noeud f auquel on attribut comme quite de fils s. Les étiquettes de f et u sont modifiées afin qu'elles représentent l'étiquette d'origine de u.

La fusion d'arcs consiste à regrouper deux arcs. Celle-ci est représentée dans la figure 3.10. Soit a l'arc arrivant au nœud u et a' un arc partant du nœud u pour arriver à l'un de ses fils f_i. La fusion de l'arcs a avec l'arc a' consiste à remplacer la destination de a par f_i. L'étiquette de a est modifiée selon une fonction $renomme_{arc} : \Sigma \times \Sigma \rightarrow \Sigma$ qui calcule une nouvelle étiquette à partir des étiquettes de a et a'. On remarquera que tous les sous-arbres attachés aux nœuds f_j pour $j \neq i$ sont détruits dans cette opération. En effet l'opération de fusion d'arcs sert, entre autre, à regrouper deux hélices (ou tiges) séparées par une boucle interne (pas de sous-arbres) ou éventuellement une boucle multiple (sous-arbres). Une alternative aurait pu être de rattacher ces sous-arbres au père de u ou bien au noeud f_i ce qui reviendrait à déplacer une partie de la structure secondaire (sous-arbre), la délétion de ces sous-arbres semble être le choix le plus pertinent.

La scission d'arcs est l'opération inverse de la fusion d'arcs, elle correspond à l'éclatement d'un arc en deux arcs tels que leurs étiquettes représentent celle de l'arc d'origine.

Nouvelle distance et fonction de coût

Le choix des fonctions de coût associées à ces nouvelles opérations est très important.

Comme nous l'avons mentionné précédemment, nous utilisons des arbres dont les nœuds et les arcs sont étiquetés, mais un arc et le nœud sur lequel il pointe sont solidaires vis-à-vis des opérations d'édition. Soient deux nœuds u et v, le coût de la substitution du nœud u par le nœud v est noté $sub(u, v)$. Il consiste en la substitution de l'étiquette du nœud u par celle de v ainsi que la substitution de l'étiquette de l'arc pointant sur u par l'étiquette de l'arc pointant sur v. Lorsque nous voudrons faire référence séparément à chacune de ces opérations, nous écrirons sub_a et sub_n : $sub(u, v) = sub_a(u, v)+sub_n(u, v)$. Il en sera de même pour la délétion dont le coût est noté *del* et composée de del_a et del_n, et l'insertion *ins* composée de ins_a et ins_n.

Voyons en premier lieu les conditions nécessaires pour que la distance d'édition augmentée de ces quatre nouvelles opérations soit toujours une distance.

Proposition 13. *Soit $D(T, T')$ la distance d'édition entre deux arbres T,T' enracinés, ordonnés et étiquetés utilisant les sept opérations suivantes :*

- *La substitution des labels (arcs et nœuds) d'un nœud u par ceux du nœud v. Son coût est $sub(u, v)$.*
- *La délétion d'un nœud u de T dont le coût est $del(u)$.*
- *L'insertion d'un nœud u' de T' dont le coût est $ins(u')$.*
- *La fusion de deux nœuds u et v de T. Cette fusion produit un nouveau nœud r dans T tel que l'étiquette de ce nœud est $renomme_{nœud}(u, v)$ et l'étiquette de l'arc pointant sur r est identique à celle de l'arc pointant sur u. Son coût est noté $fusion_{nœuds}(u, v, r)$.*
- *La scission du nœuds r' de T' en deux nœuds u' et v'. Son coût est noté $scission_{nœuds}(u', v', r')$.*
- *La fusion des deux arcs a et b de T tels que a pointe sur le nœud u et b sur le nœud v. Le résultat de cette fusion est le nœud r dont l'étiquette est identique à celle de v et l'étiquette de l'arc pointant sur r est $renomme_{arc}(a, b)$. Le coût de cette opération est noté $fusion_{arcs}(u, v, r)$.*
- *La scission de l'arcs pointant sur le nœud r de T' en deux arcs a' et b' pointant sur les nœuds u' et v'. Son coût est $scission_{arcs}(u', v', r')$.*

D est une distance si et seulement si :

- *sub est une distance.*

– $del(u) = ins(u) > 0$.
– $fusion_{nœuds}(u, v, r) = scission_{nœuds}(u, v, r) > 0$.
– $fusion_{arcs}(u, v, r) = scission_{arcs}(u, v, r) > 0$.

Démonstration. On note $\emptyset$ l'arbre vide.

L'implication ($\Rightarrow$) se démontre comme suit : soient les deux nœuds u et v, on a $D(u, v) = sub(u, v)$. Ainsi, sub est une distance.

D est une distance, on a donc $del(u) = D(u, \emptyset) = D(\emptyset, u) = ins(u)$ et donc $del(u) = ins(u) > 0$.

Soient deux arbres T et T', et deux nœuds u et v de T tels que la fusion de u et v transforme T en T'. Alors, $fusion_{nœuds}(u, v, u') = D(T, T') = D(T', T) = scission_{nœuds}(u, v, u') \geq 0$. En raisonnant de même pour la fusion d'arcs, il en résulte les conditions énoncées.

Pour montrer la réciproque, nous devons prouver les 4 propriétés que D doit vérifier pour être une distance, c'est-à-dire :

1. **Positivité :** D est positive car c'est la somme de coûts eux-mêmes positifs.
2. **Séparation :** Si $D(u, u') = 0$ alors $u = u'$ car seul le coût de la substitution peut être nul et sub est une distance. De même si $u = u'$ alors $D(u, u') = 0$.
3. **Symétrie :** Montrons que $D(T, T') = D(T', T)$. Si $D(T, T')$ est le score de la suite $\{s_1, \ldots, s_l\}$, on peut construire la suite $D' = \{s'_l, \ldots, s'_1\}$ telle que pour tout $i \in [1; n]$
   – si $s_i = sub(n, m)$ alors $s'_i = sub(m, n)$
   – si $s_i = del(n)$ alors $s'_i = ins(n)$
   – si $s_i = ins(n)$ alors $s'_i = del(n)$
   – si $s_i = fusion_{nœuds}(n, n', r)$ alors $s'_i = scission_{nœuds}(n, n', r)$.
   – si $s_i = fusion_{arcs}(n, n', r)$ alors $s'_i = scission_{arcs}(n, n', r)$.
   – si $s_i = scission_{nœuds}(n, n', r)$ alors $s'_i = fusion_{nœuds}(n, n', r)$.
   – si $s_i = scission_{arcs}(n, n', r)$ alors $s'_i = fusion_{arcs}(n, n', r)$.
   Les fonctions d'édition symétriques sont égales, il en résulte que la suite $\{s'_1, \ldots, s'_l\}$ réalisant l'édition de T' en T est de score minimal et donc que $D(T, T') = D(T', T)$.
4. **Inégalité triangulaire :** Supposons qu'il existe un arbre U tel que $D(T, T') > D(T, U) + D(U, T')$. Alors si l'on concatène la suite des opérations d'édition de $D(T, U)$ et $D(U, T')$, on obtient une suite d'opérations réalisant l'édition de T en T' et dont le coût est inférieur à $D(T, T')$ ce qui contredit la définition de la distance d'édition.

□

Ces conditions sont le minimum requis pour que l'édition soit toujours une distance. Nous avons ajouté certaines autres conditions sur les fonctions de renommage ($renomme_{arc}$ et $renomme_{nœud}$) ainsi que sur les fonctions de coût.

La fonction de renommage et la délétion

Les deux fonctions de renommage ($renomme_{nœud}$ et $renomme_{arc}$) calculent l'étiquette correspondante à la fusion de deux autres étiquettes. Il semble raisonnable d'imposer que :

$$\forall a, b, c \in \Sigma \text{ tels que } renomme_{arc|nœud}(a, b) = c, del(a) + del(b) \leq del(c)$$

En effet, si cette condition n'est pas remplie, la distance d'édition aura tendance à systématiquement fusionner les éléments pour les détruire ensuite car le coût de la délétion des éléments fusionnés sera plus faible que la délétion des éléments séparés.

Délétion et fusion

Comme nous l'avons vu, la fusion de nœuds peut-être "simulée" avec une délétion suivie d'une substitution. Ceci est illustré dans la figure 3.11. Si l'on suit les opérations du chemin bleu, alors nous avons la fusion du nœud 5 avec le nœud 3, puis la substitution du nœud 8 par le nœud 9. Par le chemin rouge, nous avons la substitution du nœud 5 par le nœud 9 puis la délétion du nœud 3.

Le score du chemin bleu est $fusion_{nœuds}(5, 3, 8) + sub(8, 9)$ et celui du chemin rouge $sub(5, 9) + del(3)$. Nous voyons ici que le choix de la fonction $fusion_{nœuds}$ est délicate car c'est elle qui "décidera" si ce sera plutôt une fusion ou une délétion qui devra être effectuée. Dans le cas exposé, cela revient à regarder si le nœud 8 créé par la fusion a un score de substitution avec le nœud 9 meilleur que celui du nœud 5 avec le nœud 9. C'est pourquoi nous avons choisi d'utiliser la fonction suivante pour la fusion :

$$fusion_{nœuds}(u, v, r) = del(v) + t = del_a(v) + del_n(v) + t$$

La variable t est un paramètre extérieur que nous avons appelé *paramètre de fusion*. Dans l'exemple, on obtient alors que $fusion_{nœuds}(5, 3, 8) + sub(8, 9)$

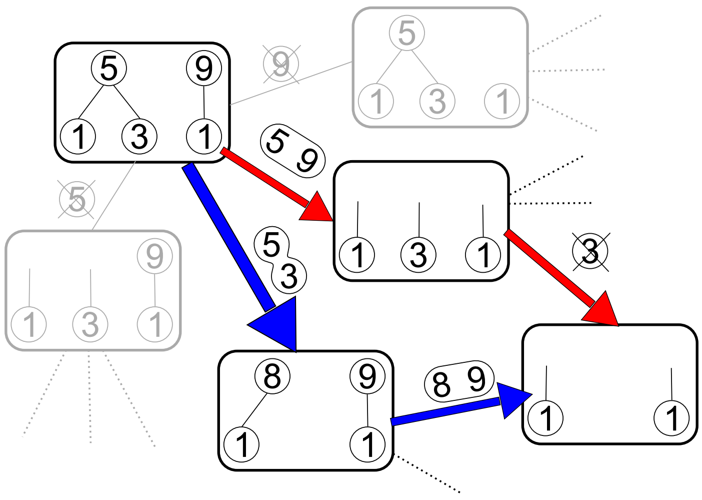


FIG. 3.11 – Illustration de la fusion de nœuds (en bleu) vis-à-vis d'une délétion suivie d'une substitution (en rouge).

équivaut à $del(3) + t$. Si l'on compare cette nouvelle valeur avec le chemin rouge dont le coût est $sub(5, 9) + del(3)$, on voit que la fusion est choisie si le coût de substitution en utilisant la nouvelle étiquette améliore le score de substitution avec l'ancienne étiquette d'au moins t.

Pour la fusion d'arcs, le score doit prendre en compte le coût de la délétion des sous-arbres détruits par cette opération. Ainsi, si l'on nomme $del_{st}(u)$ le coût de la délétion du sous-arbre enraciné en u (somme des délétions de l'ensemble de nœuds et arcs du sous-arbre)

$$fusion_{arcs}(u, v, r) = \sum_{s \text{ frère de } v} del_{st}(s) + del(u) + t'$$

Le raisonnement sur le paramètre de fusion est le même que pour la fusion de nœuds.

Une des conséquences de cette fonction de fusion est qu'une fusion d'arcs ne peut suivre une fusion de nœuds. En effet, comme le montre la figure 3.12, le long du chemin rouge, nous faisons une fusion de nœuds dont le coût est $del_a(b) + del_n(B) + t$ suivie d'une fusion d'arcs dont le coût est $del_a(a)+del_n(AB)+t$. Le chemin bleu consiste en une délétion dont le coût est $del_a(b)+del_n(B)$ suivie d'une fusion de coût $del_a(a)+del_n(A)+t$. La différence

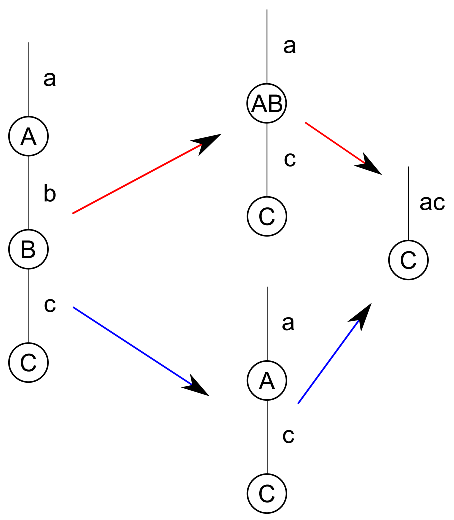


Fig. 3.12 – Fusion de nœuds suivie d'une fusion d'arcs.

de coût entre le chemin rouge et le chemin bleu est $del_n(AB) + t - del_n(A)$. D'après la condition que nous avons posée sur la fonction de renommage au-dessus, cette valeur est toujours positive. Une fusion d'arcs ne peut donc pas suivre une fusion de nœuds. Ainsi, l'utilisation de ces fonctions de coûts nous permet ainsi de réduire le nombre de fusions successives possibles par noeud comme nous le verrons lors du calcul de la complexité de l'algorithme.

Nous allons maintenant voir l'algorithme permettant le calcul de la distance d'édition munie de ces 7 opérations. Cet algorithme est indépendant des fonctions de score choisies.

3.1.3 Algorithme

Nous avons adapté l'algorithme de Zhang et Shasha présenté dans le chapitre 4 afin de prendre en compte nos nouvelles opérations. La figure 3.13 montre de manière graphique cet algorithme. Comme on peut le voir, nous avons ajouté lors du calcul de la distance entre deux arbres la possibilité pour la racine d'effectuer une fusion (ou scission) d'arcs ou de nœuds avec un de ces fils.

Voici maintenant une version formelle des formules de récurrence de notre algorithme. Soit T un arbre enraciné, ordonné et étiqueté composé de $|T|$ nœuds. Les nœuds de T sont numérotés selon un parcours postfixe, t_i désigne alors le $i^{\text{ème}}$ nœud de T. Lorsque le contexte sera clair, on dénotera le nœud t_i par i tout simplement. On note $dpg(i)$ l'indice du descendant le plus à gauche du nœud i. Enfin, $T(i \dots j)$ désigne la forêt composée des nœuds $t_i, \dots, t_j$ que l'on notera aussi $i \dots j$ dans les formules ci-dessous. Rappelons tout d'abord la formule utilisée par Zhang et Shasha pour le calcul de la distance d'édition :

$$\boxed{distance(i_1 \dots i_2, j_1 \dots j_2) =}$$

Si $((i_1 == dpg(i_2))\ et\ (j_1 == dpg(j_2)))$

$$MIN \begin{cases} distance(& i_1 \dots i_2 - 1 & , j_1 \dots j_2 &) + del(i_2) \\ distance(& i_1 \dots i_2 & , j_1 \dots j_2 - 1 &) + ins(j_2) \\ distance(& i_1 \dots i_2 - 1 & , j_1 \dots j_2 - 1 &) + sub(i_2, j_2) \end{cases} \tag{3.1}$$

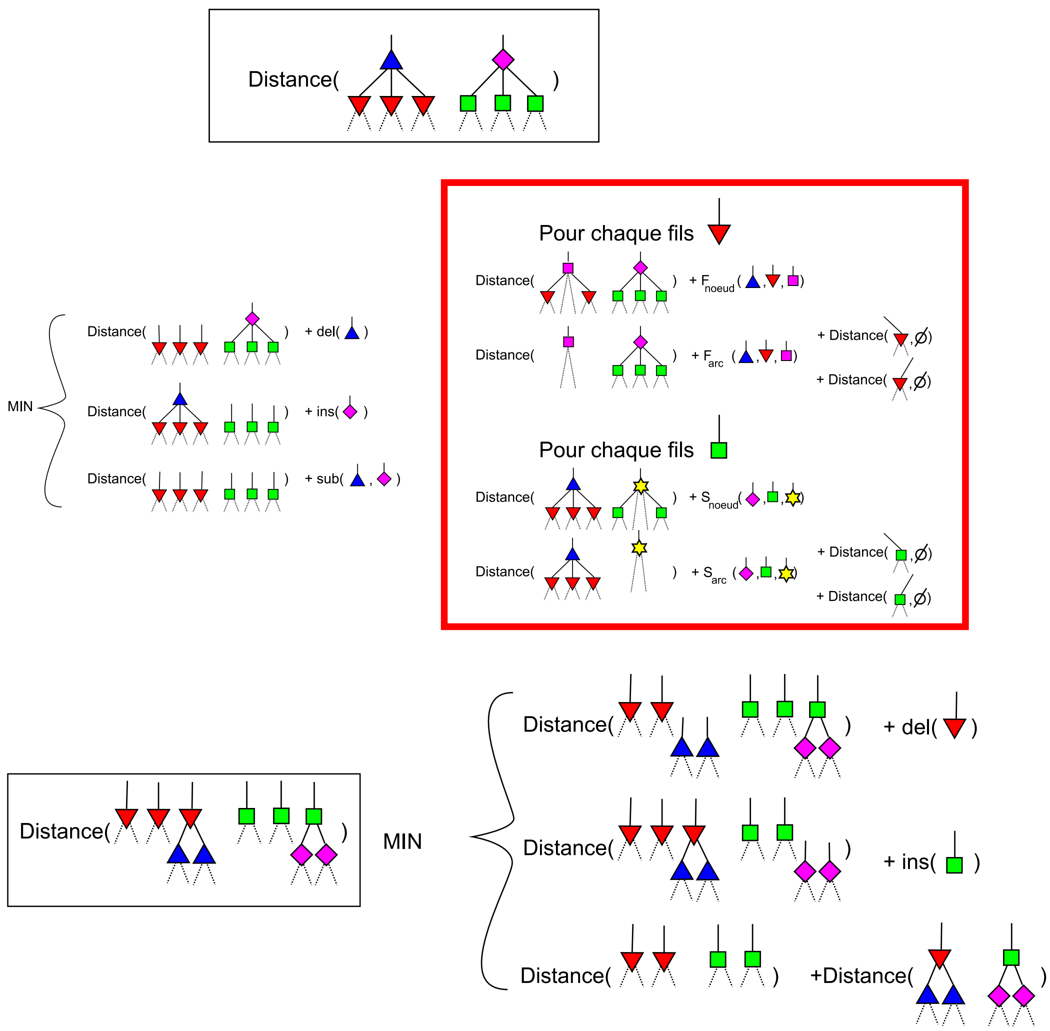


FIG. 3.13 – L'algorithme d'édition d'arbre prenant en compte les opérations de fusion ; les opérations ajoutées à l'algorithme de Zhang et Shasha sont dans le cadre rouge.

Sinon

$$MIN\left\{\begin{array}{llll} distance(& i_1 \dots i_2 - 1 & , j_1 \dots j_2) &) + del(i_2) \\ distance(& i_1 \dots i_2) & , j_1 \dots j_2 - 1 &) + ins(j_2) \\ distance(& i_1 \dots dpg(i_2) - 1 & , j_1 \dots dpg(j_2) - 1 &) \\ \quad +distance(& dpg(i_2) \dots i_2 & , dpg(j_2) \dots j_2 &) \end{array}\right. \tag{3.2}$$

La formule est décomposée en deux parties, la première (3.1) calcule la distance d'édition entre deux arbres et la deuxième (3.2) la distance d'édition entre deux forêts. Voici maintenant les formules permettant de calculer la distance d'édition en prenant en compte les fusions et scissions.

$$\boxed{distance((\{i_1, \dots, i_k\}, liste), (\{j_1, \dots, j_{k'}\}, liste')) =}$$

Si $((i_1 \geq dpg(i_k))$ et $(j_1 \geq dpg(j_{k'})))$

$$MIN\left\{\begin{array}{l} \begin{array}{llllll} distance(& (\{i_1 \dots i_{k-1}\} & , \emptyset) & , (\{j_1 \dots j_{k'}\} & , liste') &) + del(i_k) \\ distance(& (\{i_1 \dots i_k\} & , liste) & , (\{j_1 \dots j_{k'-1}\} & , \emptyset) &) + ins(j_{k'}) \\ distance(& (\{i_1 \dots i_{k-1}\} & , \emptyset) & , (\{j_1 \dots j_{k'-1}\} & , \emptyset) &) + sub(i_k, j_{k'}) \end{array} \\ pour\ chaque\ fils\ i_c\ de\ i_k\ dans\ \{i_1, \dots, i_k\},\ i_l = dpg(i_c) \\ \quad distance((\{i_1 \dots i_{c-1}, i_{c+1} \dots i_k\}, liste.(u, i_c)), (\{j_1 \dots j_{k'}\}, liste')) \\ \qquad +fusion_{nœuds}(i_k, i_c)\ \textbf{rmq :}\ \text{les données de } i_k \text{ sont modifiées} \\ \quad distance((\{i_l \dots i_{c-1}, i_k\}, liste.(a, i_c)), (\{j_1 \dots j_{k'}\}, liste')) \\ \qquad +fusion_{arcs}(i_k, i_c) + distance((\{i_1 \dots i_{l-1}\}, \emptyset), (\emptyset, \emptyset)) \\ \qquad +distance((\{i_{c+1} \dots i_k - 1, \emptyset), (\emptyset, \emptyset))\ \textbf{rmq :}\ \text{les données de } i_k \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sont modifiées} \\ pour\ chaque\ fils\ j_{c'}\ de\ j_{k'}\ dans\ \{j_1, \dots, j_{k'}\},\ j_{l'} = dpg(j_{c'}) \\ \quad distance((\{i_1 \dots i_k\}, liste), (\{j_1 \dots j_{c'-1}, j_{c'+1} \dots j_{k'}, liste'.(u, j_{c'}))) \\ \qquad +scission_{nœuds}(j_{k'}, j_{c'})\ \textbf{rmq :}\ \text{les données de } j_{k'} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sont modifiées} \\ \quad distance((\{i_1 \dots i_k\}, liste), (\{j_{l'} \dots j_{c'}, j_{k'}, liste'.(a, j_{c'}))) \\ \qquad +scission_{arcs}(j_{k'}, j_{c'}) + distance((\emptyset, \emptyset), (\{j_1 \dots j_{l'-1}\}, \emptyset)) \\ \qquad +distance((\emptyset, \emptyset), (j_{c'+1} \dots j_{k'-1}, \emptyset))\ \textbf{rmq :}\ \text{les données de } j_{k'} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sont modifiées} \end{array}\right. \tag{3.3}$$

Sinon $set\ i_l = dpg(i_k)\ and\ j_{l'} = dpg(j_{k'})$

$$MIN\left\{\begin{array}{llllll} distance(& (\{i_1 \dots i_{k-1}\} & , \emptyset) & , (\{j_1 \dots j_{k'}\} & , liste') &) + del(i_k) \\ distance(& (\{i_1 \dots i_k\} & , liste) & , (\{j_1 \dots j_{k'-1}\} & , \emptyset) &) + ins(j_{k'}) \\ distance(& (\{i_1 \dots i_{l-1}\} & , \emptyset) & , (\{j_1 \dots j_{l'-1}\} & , \emptyset) &) \\ \quad +distance(& (\{i_l \dots i_k\} & , liste) & , (\{j_{l'} \dots j_{k'}\} & , liste') &) \end{array}\right. \tag{3.4}$$

Remarquons tout d'abord que nous avons ajouté deux variables au calcul de la distance : $liste$ et $liste'$. Ces deux paramètres sont des listes renseignant sur la suite des fusions effectuées sur les nœuds i_k et j_k. Ces listes sont formées par des couples (a ou u, i) indiquant qu'il y a eu une fusion d'arcs (a) ou de nœuds (u) avec le nœud i. Ainsi, la forêt décrite par $\{i_1, \ldots, i_k\}$ et $liste$ correspond à la forêt $T(i_1, \ldots, i_k)$ pour laquelle i_k a été modifié selon la suite de fusions décrites par $liste$.

Par ailleurs, l'algorithme d'origine utilise des intervalles pour faire référence aux forêts en cours de comparaison. Dans le cas de la fusion, il nous faut faire explicitement référence à l'ensemble des nœuds formant les forêts. En effet, lorsqu'il y a une fusion entre la racine i_k et l'un de ses fils i_c, le nœud i_c est supprimé de la liste des nœuds formant la forêt.

En suivant un raisonnement identique à celui de Zhang et Shasha, on voit que pour obtenir un algorithme polynomial, on doit maintenir en mémoire le calcul de la distance d'édition entre tous les sous-arbres de T et T', c'est-à-dire conserver la distance entre tous les $(i, liste)$, $(j, liste')$. Le couple $(i, liste)$ désigne le sous-arbre de T enraciné au nœud t_i et tel que les fusions décrites par la liste $liste$ ont été appliquées à t_i.

La complexité est donc liée au nombre de valeurs différentes que la variable $liste$ peut prendre pour chaque nœud des deux arbres. Supposons que l'on a un arbre complet de degré d, on note $LISTE_l = \{(a/u, f_1), \ldots, (a/u, f_l)\}$ l'ensemble des suites $liste$ de longueur l possibles, c'est-à-dire le nombre de façons différentes d'effectuer l fusions **successives** à partir de la racine r de cet arbre. La première fusion ne peut être effectuée qu'à partir des fils de r. Ainsi il y a d choix possibles pour f_1. La deuxième fusion peut se faire avec tous les fils ou les petits fils de r, c'est-à-dire avec les nœuds de l'arbre dont la hauteur est inférieur ou égale à deux (la hauteur de la racine est 0), il y en a $d + d * d$. On a ainsi $d + d^2$ choix possibles pour f_2. De même, f_l peut être n'importe lequel des descendants de r dont la hauteur est inférieure ou égale à l, il y en a $d + d^2 + \cdots + d^l = \frac{d^{l+1}-1}{d-1}$. On peut alors déterminer le nombre P_f de suites possibles pour $(f_1 \ldots f_l)$:

$$P_f = \prod_{i=1}^{i=l} \sum_{j=1}^{j=i} d^i = \prod_{i=1}^{i=l} \frac{d^{i+1} - 1}{d - 1}$$

c'est-à-dire

$$P_f = (\frac{1}{d-1})^l \prod_{i=2}^{i=l+1} (d^i - 1) < (\frac{1}{d-1})^l d^{\frac{(l+1)(l+2)}{2}}$$

Pour chacune des suites $(f_1 \dots f_l)$, on peut effectuer une fusion de nœuds ou d'arcs avec le nœud f_i, on a donc $2^{l+1}-1$ possibilités différentes de fusionner r avec une suite particulière. Ainsi, on peut en déduire que la racine a $O((2d)^l)$ façons différentes de fusionner l fois de suite.

Si l'on note d le degré maximal de T, d' celui de T', n le nombre de nœuds de T, n' celui de T' et l le nombre maximal de fusions **consécutives** qu'un nœud de l'arbre peut effectuer, l'algorithme devra conserver en mémoire la distance entre les $O(n*(2d)^l)$ arbres possibles de T et les $O(n'*(2d')^l)$ de T'. Il en résulte que la complexité mémoire de notre algorithme est :

$$O(n*n'*(2d)^l*(2d')^l)$$

et la complexité en temps est :

$$O(n*n'*(2d)^l*(2d')^l*min(haut(T),|RG(T)|)*min(haut(T'),|RG(T')|))$$

où $haut(T)$ est la hauteur de T et $|RG(T)|$ est son nombre de racines gauches (voir l'algorithme de Zhang et Shasha dans la partie précédente).

Il est important de noter que cette complexité ne dépend pas du nombre total de fusions pouvant être faites aux arbres mais du nombre de fusions successives pouvant être fait en chacun des nœuds.

Remarquons aussi qu'en utilisant les fonctions de coûts décrites précédemment (avec paramètre de fusion) pour les opérations de fusions, une fusion d'arcs ne peut suivre une fusion de noeuds, ainsi le nombre de fusions successives possibles par noeud est $O(d^l)$.

Dans le cas des ARN le degré des arbres est souvent faible (inférieur à 5 en général) et nous utilisons un nombre de fusions successives maximal assez faible (2 ou 3), ce qui rend cet algorithme tout à fait adapté à la comparaison des structures secondaires d'ARN comme nous allons le voir.

3.1.4 Résultats pratiques

Lorsque l'on utilise des arbres dont les nœuds représentent les éléments de structure afin de comparer des ARN, ceux-ci sont relativement petits. Par exemple, la petite sous-unité de l'ARN ribosomal de *Sulfolobus acidocaldarius* (D14876) est composée de 1147 bases. Si l'on code sa structure par un arbre où les nœuds représentent les paires de bases et les feuilles les bases non appariées, on obtient 440 nœuds internes et 567 feuilles, soit 1007 nœuds dans

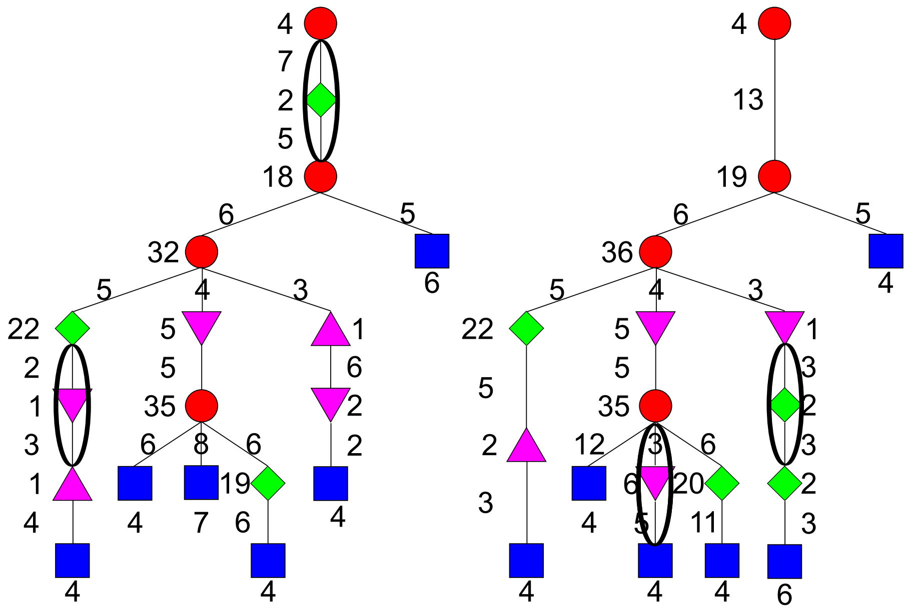


FIG. 3.14 – Résultat de l'édition d'arbre avec les opérations de fusion sur les arbres de la figure 3.4. Les fusions d'arcs sont représentées par une ellipse entourant les arcs fusionnés.

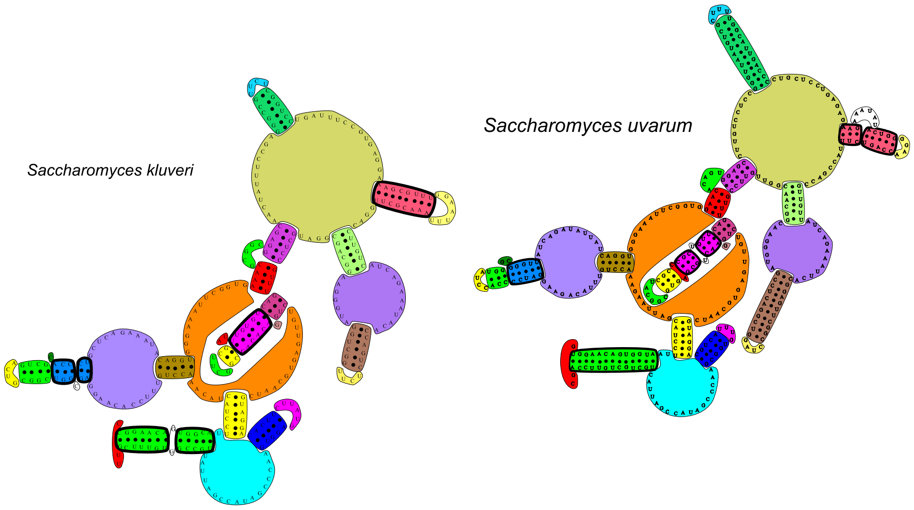


FIG. 3.15 – Résultat de l'édition d'arbre avec les opérations de fusion de la figure 3.14 sur les ARN. Les contours en gras indiquent les hélices fusionnées.

l'arbre. Dans le cas d'un arbre où les nœuds codent pour les éléments de structure secondaire (boucles multiples, renflements, boucles internes, boucles terminales) et les arcs pour les hélices, on obtient un arbre composé de 78 nœuds. Un codage encore plus compact dans lequel on modélise les tiges, boucles terminales et boucles multiples mène à un arbre de 48 nœuds.

Ainsi, lorsque l'on code la structure secondaire des ARN par des arbres représentant les éléments de cette structure, on obtient des arbres de faible taille (moins de 100 nœuds). Si, de plus, on limite le nombre de fusions successives à des valeurs inférieures à 3, on obtient un algorithme tout à fait utilisable en pratique : quelques minutes de temps d'exécution et moins d'un giga-octet d'occupation mémoire.

Dans le cas des deux ARN sur lesquels nous avons observé le problème de l'association bijective de l'édition d'arbre (Figure 3.3), nous avons obtenu avec succès les fusions escomptées. Le résultat de la comparaison des arbres est présenté dans la figure 3.14.

Nous avons ainsi un algorithme permettant de résoudre deux des trois problèmes signalés au début de ce chapitre. Le troisième problème est l'effet de dispersion observé lors de l'édition dans des régions à faible similarité.

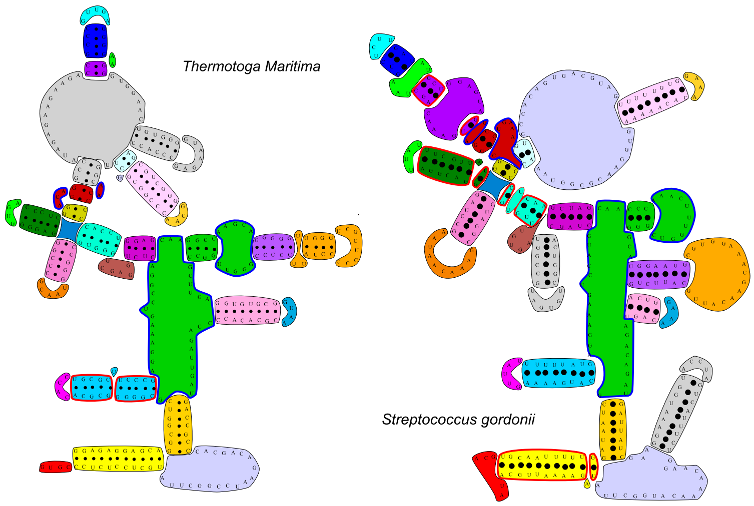


FIG. 3.16 – Résultat de l'édition d'arbre avec les opérations de fusion pour la comparaison des ARN de la figure 3.1. Les contours en gras indiquent les fusions de nœuds (contour bleu) et d'arcs (contour rouge). Les zones grises correspondent aux nœuds détruits/insérés dans le calcul de la distance d'édition.

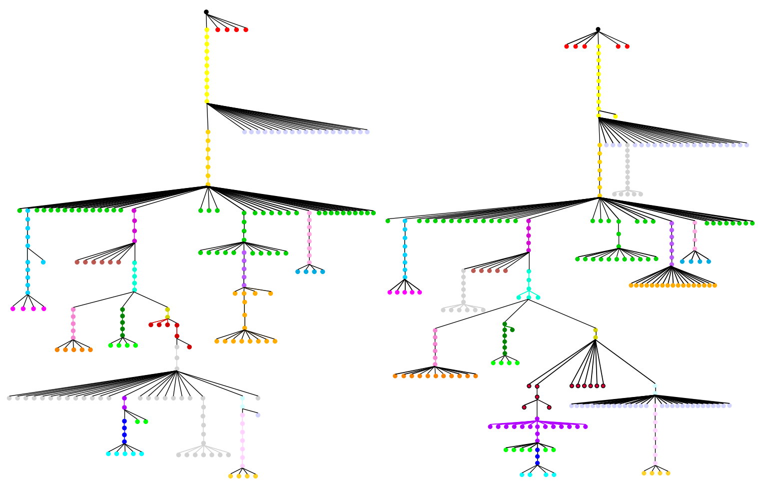


FIG. 3.17 – Ces arbres représentent les deux ARN de la figure 3.1. Les couleurs correspondent aux couleurs de la figure 3.16.

Si l'on se place dans le cadre général de la comparaison d'arbres ordonnés, ce problème n'a pas de solution. Cependant, dans le contexte de la comparaison de structures secondaires d'ARN, on peut réduire, voir éliminer ce problème en prenant en compte des informations de plus haut niveau tel que le type d'élément de structure secondaire dont fait partie une base libre. Reconsidérons les deux structures de la figure 3.1 que nous avions modélisé par un arbre codant les paires de bases et les bases non appariées. Nous pouvons tout à fait représenter ces ARN par des arbres codant pour les éléments de structure secondaire, puis effectuer l'édition de ces deux arbres en utilisant la fusion. Le résultat est montré dans la figure 3.16. Ensuite, nous pouvons exploiter ce résultat en reportant sur les arbres de la figure 3.2 les informations correspondantes, c'est-à-dire, par exemple, affecter une couleur à chaque nœud de l'arbre. Si deux nœuds correspondent à des bases faisant partie d'un même élément structural, ils ont la même couleur. Si un nœud u du premier arbre T a la même couleur qu'un nœud u' du deuxième arbre T', c'est que u fait partie d'un élément structural du premier ARN associé à un élément structural du deuxième ARN dont u' fait partie. On peut ainsi effectuer le calcul d'édition en prenant en compte ces couleurs.

Le modèle que nous allons maintenant introduire reprend cette idée : construire plusieurs arbres pour une même structure secondaire, chacun fournissant une vue différente de cet ARN.

3.2 MiGaL : "Multiple Graph Layer"

Après de nombreuses discussions avec des biologistes, il est apparu que le réseau de boucles multiples d'un ARN forme ce que l'on pourrait appeler son squelette. Celui-ci est caractéristique dans un grand nombre de familles d'ARN. De plus, comme nous l'avons déjà énoncé précédemment, il est admis que la conservation en structure est plus forte que la conservation en nucléotides sur l'ensemble de la séquence de l'ARN (ceci n'est pas forcément vrai localement). Ces deux faits biologique nous ont été confirmés par l'étude d'un certain nombre de familles d'ARN, dont les RNaseP et des ARNr.

Ainsi, nous avons mis en place un modèle basé sur les représentations arborescentes précédemment décrites prenant en compte ces différentes observations, c'est-à-dire travaillant d'abord sur la structure générale d'un ARN puis sur les similarités locales.

Dans une première section, nous présentons une définition formelle de notre modèle. Puis nous exposons une des façons d'utiliser ce modèle dans le cadre de la comparaison des ARN. La comparaison de structures secondaires ainsi modélisées sera abordée avant de voir les premiers résultats pratiques obtenus à ce jour.

3.2.1 Description du modèle

Nous allons maintenant voir une structure consistant en un ensemble de graphes liés entre eux. Nous avons choisi d'appeler cette structure MiGaL pour *MultIple GrAph Layer*. La section suivante est consacrée à l'utilisation de MiGaL pour la comparaison de structures secondaires d'ARN.

Définition 20 (MiGaL). *On définit une structure de type MiGaL, que l'on note $M(G, R)$, comme un ensemble G de N graphes et un ensemble R de $N - 1$ applications appelées* raffinements. *Chaque graphe $G_i(S_i, A_i)$ de G est composé d'un ensemble S_i de sommets et un ensemble A_i d'arcs. Le raffinement α_i est une application de S_i dans $\mathcal{P}(S_{i+1})$, c'est-à-dire qu'un sommet de S_i a pour image un sous-ensemble de S_{i+1}. L'application réciproque $\beta_i = {\alpha_i}^{-1}$ est une application surjective appelée* abstraction *qui à tout sommet de S_{i+1} associe un sommet de S_i. On pourra aussi définir une structure MiGaL par $M(G, A)$, c'est-à-dire par l'ensemble des graphes de M ainsi que des abstractions entre ces graphes. Enfin, M doit satisfaire aux deux conditions suivantes :*

1. *Pour tout sommet s de S_i, le sous-graphe de G_{i+1} induit par l'ensemble $r = \alpha_i(s)$ de sommets de S_{i+1} est connexe.*
2. *Pour tout arc de A_i connectant le nœud s au nœud s' dans le graphe G_i, il existe un arc de A_{i+1} entre un nœud de $\alpha_i(s)$ et un nœud de $\alpha_i(s')$.*

La figure 3.18 montre un exemple de structure de type MiGaL. Comme on peut le voir, notre modèle se prête bien à la représentation d'objets à différents niveaux de précision, les applications d'abstractions et raffinements permettant de faire le lien entre ces niveaux.

Voyons maintenant comment mettre en œuvre MiGaL pour modéliser les structures secondaires des ARN.

3.2.2 RNA-MiGaL

Nous allons présenter une des modélisations possibles des structure secondaires d'ARN utilisant MiGaL. Pour cela, chaque graphe de la structure

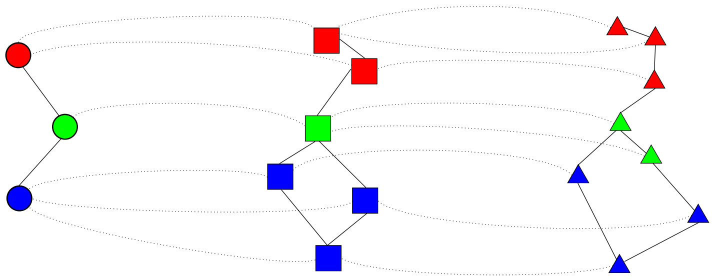


Fig. 3.18 – Exemple d’une structure de type MiGaL composée de trois graphes.

MiGaL sert à représenter la structure secondaire d’un ARN vue selon une granularité particulière. Les applications d’abstractions et de raffinements permettent de relier ces vues entre elles.

Définition 21 (RNA-MiGaL). *On appelle RNA-MiGaL une structure de type MiGaL $M_{RNA}(T, A)$ composée de 4 arbres ordonnés $T = T_0, T_1, T_2, T_3$, enracinés, étiquettés. Ces quatres arbres sont définis comme suit :*

- *L’arbre T_3 représente les bases libres et les paires de bases de la structure secondaire de l’ARN. Un nœud interne de T_3 code pour une paire de bases, une feuille pour une base libre.*
- *Dans l’arbre T_2, les arcs représentent les hélices de la structure. Un nœud de T_2 peut coder pour :*
  - *une boucle multiple.*
  - *une boucle interne.*
  - *un renflement.*
  - *une boucle terminale.*
- *L’arbre T_1 représente le réseau de tiges, boucles multiples et boucles terminales de la structure. Les arcs codent pour les tiges, les nœuds codent pour les boucles multiples et terminales.*
- *Le dernier arbre T_0 modélise le réseau de boucles multiples de la structure secondaire de l’ARN. Les nœuds représentent les boucles multiples, les arcs codent pour les tiges reliant deux boucles multiples.*

Les abstractions $\beta_0, \beta_1, \beta_2$ de M_{RNA} servent à modéliser la relation d’inclusion entre les différents nœuds de ces quatre arbres.

- *β_2 : à chaque nœud interne de T_3, représentant une paire de bases, on associe le nœud de T_2 tel que l’arc pointant sur ce nœud code pour*

l'hélice dont fait partie cette paire de base. À chaque nœud terminal de T_3 codant pour une base libre, β_2 associe le nœud de T_2 codant pour l'élément structural dont fait partie cette base.
- β_1 *: Lorsque le nœud de T_2 représente une boucle terminale ou une boucle multiple, il est mis en relation avec le nœud correspondant au même élément dans T_1. Si le nœud correspond à une boucle interne ou un renflement, celui-ci est mis en relation avec le nœud dont l'arc qui pointe dessus code pour la tige à laquelle appartient la boucle interne ou le renflement.*
- β_0 *: Les nœuds correspondant aux boucles multiples sont mis en relation avec les nœuds codant pour ces mêmes boucles multiples dans T_0. Un nœud correspondant à une boucle terminale est mis en relation avec le même nœud que son père (qui code forcément pour une boucle multiple).*

Nous verrons dans la section 3.2.5 sur les résultats pratiques obtenus, les étiquettes affectées aux nœuds et arcs de ces arbres.

De par la définition des abstractions et des arbres de RNA-MiGaL, on a la proposition suivante illustrée par la figure 3.19.

Proposition 14 (Ordre et parenté). *Dans une structure RNA-MiGaL, l'ordre entre les nœuds ainsi que le lien de parenté sont conservés par les applications d'abstractions et de raffinement à travers les arbres.*

Ainsi, soient u et v des nœuds du niveau $i+1$ et soient $u' = \beta_i(u)$ et $v' = \beta_i(v)$ leurs images dans l'arbre T_i par l'application β_i.

Si u est un des frères gauche de v, alors l'une des conditions suivantes est vérifiée :
- *u' est le père de v'.*
- *v' est le père de u'.*
- *u' est un des frères gauche de v'.*
- *u' et v' sont le même nœud.*

Si u est le père de v alors soit u' est le père de v', soit u' et v' sont le même nœud.

Démonstration. Ces conditions sont directement déduites du fait que tous les arbres d'une structure RNA-MiGaL représentent la même structure secondaire d'un ARN. Ainsi, comme chaque niveau code pour cette structure et, étant donnée la définition des abstractions, il en suit la propriété énoncée. □

La figure 3.20 donne un exemple d'une structure d'ARN et de la structure RNA-MiGaL qui lui est associée.

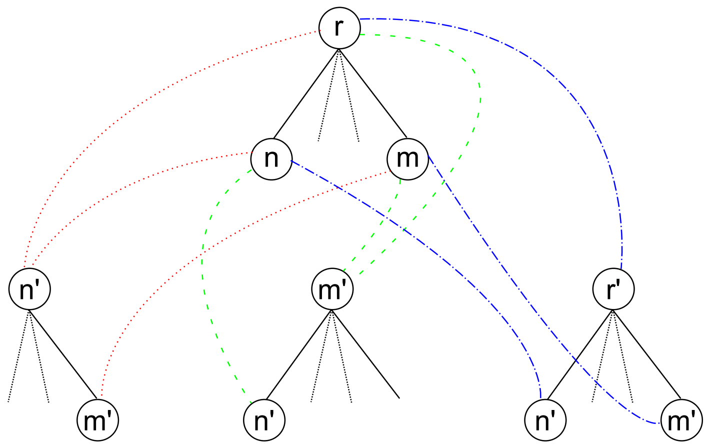


FIG. 3.19 – Conservation de l'ordre et du lien de parenté entre les nœuds au travers des applications d'abstractions dans une structure RNA-MiGaL.

En fait, il existe d'autres possibilités d'utilisation du type MiGaL pour la modélisation des structures secondaires d'ARN. En effet, on peut, par exemple, utiliser un type formé par seulement 3 arbres de RNA-MiGaL, ou encore ajouter un niveau T_4 dans lequel chaque nœud code pour une base. Nous discuterons de cela dans la partie résultat 3.2.5 à la fin de ce chapitre.

Pour comparer deux structures de type RNA-MiGaL, nous avons mis en place un algorithme basé sur l'édition d'arbre vu dans le chapitre précédent. Dans un premier temps, nous allons voir un algorithme général de comparaison de deux structures MiGaL, puis nous étudierons une version spécifique à RNA-MiGaL.

3.2.3 Algorithme d'édition à travers MiGaL

Dans un premier temps, nous allons voir l'algorithme général de comparaison de deux structures MiGaL. Une déclinaison spécifique à RNA-MiGaL sera étudiée ensuite.

Pour comparer deux structures MiGaL, on suppose que l'on a un algorithme extérieur qui prend en paramètre deux graphes colorés et qui renvoie une *association d'ordre k* entre ces deux graphes. Une *association d'ordre k* est une généralisation de la notion d'association que nous avons

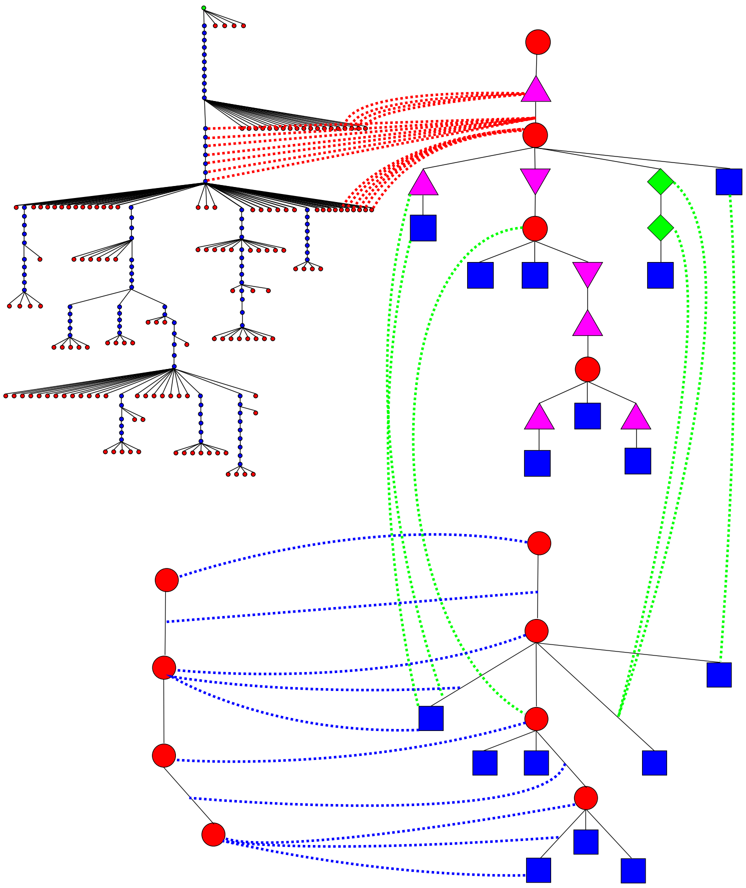


FIG. 3.20 – Exemple d'une structure de type RNA-MiGaL. Cette structure correspond à la RNaseP *Thermotoga maritima* de la figure 3.1. En haut à gauche, l'arbre T_3, à sa droite l'arbre T_2, dessous l'arbre T_1 et à gauche l'arbre T_0. Les pointillés rouges montrent l'abstraction β_2, en vert l'abstraction β_1 et en bleu l'abstraction β_0.

vue précédemment.

Définition 22 (Association d'ordre k entre deux graphes). *Soient deux graphes $G(S, A)$ et $G'(S', A')$. On définit une* association d'ordre k, $\mathcal{A}_k$, *comme étant un ensemble de couples de sous-ensembles des sommets de G et de sous-ensembles des sommets de G'.*

$$\mathcal{A}_k = \{(P, Q)|P \subset S, Q \subset S'\}$$

De plus, $\mathcal{A}_k$ doit vérifier les conditions suivantes :
- $\forall (P, Q) \in \mathcal{A}_k$, $|P| \leq k$ *et* $|Q| \leq k$
- $\forall (P_i, Q_i), (P_j, Q_j) \in \mathcal{A}_k$ $P_i \bigcap P_j = \emptyset$ *et* $Q_i \bigcap Q_j = \emptyset$

*Lorsque la valeur de k n'est pas fixée, on parlera alors d'*association généralisée. *On remarque que l'association précédemment définie entre deux arbres est une association d'ordre 1.*

L'idée générale de l'algorithme de comparaison de deux structures MiGaL est de comparer les graphes au niveau i. À partir de cette comparaison, on colore les graphes tels que deux sommets associés ont la même couleur. Enfin, on fait "descendre" ces couleurs sur les sommets des graphes du niveau $i+1$ à l'aide des raffinements α_i. On reprend alors la comparaison sur les deux graphes en tenant compte des couleurs.

On définit ainsi une association d'ordre k entre deux graphes colorés de la façon suivante :

Définition 23 (Association d'ordre k entre graphes colorés). *Un graphe coloré est un graphe dans lequel à chaque sommet est affectée une couleur. Les couleurs sont référencées par leurs indices, la couleur d'indice 0 correspondant à la couleur des sommets ne pouvant intervenir dans l'association. Une association $\mathcal{A}_k$ d'ordre k entre deux graphes colorés est une association d'ordre k qui vérifie les conditions suivantes :*
- $\forall (P, Q) \in \mathcal{A}_k$, $\forall p \in P, q \in Q$, *p et q sont de même couleur.*
- *Pour tous les sommets s de couleur 0, il n'existe pas de couple dans $\mathcal{A}_k$ contenant s.*

L'algorithme de comparaison de deux structures MiGaL $M(G, R)$, $M'(G', R')$ composées chacune de l graphes utilise un algorithme extérieur $\mathcal{C}$ calculant une association généralisée $\mathcal{A}$ entre 2 graphes colorés. L'initialisation de l'algorithme consiste à colorer les deux graphes de niveau 0 avec la même couleur. Puis, à chaque étape i, pour i allant de 0 à $l-2$, on fait :

Comparaison(M, M')

L : nombre de niveaux composant M et M'
couleur = 2
Initialiser tous les nœuds de T_0 et T'_0 à la couleur 1.
pour chaque niveau i de 0 à L
    $\mathcal{A} = association(T_i, T'_i)$
    pour chaque nœud u de T_i
        Si u est dans $\mathcal{A}$
            fixer la couleur des nœuds $\alpha_i(u)$ à *couleur*
        sinon
            fixer la couleur des nœuds $\alpha_i(u)$ à 0.
    pour chaque nœud u' de $T'i$
        Si u' est lié à u par $\mathcal{A}$
            fixer la couleur des nœuds $\alpha'_i(u')$ à la couleur des nœuds $\alpha_i(u)$.
        sinon
            fixer la couleur de u à 0.
Le résultat consiste en l'ensemble des associations calculées.

FIG. 3.21 – Algorithme général de comparaison de deux structures RNA-MiGaL M et M'. La procédure *association* calcule l'association générale entre deux graphes colorés.

- à l'aide de l'algorithme $\mathcal{C}$, calculer l'association généralisée $\mathcal{A}$ entre G_i et G'_i.
- Colorer les nœuds de G_{i+1} et G'_{i+1} à l'aide de $\mathcal{A}$. À chaque couple (P, Q) de $\mathcal{A}$ on associe une unique couleur $c \neq 0$. Puis, pour chaque nœud u de P (res. Q), on affecte la couleur c aux nœuds $\alpha_i(u)$ (resp. $\alpha'_i(u)$) de G_{i+1} (resp. G'_{i+1}).

La figure 3.21 donne le pseudo-code correspondant à cet algorithme. Le résultat de la comparaison consiste en l'union des associations calculées entre chaque graphe. Formellement, on définit cette association de la manière suivante :

Définition 24 (Association entre deux MiGaL). *Une association $\mathcal{B}$ entre deux structures MiGaL $M(G, R)$ et $M'(G', R')$ consiste en un ensemble de couples d'ensembles de sommets tels que :*
- $\forall (E, E') \in \mathcal{B}$, $\exists i$ *tq* $E \subset G_i$ *et* $E' \subset G_i$

– $\forall (E, E') \in \mathcal{B}$, $\forall s \in E$, $\forall s' \in E'$, $couleur(s) = couleur(s') \neq 0$.
– $\forall (E, E') \in \mathcal{B}$, *tq* $E \subset G_i$ *et* $E' \subset G_i$:
$\forall s \in E$, $\forall s' \in E'$
$couleur(\beta_i(s)) = couleur(\beta'_i(s')) \neq 0$.

La première condition implique que l'association met en liaison des sommets de graphes de même niveau. La deuxième condition impose que les sommets associés sont d'une même couleur différente de la couleur 0. Enfin, la troisième condition implique que si un sommet s est impliqué dans l'association avec un sommet s', alors son sommet correspondant à travers l'application d'abstraction est lui aussi impliqué dans l'association. De plus, il est lié au sommet correspondant à l'abstraction de s'.

Si l'algorithme utilisé pour comparer deux graphes a pour complexité $f(n)$, n étant la taille des deux graphes, alors l'algorithme de comparaison de deux structures MiGaL composées de k graphes de taille n est $O(k * f(n) + k * n)$. Cependant il est important de noter que dans cet algorithme, lorsqu'un choix est effectué au niveau i, il ne peut être remis en cause au niveau $i+1$. On peut ainsi espérer que l'algorithme de comparaison de graphes tire parti du coloriage pour réduire la complexité générale de la comparaison de deux structures MiGaL. De plus, dans cette complexité, nous supposons que les graphes qui composent une structure MiGaL ont tous la même taille. Or, l'idée même de MiGaL est d'avoir des graphes dont la taille croît avec les niveaux. Ainsi, la comparaison initiale est moins coûteuse et permet de réduire la complexité de la comparaison suivante, et ainsi de suite.

En outre, il ne nous est pas possible ici de réduire la complexité de la comparaison des graphes en calculant de manière séparée les comparaisons entre les sous-graphes de même couleur. En effet, comme nous allons le voir dans le cas de la comparaison de deux RNA-MiGaL, un tel découpage peut nous amener à une comparaison globale fausse au sens de l'algorithme utilisé.

3.2.4 Comparaison de RNA-MiGaL

Dans le cas RNA-MiGaL, nous avons utilisé un algorithme d'édition d'arbre pour faire la comparaison de deux niveaux. Dans un premier temps, nous allons voir une approche naïve consistant à effectuer l'édition de la totalité des arbres à chaque niveau, puis nous verrons une amélioration permettant de comparer séparément chaque sous-arbre de même couleur.

Algorithme naïf

La version naïve de l'édition d'arbres classique avec prise en compte de la couleur des nœuds consiste tout simplement à conditionner la substitution par la couleur des nœuds. Ainsi, si l'on dispose d'une fonction $couleur(i)$ qui retourne la couleur du nœud i, on obtient les formules suivantes :

$$\boxed{distance(i_1 \ldots i_2, j_1 \ldots j_2) =}$$

Si $((i_1 == dpg(i_2))\ et\ (j_1 == dpg(j_2)))$

$$MIN \begin{cases} distance(& i_1 \ldots i_2 - 1 & , j_1 \ldots j_2 &) + score(i_2, \epsilon) \\ distance(& i_1 \ldots i_2 & , j_1 \ldots j_2 - 1 &) + score(\epsilon, j_2) \\ \mathbf{Si\ couleur(i_2) == couleur(j_2) \neq 0} \\ \quad distance(& i_1 \ldots i_2 - 1 & , j_1 \ldots j_2 - 1 &) + score(i_2, j_2) \end{cases} \tag{3.5}$$

Sinon

$$MIN \begin{cases} distance(& i_1 \ldots i_2 - 1 & , j_1 \ldots j_2) &) + score(i_2, \epsilon) \\ distance(& i_1 \ldots i_2) & , j_1 \ldots j_2 - 1 &) + score(\epsilon, j_2) \\ distance(& i_1 \ldots dpg(i_2) - 1 & , j_1 \ldots dpg(j_2) - 1 &) \\ \quad +distance(& dpg(i_2) \ldots i_2 & , dpg(j_2) \ldots j_2 &) \end{cases} \tag{3.6}$$

Cette modification ne change en rien la complexité de l'algorithme.

Dans le cas de l'algorithme avec fusion, nous devons aussi veiller à ce que les fusions ne se produisent qu'entre nœuds de même couleur.

$$\boxed{distance((\{i_1, \ldots, i_k\}, liste), (\{j_1, \ldots, j_{k'}\}, liste')) =}$$

Si $((i_1 \geq dpg(i_k))\ et\ (j_1 \geq dpg(j_{k'})))$

$$MIN \left\{ \begin{array}{l}
distance(\quad (\{i_1 \dots i_{k-1}\} \quad ,\emptyset) \quad ,(\{j_1 \dots j_{k'}\} \quad ,liste') \quad) + del(i_k) \\
distance(\quad (\{i_1 \dots i_k\} \quad ,liste) \quad ,(\{j_1 \dots j_{k'-1}\} \quad ,\emptyset) \quad) + ins(j_{k'}) \\
\textbf{Si couleur}(\mathbf{i_k}) == \textbf{couleur}(\mathbf{j_{k'}}) \neq \mathbf{0} \\
\quad distance(\quad (\{i_1 \dots i_{k-1}\} \quad ,\emptyset) \quad ,(\{j_1 \dots j_{k'-1}\} \quad ,\emptyset) \quad) + sub(i_k, j_{k'}) \\
pour\ chaque\ fils\ i_c\ de\ i_k\ dans\ \{i_1, \dots, i_k\},\ i_l = dpg(i_c) \\
\quad \textbf{Si couleur}(\mathbf{i_k}) == \textbf{couleur}(\mathbf{i_c}) \neq \mathbf{0} \\
\quad\quad distance((\{i_1 \dots i_{c-1}, i_{c+1} \dots i_k\}, liste.(n, i_c)), (\{j_1 \dots j_{k'}\}, liste')) \\
\quad\quad\quad +fusion_{nœuds}(i_k, i_c)\ \textbf{rmq :} \text{ les données de } i_k \text{ sont modifiées} \\
\quad\quad distance((\{i_l \dots i_{c-1}, i_k\}, liste.(a, i_c)), (\{j_1 \dots j_{k'}\}, liste')) \\
\quad\quad\quad +fusion_{arcs}(i_k, i_c) + distance((\{i_1 \dots i_{l-1}\}, \emptyset), (\emptyset, \emptyset)) \\
\quad\quad\quad +distance((\{i_{c+1} \dots i_k - 1, \emptyset), (\emptyset, \emptyset))\ \textbf{rmq :} \text{ les données de } i_k \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sont modifiées} \\
pour\ chaque\ fils\ j_{c'}\ de\ j_{k'}\ dans\ \{j_1, \dots, j_{k'}\},\ j_{l'} = dpg(j_{c'}) \\
\quad \textbf{Si couleur}(\mathbf{j_{k'}}) == \textbf{couleur}(\mathbf{j_{c'}}) \neq \mathbf{0} \\
\quad\quad distance((\{i_1 \dots i_k\}, liste), (\{j_1 \dots j_{c'-1}, j_{c'+1} \dots j_{k'}, liste'.(n, j_{c'})))) \\
\quad\quad\quad +scission_{nœuds}(j_{k'}, j_{c'})\ \textbf{rmq :} \text{ les données de } j_{k'} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sont modifiées} \\
\quad\quad distance((\{i_1 \dots i_k\}, liste), (\{j_{l'} \dots j_{c'}, j_{k'}, liste'.(a, j_{c'})))) \\
\quad\quad\quad +scission_{arcs}(j_{k'}, j_{c'}) + distance((\emptyset, \emptyset), (\{j_1 \dots j_{l'-1}\}, \emptyset)) \\
\quad\quad\quad +distance((\emptyset, \emptyset), (j_{c'+1} \dots j_{k'-1}, \emptyset))\ \textbf{rmq :} \text{ les données de } j_{k'} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{sont modifiées}
\end{array} \right. \tag{3.7}$$

Sinon *set* $i_l = dpg(i_k)$ *and* $j_{l'} = dpg(j_{k'})$

$$MIN \left\{ \begin{array}{l}
distance(\quad (\{i_1 \dots i_{k-1}\} \quad ,\emptyset) \quad ,(\{j_1 \dots j_{k'}\} \quad ,liste') \quad) + del(i_k) \\
distance(\quad (\{i_1 \dots i_k\} \quad ,liste) \quad ,(\{j_1 \dots j_{k'-1}\} \quad ,\emptyset) \quad) + ins(j_{k'}) \\
distance(\quad (\{i_1 \dots i_{l-1}\} \quad ,\emptyset) \quad ,(\{j_1 \dots j_{l'-1}\} \quad ,\emptyset) \quad) \\
\quad +distance(\quad (\{i_l \dots i_k\} \quad ,liste) \quad ,(\{j_{l'} \dots j_{k'}\} \quad ,liste') \quad)
\end{array} \right. \tag{3.8}$$

Cette modification n'est pas sans conséquence sur les performances de l'algorithme. En effet, le nombre de fusions successives pouvant être réalisées à chaque nœud se trouve limité par le nombre de descendants de même couleur de ce nœud. Ainsi, comme nous le verrons dans la section résultats, en pratique nous pouvons autoriser un nombre de fusions successives beaucoup plus grand que dans le cas des arbres non colorés tout en conservant des temps de calcul performants.

D'un point de vue théorique, si l'on note m la taille maximale de chaque sous-structure de couleur, la complexité de l'algorithme devient dans le pire des cas :

$$O((2d)^m * n^4)$$

avec n la taille des deux arbres comparés.

Bien que ces approches soient utilisables en pratique, on pourrait espérer des algorithmes plus efficaces pour faire la comparaison d'arbres colorés tels que nous l'avons définie. C'est pour cela que nous avons mis en place un algorithme dont l'idée est de calculer séparément l'édition de chaque sous arbre.

Algorithme de partage des couleurs

Il paraît naturel de calculer de façon séparée la distance entre chaque sous-structure de même couleur. Dans un premier temps, nous allons définir formellement le problème de l'édition d'arbre à coloration connexe, puis nous verrons un exemple illustrant le problème de la conservation d'ordre entre les nœuds par une édition séparée. Enfin, nous donnerons un algorithme effectuant le calcul de la distance.

Définition 25 (Arbre à coloration connexe). *Soit T un arbre. On dit que T est à coloration connexe si à chaque nœud de T est affecté une couleur et que pour chaque couleur, le sous-arbre de T induit par les nœuds de cette couleur est connexe.*

Pour chaque couleur c d'un arbre T à coloration connexe, on définit l'ensemble $dep(c)$ comme l'ensemble des couleurs c' telles qu'il existe un nœud dans T de couleur c ayant pour fils un nœud de couleur c' dans T. Dans l'exemple de la figure 3.22, pour l'arbre en haut à gauche, on a $dep(bleu) = \{rouge, vert\}$ et $dep(rouge) = \emptyset$.

Définition 26 (Problème). *Soient deux arbres à coloration connexe T et T'. On impose que ces arbres soient colorés avec les mêmes couleurs et que pour chaque couleur c, $dep(c) = dep'(c)$. Le problème consiste à calculer la distance d'édition entre T et T' telle que la substitution ne mette en relation que des nœuds de la même couleur. Dans le cas de l'édition avec fusion, ces fusions ne peuvent se produire qu'entre nœuds de même couleur.*

La première idée que l'on peut avoir est d'effectuer de manière séparée le calcul de la distance entre chaque sous-structure de même couleur puis de regrouper le résultat de ces comparaisons. Comme le montre l'exemple qui

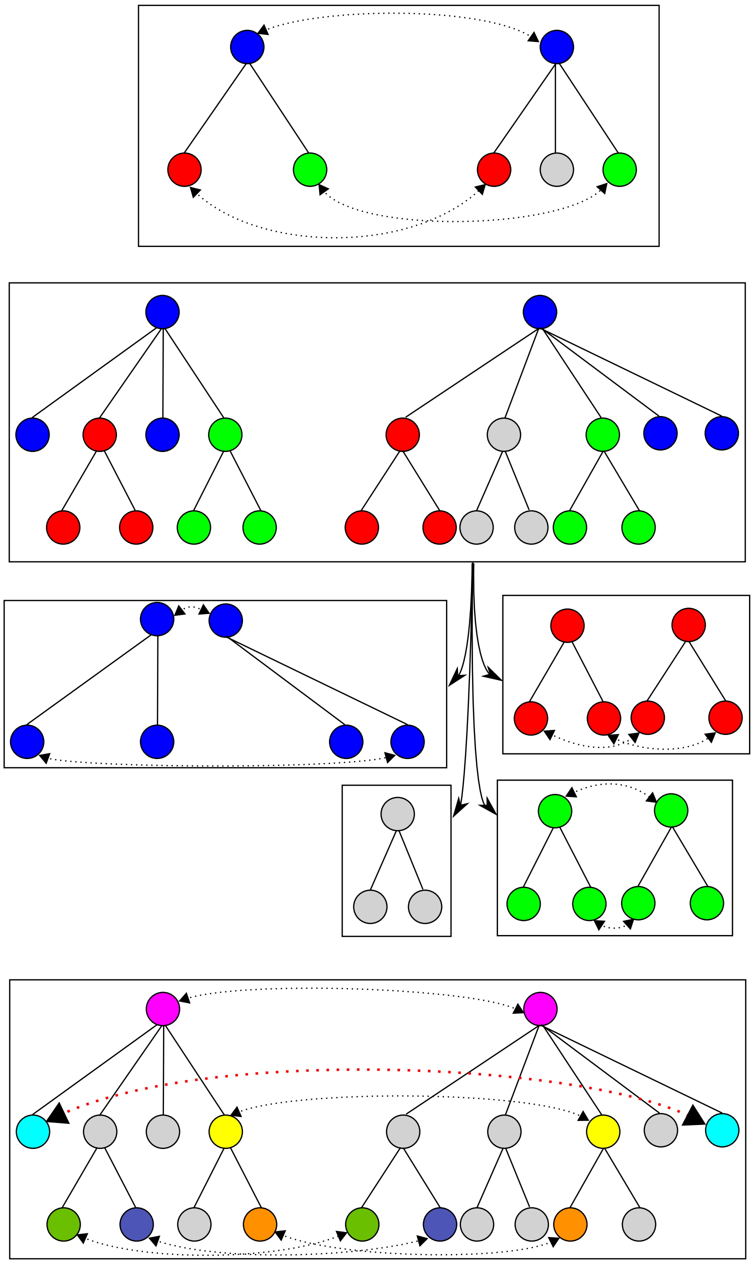


FIG. 3.22 – Exemple d'erreur pouvant apparaître lors de l'édition séparée des sous-structures.

suit, ceci pose un problème vis-à-vis de la conservation de l'ordre entre les fils par l'édition d'arbre.

Examinons le cas présenté dans la figure 3.22. En haut, nous voyons le résultat de la comparaison des deux arbres de niveau 0. En dessous, sont représentés les arbres de niveau 1 dans lesquels nous avons reporté la couleur des nœuds des arbres de niveau 0. Puis nous avons découpé chaque arbre en sous arbres en fonction des couleurs et nous avons appliqué le calcul d'édition entre chaque sous-structure de même couleur. La dernière figure représente les deux arbres de niveau 1 colorés en fonction du résultat des calculs de la distance d'édition. Comme on peut le voir, l'association résultante entre les deux arbres de niveau 1 est incorrecte dans le contexte de l'édition d'arbres **ordonnés**. En effet, comme le montre la flèche en rouge, l'association entre les deux nœuds bleus clairs ne respecte par l'ordre entre les nœuds. En fait, lors de la réunion des différents résultats d'édition, il faut se demander s'il est plus coûteux de ne pas associer les deux nœuds bleus clairs ou de ne pas associer les nœuds jaunes, bleus, verts et oranges.

Intuitivement, il apparaît que nous devons tenir compte de la dépendance qui existe entre les sous-structures de couleurs. Ainsi, nous devons tout d'abord calculer la distance entre les sous-structures de couleur c telle que $dep(c) = \emptyset$. En effet, celles-ci ne présentent pas de dépendance vis-à-vis des sous-structures des autres couleurs. Puis, nous devons considérer les couleurs restantes telles que pour chaque couleur c, la distance entre les sous-structures de couleurs $dep(c)$ soit déjà calculée. De plus, nous devons modifier les sous-structures de couleur c afin que la dépendance avec les autres sous-structures de couleurs $dep(c)$ soient prises en compte dans le calcul de la distance.

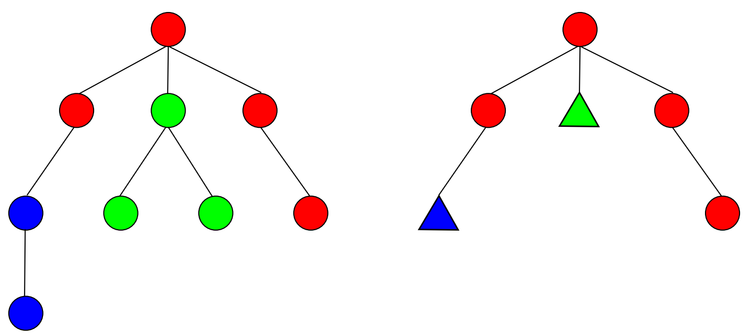


FIG. 3.23 – Ancres ajoutées à la sous-structure de couleur rouge.

Ainsi, supposons que l'on compare les deux sous-structures S_c et S'_c de couleur c des deux arbres T et T'. Nous allons ajouter des feuilles, que nous appellerons des ancres, dans l'arbre S_c. Une ancre sera adjointe au nœud u de S_c, si u possède un fils dont la couleur est différente de c dans T. Cette ancre sera étiquetée avec la couleur du fils dont elle a pris la place. Cette opération est illustrée par la figure 3.23. On fait de même pour la sous-structure S'_c.

Nous pouvons alors calculer la distance entre S_c et S'_c en imposant que l'ancre de couleur d de S_c ne puisse être substituée que par l'ancre de couleur d de S'_c et le score de cette substitution est égal à la distance entre S_d et S'_d. De plus, le score de la délétion de l'ancre de couleur d est égal à la distance de S_d à l'arbre vide. De même, le score de l'insertion de l'ancre de couleur d est égal à la distance entre l'arbre vide et S'_d. Dans le cas où l'on dispose des opérations de fusion, celles-ci ne peuvent avoir lieu avec une ancre.

Du fait que nous ordonnons le calcul de distance entre les sous-structures selon leur dépendance en couleur, tous les scores relatifs aux ancres décrits ci-dessus sont déjà établis lors de l'édition de S_c et S'_c.

La complexité de cet algorithme dépend du nombre de couleurs et surtout de la taille des sous-structures pour chaque couleur. Ainsi, si l'on suppose que les deux arbres sont colorés avec k couleurs différentes, chaque couleur définissant une sous-structure S_k dans T et S'_k dans T' ($\sum_k |S_k| = |T|$ et$\sum_k |S'_k| = |T'|$), la complexité en temps du calcul de l'édition dans le pire des cas de l'algorithme est

$$O(\sum_k |S_k|^2 |S'_k|^2)$$

dans le cas où l'on utilise les opérations d'édition classiques et

$$O((2d)^l \sum_k |S_k|^2 |S'_k|^2)$$

avec les opérations de fusion. La complexité en mémoire quant à elle est le maximum de la complexité mémoire du calcul pour chaque sous-arbre, c'est-à-dire :

$$O(max_k(|S_k| * |S'_k|)^2)$$

et

$$O((2d)^{2l} max_k(|S_k| * |S'_k|)^2)$$

pour l'édition avec fusion.

3.2.5 Résultats pratiques

Pour clore notre travail sur MiGaL, nous allons maintenant présenter les résultats que nous avons obtenus en pratique. Dans un premier temps, nous donnerons une description des étiquettes utilisées dans les arbres de RNA-MiGaL. Une fois ces étiquettes établies, nous définissons les fonctions de coût des différentes opérations d'édition. Avant de conclure par un exemple de comparaison d'ARN selon notre modèle, nous fournirons quelques détails quant à l'implémentation réalisée.

Définition de RNA-MiGaL

En pratique, nous avons choisi d'étiqueter les arbres $T_{0...3}$ de la manière suivante :

- Un nœud de T_3 qui code pour une paire de bases est étiqueté par les deux bases formant cette paire. Une feuille est étiquetée selon la base libre qu'elle représente.
- Dans l'arbre T_2, les arcs qui représentent les hélices de la structure, contiennent une étiquette indiquant le nombre de paires de bases formant cette hélice. Un nœud de T_2 peut coder pour :
  - une boucle multiple : l'étiquette de ce nœud est alors une suite de valeurs indiquant la taille de chacune des séquences de bases libres formant cette boucle multiple.
  - une boucle interne ou un renflement : l'étiquette consiste en deux valeurs, la première donnant la taille de la séquence libre de la boucle interne du côté 5', l'autre la taille de la séquence libre coté 3'. Dans le cas d'un renflement, une de ces deux valeurs est nulle.
  - une boucle terminale : l'étiquette est la taille de cette boucle.
- L'arbre T_1 représente le réseau de tiges, boucles multiples et boucles terminales de la structure. Les arcs, codant pour les tiges, sont étiquetés par deux valeurs indiquant le nombre de bases libres et le nombre de paires de bases contenues par la tige. Un nœud code pour une boucle multiple ou une boucle terminale, l'étiquette donne le nombre de bases libres formant cette boucle.
- Le dernier arbre T_0 code le réseau de boucles multiples de la structure secondaire de l'ARN. Les nœuds représentent les boucles multiples, ils sont étiquetés par le nombre de tiges connectées à cette boucle. Les arcs codent pour les tiges et contiennent le nombre de bases libres et

de paires de bases de cette tige.

Ainsi, nous avons choisi de n'apporter les informations sur les nucléotides qu'au niveau T_3, les autres niveaux comportant des informations sur la dimension des éléments. Il existe bien entendu de nombreuses autres possibilités pour ces étiquettes telles que l'energie libre, la composition en nucléotides etc. Nous avons volontairement choisi des attributs "relativement" simples à comparer pour nos premières expériences, ceux-ci pouvant évoluer en fonction des résultats obtenus.

Fonctions de coût

Le choix des fonctions de coût est un problème important et délicat dans le cadre du calcul de la distance d'édition avec fusion. En effet, comme nous l'avions vu, la fusion est choisie si elle améliore le score de substitution d'au moins t, paramètre de la fusion (voir section 3.1.2). Les fonctions de coûts associées aux opérations de fusion sont définies comme indiqué dans la section 3.1.2), c'est-à-dire :

$$fusion_{nœuds}(u, v, r) = del(v) + t = del_a(v) + del_n(v) + t$$

et

$$fusion_{arcs}(u, v, r) = \sum_{s \text{ frère de } v} del_{st}(s) + del(u) + t$$

Nous devons maintenant définir les fonctions de délétion, insertion et substitution pour les étiquettes des nœuds ainsi que celles des arcs pour chaque niveau.

Rappelons la condition émise sur la fonction de délétion : Quelles que soient deux étiquettes a et b, dont la fusion produit l'étiquette c, on doit avoir : $del(c) \geq del(a) + del(b)$.

Pour répondre à cette condition sur la fonction de délétion (et donc d'insertion), nous avons décidé que le coût de la délétion d'un nœud ayant résulté de la fusion de deux (ou plus) nœuds coûte $+\infty$. Il en est de même pour le fusion d'arcs. Ce choix est justifié par le fait que la fusion n'est pas une opération destinée à détruire des arcs et des nœuds mais à améliorer la substitution.

Outre ce cas spécial de la délétion, dans les autres cas, nous avons utilisé des fonctions linéaires. Pour cela, nous avons choisi de manière empirique la valeur pour laquelle la délétion vaut 1, c'est-à-dire l'inverse de la pente de la fonction. Par exemple, dans le cas de la délétion d'un arc de l'arbre de niveau 2 correspondant à une hélice, nous avons choisi de fixer le score à 1 lorsque le nombre de paires de bases vaut 5.

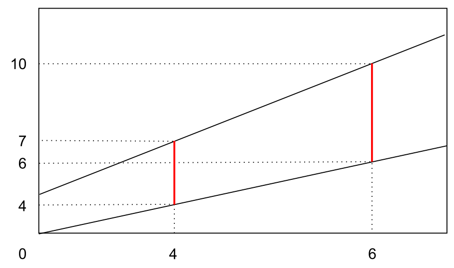


FIG. 3.24 – Fonction linéraire donnant, pour une valeur x, la valeur y telle que la substitution de x à y coûte 1. Ainsi, la substitution de 6 en 10 coûte 1, de 6 en 6 coûte 0 et de 6 en 8 coûte 0.5.

Pour la fonction de substitution, nous utilisons une fonction affine qui associe à x la valeur y telle que la substitution de x par y coûte 1. En d'autres termes, cette fonction fournit la pente de la fonction linéaire utilisée pour calculer la substitution de x en y ($x \leq y$).

Afin d'assurer la cohérence des deux fonctions de délétion et de substitution, nous nous assurons lors de la définition de la fonction de substitution que la valeur de notre fonction en 0 est bien égale à la valeur dont la délétion coûte 1. Ainsi, la délétion devient un cas spécial de la substitution avec la valeur 0.

On voit ainsi, qu'il nous faut établir un grand nombre de paramètres pour les fonctions de coûts et maintenir la cohérence entre ces fonctions (substitution et délétion). Actuellement nous avons établi ces valeurs de façon empirique, il est prévu qu'elles soient prochainement établies sur la base des expériences et résultats obtenus avec RNA-MiGaL.

En ce qui concerne le niveau 3, les étiquettes ne consistent pas en des valeurs numériques mais en des nucléotides. Pour la comparaison des arbres T_3, nous avons utilisé la distance d'édition classique (la fusion n'a pas de sens dans ce cas) avec des fonctions de coûts unitaires, c'est-à-dire que la substitution d'une lettre par elle-même coute 0, 1 si les lettres sont différentes et la délétion d'une lettre coûte 1.

Implémentation et performances

Nous avons implémenté la construction de RNA-MiGaL, l'algorithme de calcul de distance d'édition classique et avec fusion ainsi que l'algorithme "naïf" de comparaison de deux structures RNA-MiGaLs (sans le partage utilisant les couleurs). Pour cela, nous avons choisi d'utiliser un language de programmation objet : le C++. L'ensemble de ce projet se décompose en 63 classes répartie selon 4 espaces de noms.

Les classes faisant partie de **data** définissent les types de nœuds et arcs de chacun des arbres de RNA-MiGaL, toutes dérivent d'une classe *Object*, puis soit de la classe *Node*, soit de la classe *Edge*.

L'espace **visitor** regroupe l'ensemble des visiteurs sur les classes de **data**. Ils définissent l'ensemble des méthodes que l'on souhaite appliquer à un ou deux objets de **data** telles que le calcul de coût de la délétion ou encore la construction du nœud résultant de la fusion de deux autres nœuds.

Dans **edit** on trouvera les algorithmes de comparaison de deux arbres enracinés selon l'algorithme classique ainsi que l'algorithme avec fusion. L'algorithme permettant la comparaison de deux structures RNA-MiGaLs se trouve aussi dans cet espace.

Enfin, l'espace de nom **util** constitue l'ensemble des classes utilitaires pour l'application. Entre autres, c'est dans cet espace que nous avons placé les objects permettant le chargement de fichier ainsi que l'allocateur des objets de l'application basé sur *mmap*.

Le code, ainsi que sa documentation au format html, sous licence GPL/FDL [58][57], est disponible sur simple demande, sa mise en ligne étant prévue une fois que l'algorithme utilisant le partage par couleur sera implémenté.

Enfin, en plus du programme **migal** permettant la comparaison de deux ARN, soit en utilisant l'édition sur les arbres ordonnés de chaque niveau séparément, soit en utilisant le modèle RNA-MiGaL, nous avons écrit deux petits utilitaires. Le premier, **rnaconverter**, permet de charger les fichiers au format *ct*, *ct2* ou encore *bpseq* et de sauvegarder la structure RNA-MiGaL correspondante au format XML. Le deuxième, **rna2tex**, dessine dans un fichier *LaTeX* les arbres de la structure RNA-MiGaL en utilisant la librairie *pstricks*.

Du point de vue des performances en temps, notre programme est tout à fait utilisable en pratique. Voici deux exemples de temps de calcul pour la comparaison de deux ARN sur une machine PC à 1GHz avec 256 méga de mémoire vive sur un système d'exploitation Debian GNU/Linux.

Nous avons effectué en une minute la comparaison de deux Introns de Groupe I extraits de la base [15], le premier provenant de *Acanthamœba griffini* (S81337) dont la séquence fait 526 nucléotides, le deuxième provenant de *Aureoumbra Lagunensis*(U40258) dont la séquence fait 468 nucléotides.

La deuxième comparaison concerne des ARN ribosomaux 16S composés de 1500 bases chacun appartenant respectivement à *Archaeoglobus fulgidus* (X05567 AE000965) et à *Aeropyrum pernix* (AP000062). La comparaison des structures RNA-MiGaL correspondantes à ces ARN a été calculée en quarante minutes.

Dans ces deux cas, la comparaison des arbres $T_{0...2}$ des structures RNA-MiGaLs prend moins de 5% du temps total de comparaison. En effet, ces arbres de haut niveau sont de faible taille tandis que les arbres de niveau 3 sont volumineux. Ils sont composés en moyenne de $\frac{3*n}{4}$ nœuds, n étant la longueur de la séquence.

Avec l'algorithme de partage de couleurs, on peut donc s'attendre à une forte amélioration de ces temps de calcul, de l'ordre de la minute pour la comparaison des deux ARN ribosomaux 16S.

Exemples

Nous allons donner un exemple de comparaison d'ARN utilisant RNA-MiGaL. Nous avons utilisé deux introns de Groupe I extraits de la base [15] et présentés sur la figure 3.26.

La figure 3.27 montre le résultat du calcul de la distance d'édition avec fusion sur les arbres de niveau 0. Entre autres, nous pouvons voir une fusion d'arcs au niveau des flèches sur l'arbre gauche.

La figure 3.28 illustre le résultat de l'édition d'arbre sur le niveau 1 des structures. On remarquera que la fusion d'arcs faite au niveau 0 est reproduite à ce niveau. On notera aussi la fusion de nœuds sur l'arbre droit entre les deux nœuds jaunes.

La figure 3.29 correspond au résultat de la comparaison au niveau 2 des deux RNA-MiGaLs. À ce niveau, certaines délétions ont été réalisées comme la tige boucle grise en haut à gauche de l'ARN de gauche. On notera aussi la fusion de nœuds entre la boucle multiple rouge et la boucle interne rouge de l'arbre gauche. Celle-ci s'est produite en partie parce que la boucle multiple de droite comporte beaucoup plus de bases que celle de gauche (certaines aux extrémités 5' et 3' n'ont pas été dessinées). Ainsi la réalisation de cette fusion conduit à l'augmentation de la taille de la boucle multiple et, ainsi, à l'amélioration du score de substitution avec la boucle multiple de l'ARN de droite. En effet, bien que les boucles multiples soient représentées par des suites de valeurs, chacune correspondant à la longueur d'une séquence de bases libres entre deux hélices, pour la comparaison nous utilisons la somme de ces longueurs. Il est fort probable que l'utilisation d'une fonction de coût plus sophistiquée comparant la longueur de chaque "morceau" permette d'éviter ce type de fusion non souhaité.

Enfin, on peut observer le résultat de la comparaison au niveau des nucléotides dans la figure 3.30. Du point de vue général, le résultat de la comparaison est plutôt satisfaisant. Cependant, si l'on regarde les bases en jaune en haut à droite de chaque structure, on peut voir un problème sur les appariements effectués. En effet, au niveau supérieur, les deux nœuds en jaune de l'arbre droit ont été fusionés afin d'être mis en relation avec le nœud jaune de l'arbre gauche. Cette fusion doit être vue comme la cassure des liaisons de l'hélice jaune. Le problème ici est que l'on ne peut reproduire cette cassure au niveau 3. Ainsi, les bases libres de l'ARN gauche, qui correspondent à des feuilles dans l'arbre, ne peuvent être mises en relation avec les paires de bases de l'ARN de droite, qui correspondent à des nœuds internes.

Actuellement, nous travaillons sur deux solutions simples pour résoudre ce problème. La première solution consiste à ajouter une opération d'édition permettant de transformer un paire de bases en deux bases non appariées. Une autre solution serait de changer la définition de l'arbre de niveau 3 de telle façon à ce que chaque base soit représentée par une feuille dans l'arbre, les nœuds internes modélisant alors une liaison entre le fils le plus à gauche et celui le plus à droite. Un tel arbre est présenté à droite dans la figure 3.25.

D'une façon générale, l'algorithme de comparaison a correctement associé les éléments structuraux de part et d'autre des structures. Il apparaît clairement que la distance d'édition avec les opérations de fusion est indispensable pour la comparaison des arbres de niveau 0, 1 et 2.

Conclusion

Les premiers résultats obtenus avec les structures RNA-MiGaLs sont très satisfaisants. En effet, la modélisation par niveau, outre son intérêt pour la comparaison, semble très utile pour appréhender les régions ayant des structures communes. Le niveau 0 nous permet de visualiser rapidement les différentes parties d'une structure, les niveaux suivants fournissant plus de

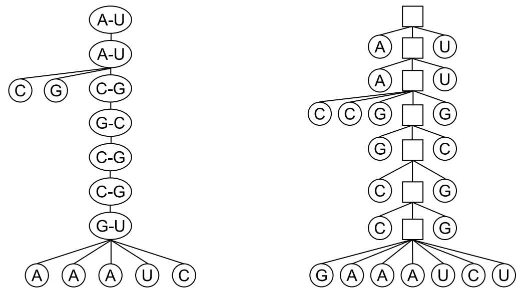


FIG. 3.25 – Deux codages possibles pour l'arbre de niveau 3 de RNa-MiGaL.

détails. Actuellement, seul le niveau 3 semble poser un problème de par l'impossibilité de séparer une paire de bases en deux bases libres. Cependant, nous avons déjà plusieurs solutions simples possibles pour résoudre ce problème.

Du point de vue pratique, il ne fait aucun doute que nos algorithmes répondent aux exigences de temps d'exécution d'espace mémoire occupée. L'implémentation de l'algorithme de comparaison utilisant le partage de couleurs devrait sans doute faire de RNA-MiGaL la méthode de comparaison de structures secondaires la plus rapide existante.

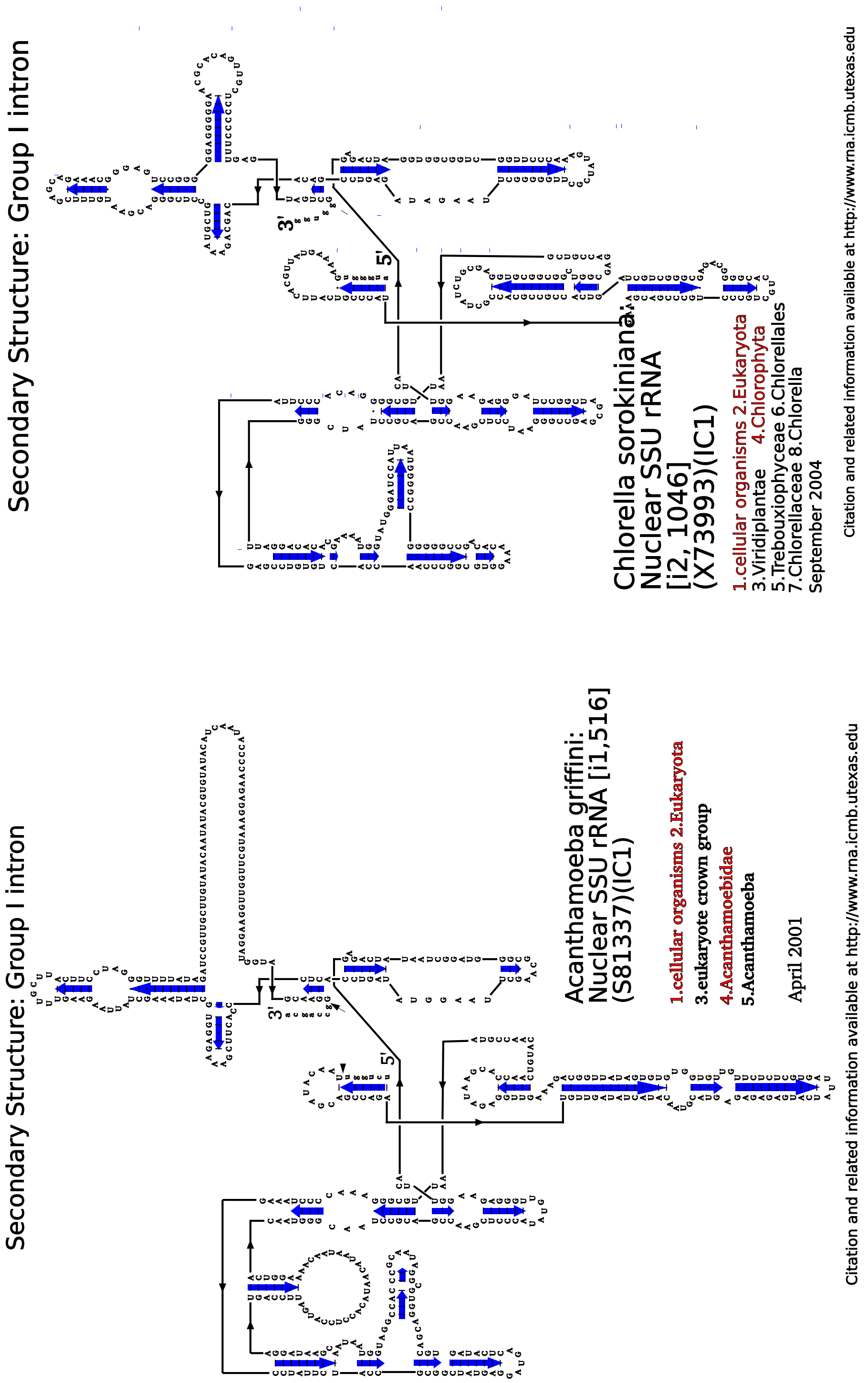


FIG. 3.26 – Deux ARN extraits de la base [15]. A gauche, l’intron de Groupe I de *Acanthamœba griffini* et à droite celui de *Chlorella sorokiniana*.

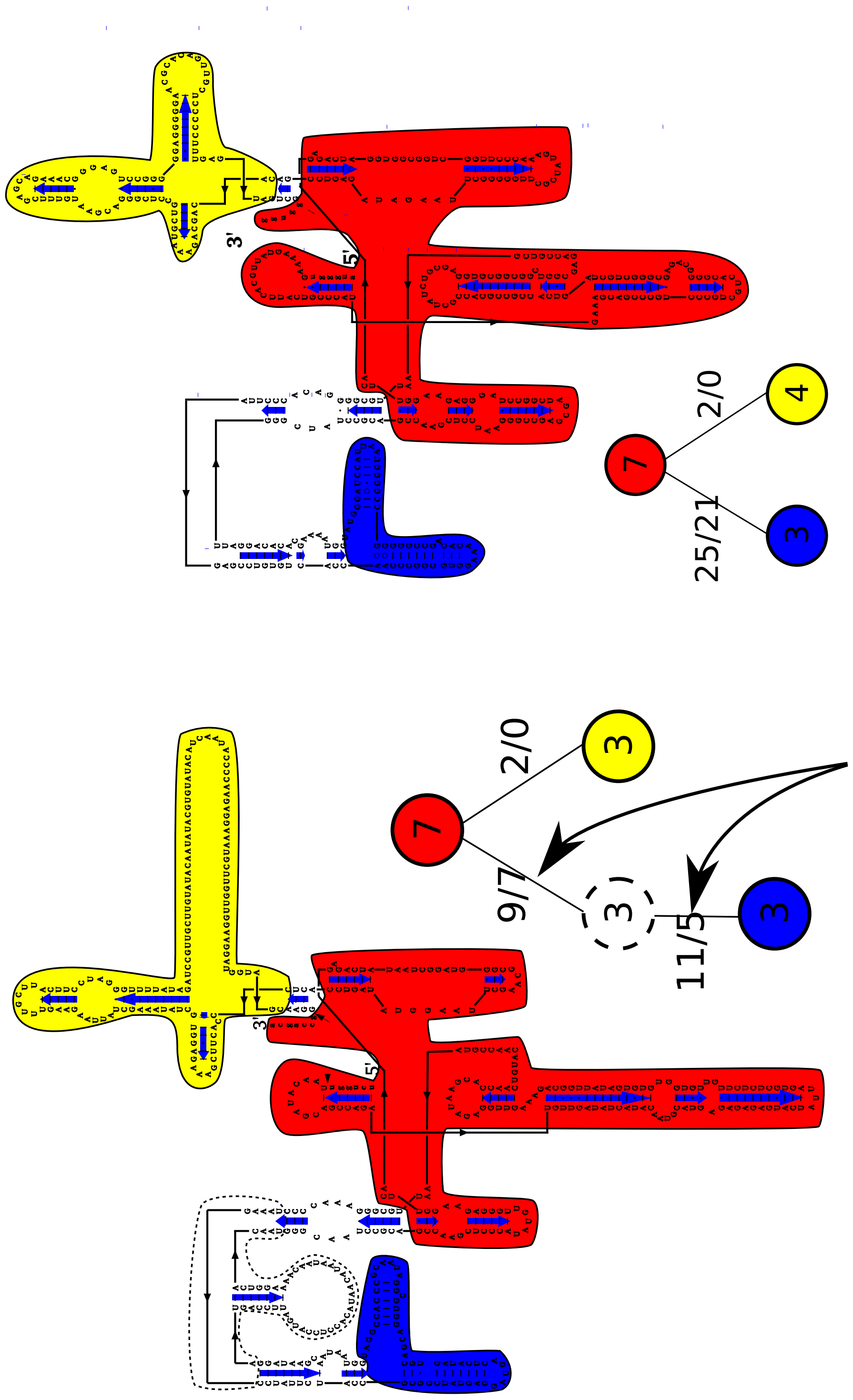


FIG. 3.27 – Résultat de la comparaison des ARN de la figure 3.26 au niveau 0.

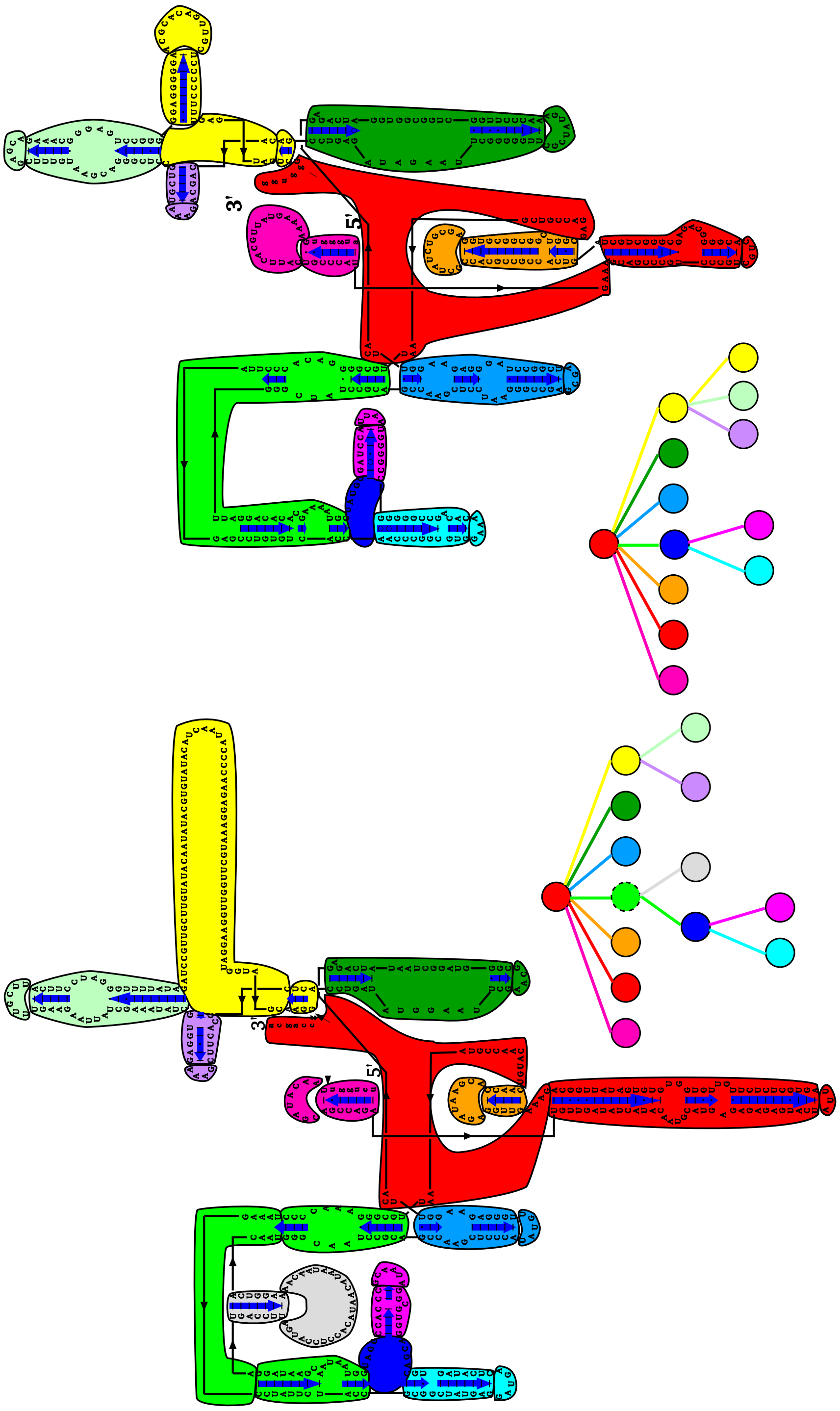


FIG. 3.28 – Résultat de la comparaison des ARN de la figure 3.26 au niveau 1.

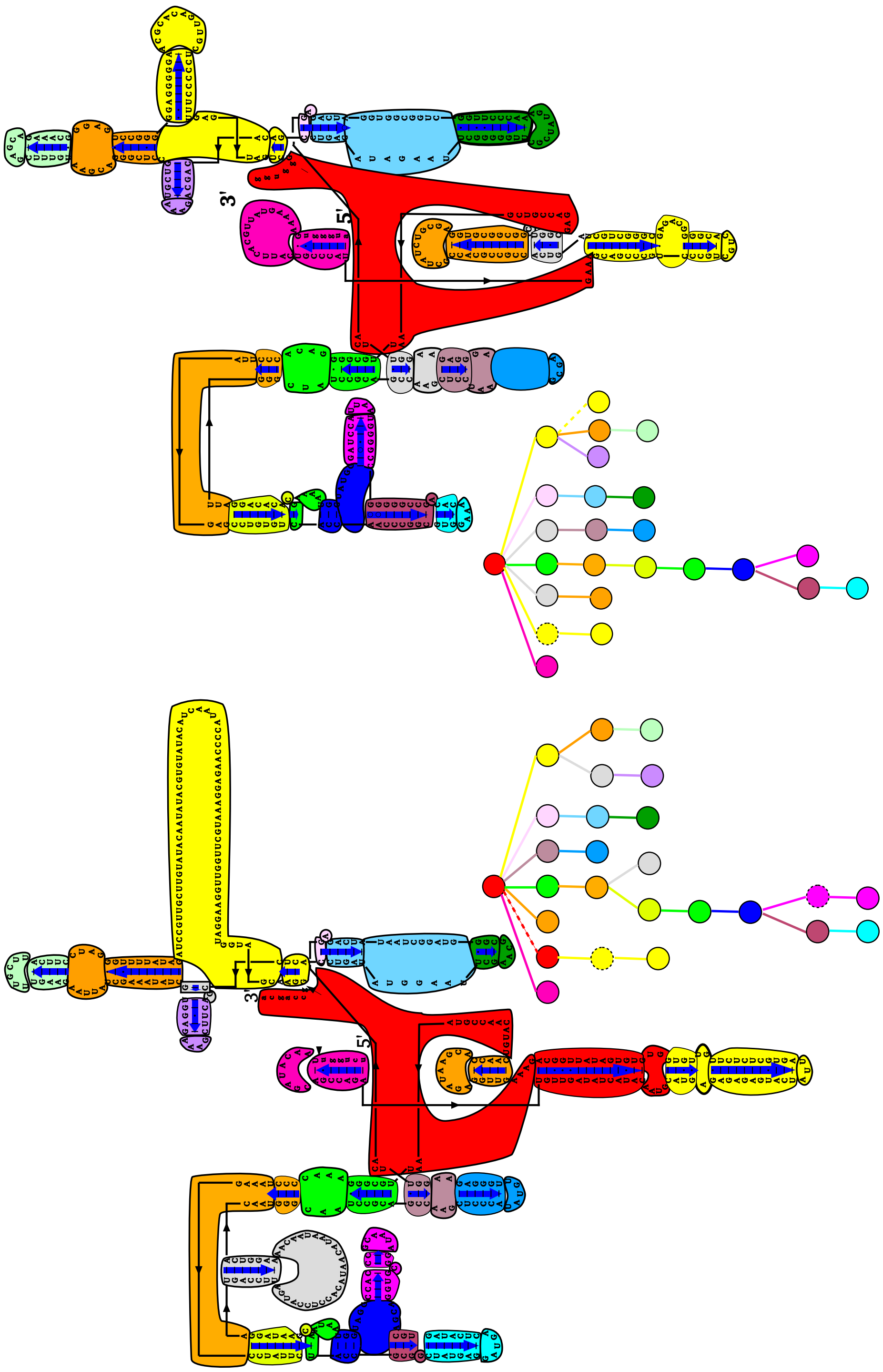


FIG. 3.29 – Résultat de la comparaison des ARN de la figure 3.26 au niveau 2.

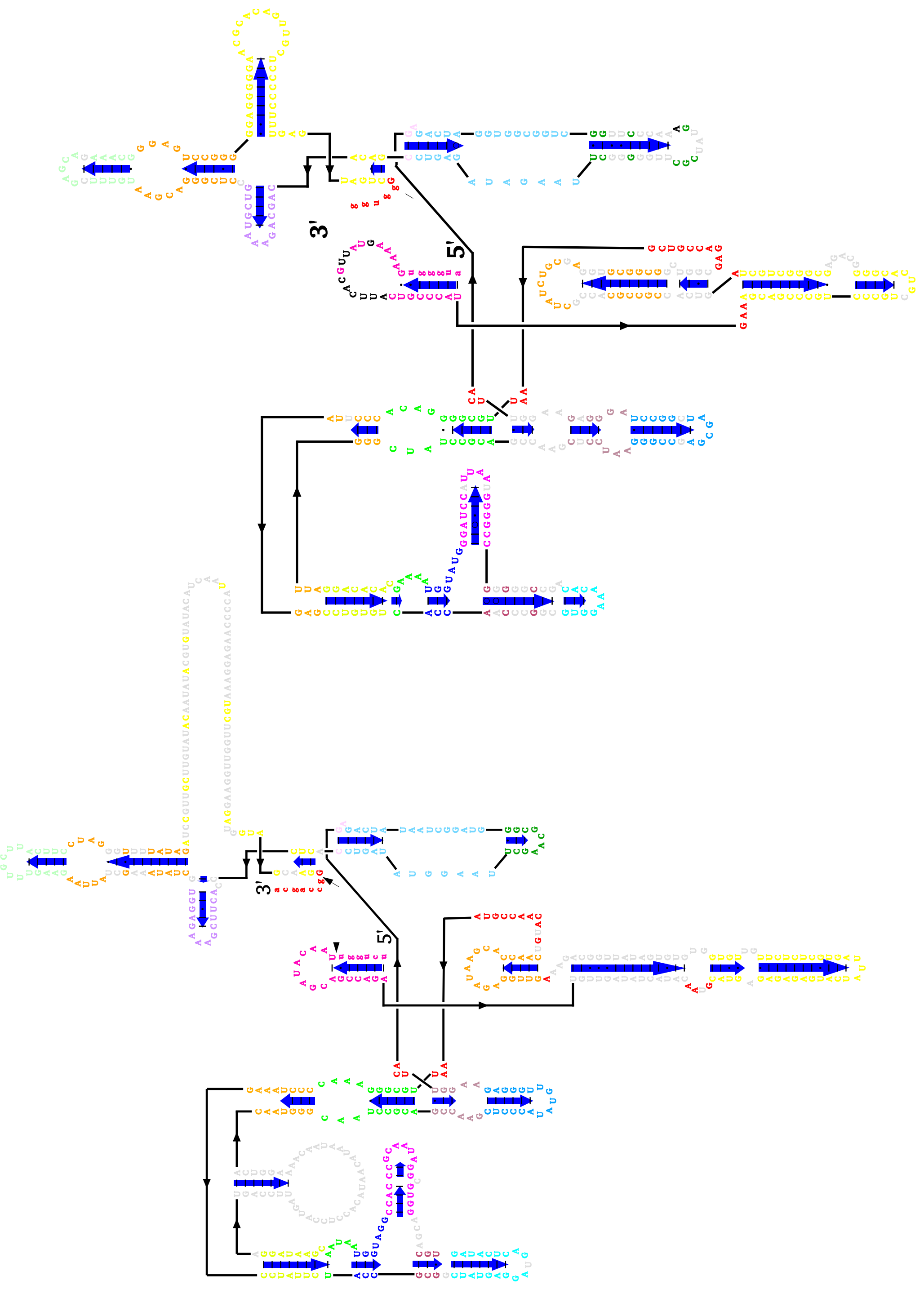


FIG. 3.30 – Résultat de la comparaison des ARN de la figure 3.26 au niveau 3.

Conclusion

Nous avons introduit un nouveau type de modélisation de structures secondaires d'ARN. Celui-ci est basé sur une représentation à différents niveaux d'abstraction de la structure, appelée RNA-MiGaL.

Il semble que vis-à-vis des approches existantes pour la modélisation des structures secondaires, à savoir les séquences annotées, les 2-intervalles ou encore les arbres, notre modèle présente un atout capital : il ne se fige pas dans une seule façon d'appréhender une structure secondaire. En effet, les autres modèles considèrent une structure secondaire uniquement soit comme un ensemble de paires de bases (séquences annotées, arbres) soit comme une suite d'hélices (2-intervalles, arbres).

Du point de vue de l'utilisation de cette structure, nous avons produit un algorithme efficace pour la comparaison de deux RNA-MiGaL.

Dans un premier temps, nous avons introduit une nouvelle distance d'édition d'arbres permettant de comparer deux structures secondaires en utilisant des arbres avec un haut niveau d'abstraction. Cette distance corrige bien les problèmes observés avec la distance d'édition classique, et bien que son calcul soit plus coûteux, elle est tout à fait praticable pour les ARN.

Dans un deuxième temps, nous avons utilisé cet algorithme pour effectuer la comparaison de deux RNA-MiGaL. Cet algorithme présente de nombreux avantages du point de vue des performances : au fur et à mesure que l'on descend les niveaux, la comparaison est partagée selon les résultats obtenus au niveau précédent. Bien que l'algorithme utilisant le partage des couleurs ne soit pas encore implanté, il y a de bonnes raisons de penser qu'il devrait être extrêmement rapide, et donc permettre de résoudre certains problèmes sur les ARN nécessitant un grand nombre de comparaisons tels que la classification ou la construction d'arbre phylogénique (voir la première partie).

Notons enfin que RNA-MiGaL a été définie dans un cadre plus formel qui est celui de MiGaL, structure à plusieurs niveaux de graphes. En outre, l'algorithme de comparaison de deux structures de type MiGaL en utilisant un algorithme de comparaison de deux graphes a été présenté.

Cependant, notre approche comporte certaines limitations.

La première limitation réside dans le fait que, parfois, la partie conservée d'une famille d'ARN ne consiste qu'en une tige boucle. Ainsi, ces ARN ne possèdent pas de structure générale commune mais uniquement une petite structure locale fortement conservée. Il est clair que notre modèle n'est pas le plus adapté pour prendre en compte ce type d'ARN mais plutôt des ARN dont la structure générale est conservée.

La deuxième limitation se situe dans la complexité et le nombre important de fonctions d'édition que l'algorithme de comparaison de deux RNA-MiGaL manipule. En effet, à chaque niveau, il nous faut définir des fonctions complexes de coût pour chaque type d'opérations d'édition et pour chaque type d'étiquettes de nœud et d'arc. Ceci est directement lié à la diversité des vues de la structure secondaire de l'ARN qui sont prises en compte dans RNA-MiGaL. À ce jour, nous ne voyons pas de solution pour réduire cette complexité ; RNA-MiGaL est une structure dense en information, ce qui implique un grand nombre de fonctions de coûts.

En ce qui concerne la comparaison de structures secondaires d'ARN, il nous reste encore beaucoup de choses à faire. De nombreuses idées d'extensions et variantes du type RNA-MiGaL sont possibles, nous allons en présenter deux. Puis nous survolerons le problème de la comparaison multiple.

Pseudo-nœuds et structure tertiaire

Comment modifier RNA-MiGaL pour qu'elle prenne en compte les pseudo-nœuds ? Initialement, cette structure devait prendre en compte de tels éléments structuraux. Ainsi, le niveau 3, représentant les paires de bases et les bases non appariées, devient un graphe. Il en est de même pour le niveau 2 codant pour les éléments de structure secondaire et les hélices. Le niveau 1, quant à lui, ne contient plus les pseudo-nœuds et est similaire au niveau 1 tel qu'il est défini actuellement, de même pour le niveau 0. Il nous est apparu que par rapport à la façon dont nous comparons les structures secondaires en utilisant RNA-MiGaL, c'est-à-dire du niveau 0 au niveau 3, l'information des pseudo-nœuds intervient alors comme un filtre. En effet, lorsque l'on compare les RNA-MiGaL au niveau 2, le premier niveau prenant en compte les pseudo-nœuds, les mises en correspondance entre les deux structures sont pour ainsi dire déjà faites, et les pseudo-nœuds ne seront mis en relation que si les deux parties qui forment ces pseudo-nœuds (puisqu'un pseudo-nœud est formé de l'appariement de deux séquences) ont été mises en relation au niveau 1.

Ainsi, bien que nous pensons prendre en compte les pseudo-nœuds prochainement, vis-à-vis de la définition actuelle de RNA-MiGaL, ils ne joueront un rôle que dans les mises en correspondance locales, mais pas dans la mise en correspondance globale qui est jouée au premier niveau. Si l'on veut que les pseudo-nœuds jouent un rôle dans le calcul de la première association, il faut qu'ils soient présents dès le premier arbre, c'est-à-dire au même niveau que les boucles multiples et les tiges, ce qui implique que le premier niveau devienne un graphe avec cycle. Ceci n'est pas sans conséquence sur l'algorithme de comparaison de deux RNA-MiGaL. En effet, une des façons de coder ce graphe est d'utiliser les séquences annotées de types *croisés* et, comme nous l'avons vu, le problème d'édition entre deux séquences de ce type est NP-Complet.

Plus généralement, il serait intéressant de pouvoir prendre en compte les interactions qui forment la structure tertiaire d'un ARN. En utilisant la représentation MiGaL, la modèlisation consisterait alors en une suite de graphes. Comme nous venons de le voir, si l'on souhaite toujours utiliser un algorithme d'édition, il est fort probable que celui-ci soit NP-Complet. Nous pouvons néanmoins espérer obtenir des résultats pour deux raisons. La première est que le premier niveau est une forte abstraction de la structure secondaire, on peut s'attendre à ce qu'il soit de faible taille en pratique. Ceci pourrait nous autoriser à utiliser un algorithme de comparaison même s'il est exponentiel en la taille des données en entrée. La deuxième raison est que notre algorithme général de comparaison de deux structures MiGaL progresse selon une stratégie de découpe du graphe en fonction de l'association calculée au niveau précédent. Ainsi les comparaisons après le niveau 0 se feront entre sous-structures de faible taille, et donc seront réalisables en pratique.

Pour finir, notons que si la comparaison de structures MiGaL s'avère réalisable en pratique, il serait intéressant d'examiner la possibilité de l'étendre à d'autres problèmes tels que la comparaison de protéines.

Comparaison multiple

Une autre problématique intéressante pour la suite de nos travaux est l'étude de la comparaison multiple de structures secondaires d'ARN. Par là, on entend la comparaison simultanée de plus de deux ARN contrairement à une des solutions couramment mise en œuvre pour la comparaison multiple qui consiste à effectuer toutes les comparaisons possibles deux à deux.

Si l'on conserve la même façon d'avancer de niveau en niveau dans la structure MiGaL, lors de la comparaison de plusieurs RNA-MiGaL, il nous faut effectuer la comparaison multiple entre des arbres enracinés ordonnés. Une solution est d'utiliser l'algorithme d'édition que nous avons présenté, cependant la complexité de celui-ci devient alors exponentielle en le nombre d'arbres en entrée. Nous pouvons dès à présent assurer qu'il ne sera pas possible de comparer plus de trois ou quatre arbres en utilisant cette méthode.

Une autre solution serait d'utiliser une structure d'indexation des arbres afin de trouver dans un premier temps les sous-structures communes à l'ensemble des arbres et de partir de ces sous-structures pour finir la comparaison. Cette démarche est similaire à celle couramment adoptée pour effectuer la comparaison multiple de séquences ; dans ce cas c'est une structure telle que l'arbre des suffixes généralisé qui est utilisée. Actuellement, nous cherchons une structure d'indexation d'arbres enracinés, ordonnés permettant de trouver en temps polynomial les sous-structures communes à un ensemble d'arbres.

Une fois ces sous-structures établies (éventuellement plusieurs solutions), il sera possible d'utiliser l'algorithme d'édition d'arbre en contraignant l'édition pour qu'elle associe ces sous-structures.

Annexe A

L'arbre des k-facteurs

Cette annexe est consacré à l'arbre des facteurs de longueur au plus k ou arbre des k-facteurs. Cet arbre, basé sur l'arbre des suffixes, indexe l'ensemble des facteurs de longueur au plus k d'une séquence. En fait, on peut voir l'arbre des k-facteurs comme un arbre des suffixes tronqué à une longueur fixe k, la longueur étant la taille des mots épellés depuis la racine. L'algorithme permettant de construire cet arbre est une version modifiée de l'algorithme d'Ukkonen pour la construction de l'arbre des suffixes. Sa complexité en temps et en mémoire est linéaire, le gain mémoire obtenu vis-à-vis de l'arbre des suffixes n'a pas été prouvé d'un point de vue théorique sauf pour de faibles valeurs de k.

Malgré cela, l'arbre des facteurs est une structure puissante qui remplace avantageusement l'arbre des suffixes dans de nombreux cas. En effet, souvent on aura à utiliser l'arbre que jusqu'à une certaine hauteur, par exemple si l'on cherche des motifs dont la longueur est inférieur à k dans un texte. Quelle que soit cette borne, il est judicieux d'utiliser l'arbre des facteurs car il est clair qu'au pire, il présentera les mêmes performance que l'arbre des suffixes.

La deuxième partie de ce travail, plus pratique, a consisté à implémenter l'arbre des k-facteurs en utilisant l'un des codages les plus performants en mémoire de l'arbre des suffixes. Pour cela, nous avons du modifier ce codage, en particulier aux feuilles, afin de prendre en compte certaines particularités de l'arbre des k-facteurs. Enfin, nous avons réalisé de nombreux tests pratiques de notre structure. Entre autres, nous avons pu constater que le temps de construction est très souvent meilleur, et rarement supérieur au temps de construction de l'arbre des suffixes. Les gains mémoire quant à eux sont très variables en fonction du type de données à indexer. Pour le cas de texte structurés comme un programme en C, nous avons un gain moyen de 23% pour $k = 20$. Dans le cas de données biologiques telles que des séquences d'ADN, le gain constaté est de 32% pour une valeur de k égale à 10. Enfin, l'arbre des facteurs présente aussi une amélioration par rapport à l'arbre des suffixes pour l'accès à l'ensemble des positions correspondantes à un motif. En effet, dans l'arbre des suffixes, il faut effectuer un parcours en profondeur d'un sous-arbre afin de trouver l'ensemble des occurrences du motif correspondant à ce sous-arbre. Dans ce cas à chaque feuille correspond une position. Dans le cas, de l'arbre des facteurs, à chaque feuille peut correspondre une liste de positions. Ainsi, pour la recherche du même motif, le sous-arbre à parcourir sera plus petit. C'est ainsi que nous avons constaté un sensible gain en temps avec l'utilisation de l'arbre des k-facteurs pour fournir l'ensemble des occurrences correspondantes à un motif.

Nous avons choisi de présenter ces travaux dans cette thèse car, bien qu'élaborés en partie durant la maîtrise, ils ont été réalisés pour une grande partie lors de la thèse. Cependant, ceux-ci n'étant pas en rapport direct avec le sujet de la thèse, nous avons préféré les disposer en section annexe. Nous tenons à remercier entre autres les utilisateurs de notre implémentation de l'arbre des k-facteurs :

- Alexandra Carvalho [2], de l'Instituto Superior Técnico, Lisbonne.
- Patricia Thébault [84], du Lab. de Biométrie et Intelligence Artificielle, INRA Toulouse.
- Pierre Peterlongo, Institut Gaspard Monge, Marne-la-Vallée.

pour nous avoir aidé à en extraire les derniers *bugs*.

A.1 The at most k-deep factor tree

Résumé

Cet article présente un nouvelle structure d'indexation proche de l'arbre des suffixes. Cette structure indexe tous les facteurs de longueur au plus k d'une chaîne. La construction et la place mémoire sont linéaires en la longueur de la chaîne (comme l'arbre des suffixes). Cependant, pour des valeurs de k petites, l'arbre des facteurs présente un fort gain mémoire vis-à-vis de l'arbre des suffixes.

Mots Clefs : arbre des suffixes, arbre des facteurs, structure d'indexation.

Abstract

We present a new data structure to index strings that is very similar to a suffix tree. The new structure indexes the factors of a string whose length does not exceed k, and only those. We call such structure the *at most k-deep factor tree*, or *k-factor tree* for short. Time and space complexities for constructing the tree are linear in the length of the string. The construction is on-line. Compared to a suffix tree, the k-factor tree offers a substantial gain in terms of space complexity for small values of k, as well as a gain in time when used for enumerating all occurrences of a pattern in a text indexed by such a k-factor tree.

Keywords : suffix tree, at most k-deep factor tree, string index, pattern matching

A.1.1 Introduction

The suffix tree is one of the best known structures in text algorithms. It is used in many fields such as string matching [19] [61], data compression [55][54], computational biology [12] [5] [67] [75] etc. A suffix tree indexes all suffixes of a string and can be constructed in time linear in the length of the string. Furthermore, the size and depth of the tree are also linear in the length of the string. The constant however may be big. One of the original implementations of a suffix tree [68] thus requires 28 bits per input char. Numerous efforts have been made over the years to reduce this space. In particular, S. Kurtz proposed in [52] various implementations which require

20 bits per input char in the worst case and 10 on average. The same author with co-workers suggested in another paper [32] a variant of the suffix tree, called the *lazy suffix tree*, which can be built on the fly as the need arises.

In many applications, the length of the patterns to be searched in the suffix tree can be limited. Typically, this concerns applications to motif extraction in biological sequences where the motifs of interest are often short or are composed of short parts at constrained distances from one another [86], applications to data compression using a sliding window [27], [69], etc.

There is therefore often no need to have the whole suffix tree. This is our motivation for proposing a new structure, called the *at most k-deep factor tree*, or, for short, the *k-factor tree*. The k-factor tree retains the linear properties of the suffix tree. Indeed, its construction follows closely that of the suffix tree as proposed by Ukkonen [85]. For small values of k, there is furthermore a gain in time and space that may be important. The gain in time concerns both the construction itself and the use of the tree, for instance, to do pattern matching.

Since our construction of the k-factor tree is strongly based on Ukkonen's suffix tree construction, we start by recalling Ukkonen's algorithm in Section A.1.2. We then introduce our own algorithm for the k-factor tree construction. The k-factor tree has been implemented using the coding proposed by Kurtz in [52] and the results of the experiments are discussed in Section A.1.5.

A.1.2 Suffix trees

Before recalling Ukkonen's suffix tree construction, we start by introducing some notations.

Notations

In what follows, s will denote a string over a finite alphabet Σ whose cardinality is $|\Sigma|$. The length of s is $|s|$. A string s is therefore an element of the free monoid Σ^*. The i^{th} letter of s is denoted by s_i and $s_{i...j}$ represents the factor $s_i s_{i+1} \dots s_j$ (also named substring or subword). The suffix of s starting at position i is the factor $s_{i...|s|}$. If s ends with a letter which is not in Σ, we say that s has an *ending symbol*, denoted by \$. Given two strings u and v, we note $u.v$ the concatenation of u with v.

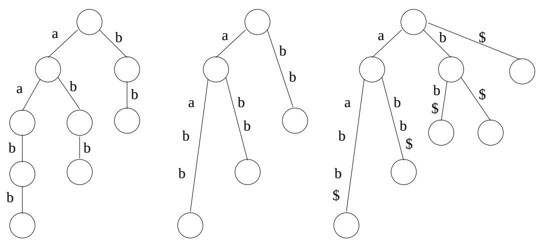


FIG. A.1 – Suffix trie of $s = aabb$ (left), implicit suffix tree of s (center) and suffix tree of $aabb\$$ (right).

Given a tree T with root R whose edges are labelled by elements of Σ^* and a node N of T, $path(N)$ is the string corresponding to the concatenation of the labels encountered along the path from R to N. The depth of N, denoted by $depth(N)$, is the number of nodes between R and N, N included ($depth(R) = 0$).

Definition of a suffix tree

The suffix trie of s is a tree with edges labelled by elements of Σ. For each factor of s, there exists a node N such that $path(N)$ is equal to that factor. If the factor is a suffix of s, some of these nodes may be leaves. If s has an ending symbol, all nodes N whose path from R spells a suffix of s are leaves.

The *implicit* suffix tree of s is a tree with edges labelled by non-empty elements of Σ^*. The tree is a compressed version of the suffix trie. Each internal node N of the suffix trie that has only one son is deleted and its two adjacent edges are replaced by an edge that goes from N's father to N's son. The label of the new edge is equal to the concatenation of the label of the edge going from N's father to N and of the label of the edge from N to its son. This tree is called implicit because not all suffixes of s lead to a leaf. The true suffix tree is obtained when an ending symbol \$ is added at the end of s.

Suffix tree construction

A first suffix tree construction algorithm of complexity linear in the length of the string was presented by Weiner in 1973 [93]. McCreight introduced a similar but more performing algorithm in 1976 [68]. In 1995, Ukkonen publi-

SuffixTree(s)
Tree T
For i from 1 to $|s|$
    For j from 1 to i
        Add(T,$s_{j \ldots i}$)

FIG. A.2 – Naive algorithm

shed an on-line construction of the suffix tree using a very different approach [85]. Later, Giegerich and Kurtz [31] showed how all three constructions are in fact similar.

We now briefly present Ukkonen's algorithm which is the basis for the k-factor tree construction. For more details, we refer the reader to the original paper [85]. We follow here the description of the algorithm given in [36].

A.1.3 Ukkonen's algorithm

The algorithm is divided into $|s|$ phases. The i^{th} (for $1 \leq i \leq |s|$) phase consists in the insertion of all suffixes of $s_{1 \ldots i}$ into the tree. Each phase i is then divided into j steps, one step for each insertion of a suffix of $s_{1 \ldots i}$ (see Figure A.2). This naive algorithm is clearly in $O(|s|^3)$. We show now how Ukkonen obtains a linear complexity.

Cases met when inserting a word w in the tree

Let us call v the longest prefix of w which is already in the tree. We have $w = vu$. During the insertion of w, three cases may occur :

1. u is the empty word so that w is already in the tree : there is nothing to do.
2. spelling v from the root leads to a leaf : all we need to do is append u to the label of the edge leading to this leaf.
3. the last case may be divided into two subcases :
   (a) spelling v leads to an internal node N : we attach a new leaf to this node ; the edge from N to the leaf is labelled u.

(b) spelling v leads to the middle of an edge : we have to cut this edge at the position reached and insert a new node N ; to the node N thus created, we then attach a new leaf ; the edge from N to the leaf is labelled u.

Suffix links

Suffix links are fundamental elements in all linear time suffix tree construction algorithms. We shall see later that they enable to reduce the cost of string insertion during the algorithm.

A suffix link is an oriented link between two internal nodes in the suffix tree. Given a node N, its suffix link points to the node, denoted by $S_l(N)$, such that $path(S_l(N))$ is equal to $path(N)_{2\ldots|path(N)|}$ (that is, to $path(N)$ without the first letter). In other words, $path(S_l(N))$ is the longest proper suffix of $path(N)$. All internal nodes have a suffix link.

Suffix links are added to the tree during construction as follows : let us suppose that, during the insertion of string $s_{j\ldots i}$, a node N is created to which a new leaf is attached (case 3b). We know that $s_{j\ldots i-1}$ is in the tree because we inserted it during the previous phase. We therefore have that $path(N)$ is equal to $s_{j\ldots i-1}$. By construction, when a string is in the tree at the end of a phase, all suffixes of this string are also in the tree. The factor $s_{j+1\ldots i-1}$ is thus already in the tree. Two cases may then happen : the path labelled $s_{j+1\ldots i-1}$ ends in a node, let us denote it by N', or it ends inside an edge. In the second case, the insertion of factor $s_{j+1\ldots i}$ leads to the creation of a node, that we also denote by N'. In both cases, the suffix link of N points to N'.

Fast insertion

Let N be the last node reached during the insertion of $s_{j\ldots i}$ that has a suffix link. We have that $s_{j\ldots i}$ is equal to $path(N).w.\sigma$, where w can be empty and $\sigma \in \Sigma$. If $S_l(N)$ is the root, then the insertion of $s_{j+1\ldots i}$ is done naively. If $S_l(N)$ is not the root, in order to insert $s_{j+1\ldots i}$, we just have to follow w from $S_l(N)$ and add σ if necessary.

During the insertion described above, we found w in the tree by successively comparing each letter of w. In fact, we can avoid such comparisons in the following way. At each node met during the insertion, we just need to know which edge to take. Once this edge is identified, we can go directly to the node it points to and, as a consequence, move in w by possibly more than one letter.

The time spent during an insertion is essentially composed of the time needed to calculate the length of an edge plus the number of nodes traversed during the insertion. The next section shows how the length of an edge can be computed in constant time and how, simultaneously, the space required by the tree may be reduced. In the proof of the time linearity of the algorithm, we show that the overall number of nodes traversed is then $O(|s|)$.

Edge coding

The labels of the edges can be coded by a pair of integers which denote, respectively, the start and end positions in s of the string labelling the edge. This allows to obtain a linear space occupancy for the tree as is detailed below. It also leads to an improvement in the time complexity of the suffix tree construction as leaves can then be automatically extended.

Automatic leaf extension and condition for ending a phase

Let us call *end position of an edge* the second value in the label of an edge, which corresponds to the end position of the string labelling that edge. Suppose a leaf is created during the insertion of $s_{j...i}$. The end position of the edge pointing to this leaf is i. In the next phase, the insertion of $s_{j...i+1}$ will lead to case 2, and the end position of the edge will now become $i+1$, and so on for all the following phases. When a leaf L with edge E leading to it is created, we can then set the end position of E to a global variable whose value is the number of the phase. This will exempt us from having to make the insertions corresponding to the extension of all edges leading to a leaf. Furthermore, if the insertion of $s_{j...i}$ requires the creation of a leaf, then all strings $s_{\ell...i}$ for ℓ from 1 to j lead to a leaf in the tree. During the next phase, if the insertion of $s_{j+1...i}$ has not created a leaf, we can therefore start the insertion of the suffixes of $s_{1...i+1}$ from the insertion of $s_{j+1...i+1}$ (that is, from step $j+1$) because all the longer suffixes are implicitly inserted. Each phase then starts from the last leaf created.

In the same way, if, during the insertion of $s_{j...i}$, we end up in case 1, we do not have to insert $s_{\ell...i}$ for ℓ from $j+1$ to i because if $s_{j...i}$ is already in the tree, then all suffixes of $s_{j...i}$ are also in the tree.

```
AddString(node,s,start,end)
endJump ← false
while(not endJump) and ((end − start) not equal to 0) do
      set child to the child of node that start with the letter s_start
      if (end − start) is greater or equal to length(node, child)
            start ← start + length(node, child)
            node ← child
      else
            endJump ← true
done
if (end − start) is equal to 0 and node has not a child for letter s_end
      add a child to node with edge label start equal to end
e ←the label of the edge between node and child
if e_{end−start+1} not equal to s_end
      split e at position end − start
      add a leaf with start position equal to end to the new node
done.
```

FIG. A.3 – The function *AddString* achieves the fast insertion.

In what follows, all non implicit insertions such as those described above are called *explicit*.

Algorithm

Figure A.4 gives the complete algorithm for Ukkonen's construction. We see that index j does not appear anymore. As just showed, the beginning of each phase is deduced from the end of the previous phase, and the end of a phase is deduced from the result of a current insertion.

Moreover, we may observe that the algorithm of Figure A.4 constructs the implicit suffix tree of s. To obtain the suffix tree of s, an ending symbol \$ must be appended to s.

```
Suffix_Tree(s)
Add to R a leaf L with edge label s_1
lastLeaf ← L
For i from 2 to |s|
    endPhase ← false
    do
        forward ← length(Father(lastLeaf), lastLeaf) − 1
        if S_l(Father(lastLeaf)) is undefined and Father(lastLeaf)! = R
            forward ← forward+
                length(Father(Father(lastLeaf)), Father(lastLeaf))
            if Father(Father(lastLeaf)) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(Father(lastLeaf))),s,i − forward,i)
        else
            if Father(lastLeaf) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(lastLeaf)),s,i − forward,i)
        if a node was created during the previous step
            set suffix link of this node to the last node reached
                during the insertion
        if a leaf was created in the call to AddString
            set lastLeaf to this leaf
        if a node was not created in the call to AddString
            endPhase ← true
    while(not endPhase)
end for
return R
```

FIG. A.4 – Ukkonen's algorithm : construction of the suffix tree for s ; R is the root of the tree. The function *length* returns the size of the edge label between two nodes.

Complexity

We now analyse the time and space complexities of the suffix tree construction. These will be the same for the k-factor tree.

Time complexity

Ukkonen's assertion [85] that his suffix tree construction algorithm is linear in the length of the string rests upon the following lemma [36] :

Lemme 1. *Let N be a node. We have that $depth(N) \leq depth(S_l(N)) + 1$.*

The implicit leaf extensions inside a phase take constant time, so the implicit insertions made over the complete algorithm take $O(|s|)$ time. The time taken by the explicit insertion of a string in the tree is directly related to the number of nodes traversed after the jump along the suffix link since the ascent from a node to its father and the jump along the suffix link can be done in constant time. Instead of counting the number of nodes traversed after the jump along the suffix link for each separate insertion, we can upper bound the number of nodes traversed during the whole construction. Let j be the index of the extension currently considered. Index j remains unchanged between two successives phases, and it never decreases. We may observe that we do at most $2|s|$ explicit insertions because we have $|s|$ phases and j is at most $|s|$. During an explicit insertion, the depth of the node currently considered is first decreased by at most two, one to reach the father of a leaf and one when jumping along the suffix link, and then increased at each skip down the tree. Since the maximal depth of the tree is $|s|$ and we do at most $2|s|$ insertions, the total number of nodes considered is $O(|s|)$. If we suppose that the access to a child of a node is done in $O(1)$ time, then the total time complexity of the suffix tree construction is in $O(|s|)$.

Space complexity

Let T be the suffix tree of s. The root of T has exactly $|\Sigma| + 1$ children. One of them corresponds to a leaf (the string label of the edge leading to it is \$). The worst case for the space complexity will be obtained if each internal node has exactly two children and each edge is labelled with a single letter (*i.e.* its length is one).

In that case, the total number of nodes whose depth is k in the tree is :

$$2^{k-1}(|\Sigma| + 1) - 2^k + 2 \texttt{ that is } 2^{k-1}(|\Sigma| - 1) + 2.$$

The number of leaves in T is $|s|+1$ and we can then deduce the depth d of T in the worst case which is :

$$2^{d-1}(|\Sigma|-1)+2+d-1=|s|+1 \texttt{ that is } 2^{d-1}(|\Sigma|-1)+d=|s|$$

An upper bound for d is :

$$d_{max}=\log_2\frac{2|s|}{|\Sigma|-1}.$$

We can now compute the total number of nodes in T which is :

$$1+\sum_{k=1}^{k=d_{max}}[2^{k-1}(|\Sigma|-1)+2]$$

that is :

$$2|s|+2d_{max}-|\Sigma|+2.$$

A.1.4 Factor trees

We now present the k-factor tree construction. A naive approach consists in building the suffix tree and then pruning the tree in such a way that for each node N of the remaining tree, $|path(N)|\leq k$. Clearly, this approach does not improve the space complexity because it requires first constructing the suffix tree.

Recall that one of the main ideas behind Ukkonen's suffix tree construction is the automatic leaf extension allowed by the use of a global variable. We must be able to preserve this idea in the k-factor tree construction if we wish to keep the linear time complexity. Let us call *length of a leaf* L, the length of the string labelling the path from R to L. Preserving Ukkonen's idea requires finding a way of stopping the extensions when a leaf reaches the length k.

Construction

The k-factor tree contruction is based on Ukkonen's algorithm. It is divided into two parts :

- building the implicit suffix tree for $s_{1\ldots k-1}$;
- adding to the tree all the suffixes of $s_{i-k+1\ldots i}$ for i from k to $|s|$.

We now detail these two parts. Henceforward, we call *end position of a leaf* what corresponds to the end position of the edge leading to that leaf.

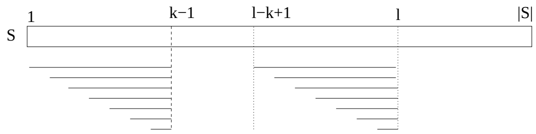


FIG. A.5 – k-Factor tree construction algorithm. The horizontal lines indicate the substrings of s which are inserted into the tree during phase i. On the left, the substrings inserted for $i = k - 1$ are shown and, on the right, the substrings of length at most k inserted for $i = \ell$ (for $\ell \geq k$).

First part : construction of the implicit tree of the suffixes of $s_{1 \ldots k-1}$

The first part is done following Ukkonen's algorithm in a straightforward way. The only difference is that, when a leaf is created, it is now added to a queue called $queue_{leaf}$.

Second part : construction of the implicit tree of the suffixes of $s_{i-k+1 \ldots i}$ for i from k to $|s|$

For each string corresponding to a suffix of $s_{i-k+1 \ldots i}$ for i from k to $|s|$ that needs to be inserted in the tree, we proceed essentially as in Ukkonen, that is, we go up to the last node which has a suffix link, jump along the suffix link and then go down in the tree. As for the first part, we add each newly created leaf to $queue_{leaf}$. We now also have to remove an element from the head of the queue at the end of each phase. Furthermore, there may be another subtle but important difference in relation to Ukkonen's algorithm depending on the state of $queue_{leaf}$ at the beginning of a phase.

Indeed, in Ukkonen's suffix tree construction, a phase starts from the last leaf created. At the start of a phase i in our algorithm, two cases may happen :

– there is a leaf L at the end of $queue_{leaf}$;
– $queue_{leaf}$ is empty.

In the first case, we proceed with phase i straightforwardly from leaf L, which corresponds to the last leaf created.

In the case where $queue_{leaf}$ is empty, we do not proceed as in Ukkonen from the last leaf that was created in the algorithm, but from the last position reached in the tree during the previous phase. This position may be in the middle of an edge. The reason for this is the following. Let i be the phase for which $queue_{leaf}$ is empty when the phase begins and let L be the last leaf created before phase i. The leaf L corresponds therefore to $s_{i-k\dots i-1}$. At phase i, we would start from leaf L, jump up to L's father, along L's father suffix link and then try to insert $s_{i-k+1\dots i}$. If $s_{i-k+1\dots i}$ were already in the tree, phase i would be terminated and phase $i+1$ would start again from the leaf L. The first string needing to be inserted would be $s_{i-k+2\dots i+1}$ and we would have to follow two suffix links from L's father in order to reach $s_{i-k+2\dots i-1}$ and check for the existence of $s_i s_{i+1}$ at the end of $s_{i-k+2\dots i-1}$. If $s_{i-k+2\dots i+1}$ were already in the tree, phase $i+1$ would also be stopped, and phase $i+2$ would again start from the leaf L and require now three suffix link jumps before trying for the insertion of the first suffix of the phase. If we proceeded in this way, we could no longer guarantee a linear complexity for the algorithm. This will not be the case if, instead, we start each phase for which $queue_{leaf}$ is empty at the last position reached in the tree during the previous phase. Inserting $s_{i-k+1\dots i}$ implies then just checking, from such a position, for the existence of s_i, which can be done in constant time. If a leaf needs to be created, we create it, add it to $queue_{leaf}$ (it will become the head of the queue) and go to the next step of the phase. If a leaf L is reached ($path(L)$ is thus already equal to $s_{i-k+1\dots i}$), we proceed with the next step of the phase. Any leaf that needs to be created during the phase is added to the queue.

At the end of phase i, the leaf L at the head of $queue_{leaf}$ is removed. The end position of L is set to i.

Lemme 2. *Stopping the automatic extension :*
For each phase i, $k \leq i \leq |s|$, if $queue_{leaf}$ is not empty let us call L the leaf at the head of $queue_{leaf}$. The length of $path(L)$ is equal to k at the end of phase i.

Démonstration. When a leaf is created in the suffix tree, its length is always equal to the length of the last created leaf minus one. This can be observed in Ukkonen's algorithm. Indeed, if in the phase i, we start with step j and we create a leaf L, it will be because during a previous phase ℓ, the insertion of $s_{j-1\dots\ell}$ ended in the creation of a leaf L', while during phases $\ell+1\dots i-1$ there was no leaf created. At the end of phase $i-1$, leaf L' will have a length of $i-(j-1)+1$ because of the automatic extensions. We then insert $s_{j\dots i}$ which creates leaf L whose length is equal to $i-j+1$.

During the first phase, which is phase $i = k$, of the second part of the algorithm (when the suffixes of $s_{i-k+1...i}$ for i from k to $|s|$ are inserted), the leaf L at the head of $queue_{leaf}$ is the one that was created during the insertion of s_1. It was extended to $k-1$ during the first part of the algorithm. At the end of phase $i = k$, L is removed from the queue, its end position is set to i and thus its length is clearly k. If, at this point, any leaf remains in the queue, the length of the one at the head is $k-1$, and will become k at the end of the next phase $(i+1)$ when it is removed from the queue, and so on.

Suppose now that at the beginning of a phase i, $queue_{leaf}$ is empty. We start the phase by trying to insert $s_{i-k+1...i}$. If this creates a leaf, it will be put in the queue and will be at the head of it. When the phase ends, the leaf will be removed and have length k. If no leaf is created in phase i, we proceed with phase $i+1$ and apply the same reasoning. □

Algorithm

The pseudo code for the first part of the k-factor tree construction algorithm is the same as for Ukkonen (Figure A.4), we just have to add to $queue_{leaf}$ the leaf created in line 21. The pseudo code for the second part of the k-factor tree construction is given in Figure A.6.

Complexity

We now analyse the time and space complexities of the k-factor tree.

Time complexity

The time taken by the construction of the k-factor tree is linear in the length of s. The proof is very similar to the one for Ukkonen's suffix tree construction presented in section A.1.3. The first part of the algorithm requires $O(k)$ time. Lemma 1 remains true for the k-factor tree. During the second part of the algorithm, we have $|s| - k + 1$ phases. We can start by observing that all operations related to $queue_{leaf}$ are done in constant time and therefore take $O(|s| - k)$ time for the whole algorithm. As in Ukkonen, between two phases, the start index j in s of the first substring of s that must be inserted can never decrease. The complexity thus still depends only on the total number of nodes encountered during all insertions. This number is at most $2|s|$ (we have $|s|-k+1$ phases and j is at most $|s|$). As the depth currently reached in the tree remains unchanged between two phases and we

```
Factor_Tree(R, s , k , queue_leaf)
For i from k to |s|
    endPhase = false
    if queue_leaf is not empty
        set lastLeaf to the leaf at the end of queue_leaf
    else
        add s_i from last position reached during the last insertion
        if a leaf is created
            add this leaf at the end of queue_leaf
            set lastLeaf to this leaf
        else
            set lastLeaf to the leaf reached
    do
        forward ← length(Father(lastLeaf), lastLeaf) − 1
        if S_l(Father(lastLeaf)) is undefined and Father(lastLeaf)! = R
            forward ← forward+
                length(Father(Father(lastLeaf)), Father(lastLeaf))
            if Father(Father(lastLeaf)) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(Father(lastLeaf))),s,i − forward,i)
        else
            if Father(lastLeaf) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(lastLeaf)),s,i − forward,i)
        if a node was created during the previous step
            set the suffix link of this node to the last node
                reached during the insertion
        if a leaf was created in the call to AddString
            set lastLeaf to this leaf
            add this leaf at the end of queue_leaf
        if a node was not created in the call to AddString
            endPhase ← true
    while(not endPhase)
    remove the leaf at the head of queue_leaf and set its end value to i
end for
return R
```

FIG. A.6 – Second part of the algorithm for the construction of the at most k-deep factor tree of a string s.

can go up by at most two nodes at each insertion, we can go up in the tree by at most $4|s|$ nodes only during the whole algorithm. Since the depth of the tree is at most k, the total number of nodes encountered is $O(|s|)$.

If we make the assumption that the cost to reach the child of a node is constant ($|\Sigma|$ is fixed), the whole algorithm therefore runs in linear time.

Space complexity

We now compute the number of nodes in the k-factor tree in the worst case. Clearly, the worst scenario corresponds to the case where each node has $|\Sigma|$ children. The number of nodes is then $\sum_{\ell=0}^{\ell=k} |\Sigma|^{\ell}$, that is :

$$\frac{|\Sigma|^{k+1} - 1}{|\Sigma| - 1}.$$

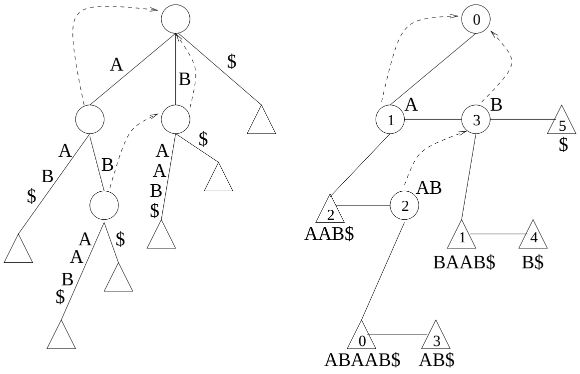
The total number of nodes in the worst case is therefore :

$$min(\frac{|\Sigma|^{k+1} - 1}{|\Sigma| - 1}, 2|s| + 2\log_2(\frac{2(|s| - 1)}{|\Sigma| - 1}) - |\Sigma| + 2).$$

We can thus guarantee a gain in memory when k is less than $\log_{|\Sigma|}(2(|\Sigma| - 1)|s|) - 1$. This result has been confirmed in practice. Experiments are presented in the next section.

A.1.5 Coding and experiments

The construction algorithm we have just presented may be used with any currently existing coding of the suffix tree. We chose to use it with the coding adopted by S. Kurtz in [52] for building suffix trees. This employs an "Improved Linked List Implementation" and is called the *illi* coding. The choice is motivated by the fact that, to the best of our knowledge, this is the best implementation in practice which is currently available. We start by explaining the coding techniques adopted by S. Kurtz then describe the modifications or extensions we had to do to it in order to adapt the coding to the construction of k-deep factor tree. Basically, the coding must be changed so that it can efficiently handle the fact that a leaf in the factor tree may now store more than one position. We end by presenting some experimental results.



Node	head(N)	|path(N)|	first child	right sibling	suffix link
0	0	0	1	--	--
1	2	1	2	3	0
2	3	2	0	--	3
3	4	1	1	5	0

Leaf	heaf(F)	|path(F)|	right sibling
0	0	6	3
1	1	5	4
2	2	4	2
3	3	3	--
4	4	2	--
5	5	1	--

FIG. A.7 – The suffix tree of $s = ABAAB\$$, the right tree shows the suffix tree as it is coded by *illi*.

Kurtz's coding

We briefly explain the *illi* coding. For further details, the reader is referred to [52].

The internal nodes and the leaves of the suffix tree are coded using two different tables. In both tables, the index of a node corresponds to the order in which the nodes are created during construction and thus put in the right table. In particular, the index of a leaf corresponds to the position in s of the suffix that is spelled by the path from the root to the leaf. In all cases, the children of a node are stored in a linked list (first-son, right-sibling). The coding for a node contains the following information :

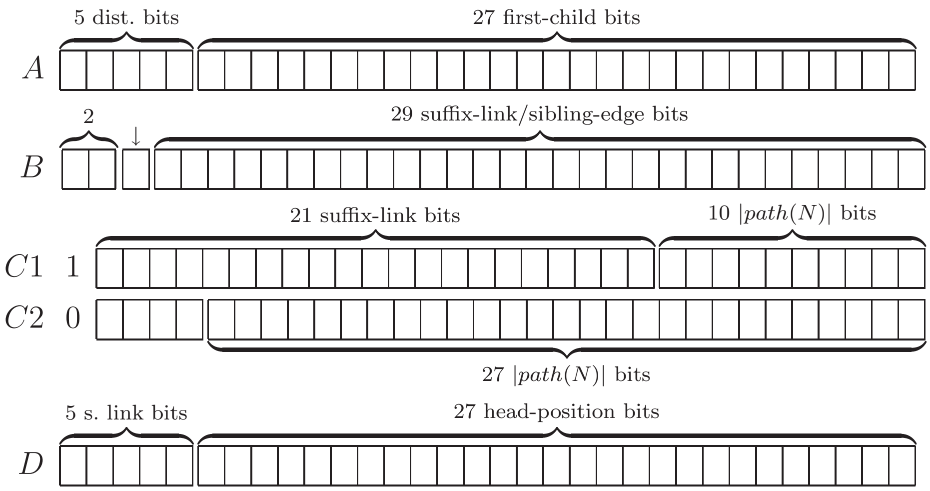
- The node is an internal node N :
  - start position of the first occurrence of $path(N)$ in s that required the creation of a node (this corresponds to the first occurrence of $path(N)$ in s that is followed by a letter different from the one following all previous occurrences ; possibly there is only one). Such position will be denoted by $head(N)$;
  - $|path(N)|$;
  - the first child of N ;
  - the right sibling of N ;
  - the suffix link of N.
- The node is a leaf F :
  - the right sibling of F.

Observe that this information is enough since the values of $head(F)$ and $|path(F)|$ for a leaf F may be deduced directly from the index of the leaf in the table. Indeed, $head(F)$ is that index (of creation of the leaf F) and $|path(F)| = |s| - head(F)$.

Figure A.7 presents a suffix tree coded according to *illi*. One of the main characteristics of the coding comes from the distinction between two types of nodes : the so-called *Small* and *Large* nodes. The idea is that nodes are created in batches, one batch (of possibly more than one node) inside each phase of the construction algorithm. During phase i (lines 12-32 in Figure A.6), if z nodes are created, say $N_1, \ldots, N_z$, then for each node N_i, $1 \leq i < z$, we have that N_{i+1} is the node to which points the suffix link of N_i (line 26), $|path(N_i)| = |path(N_z)| + (z - i)$ and $head(N_i) = head(N_z) - (z - i)$. The nodes N_i for $1 \leq i < z$ are the so-called *Small Nodes* and the node N_z is the so-called *Large Nodes*. The coding of a *Small Node* will also include a field indicating the distance (*i.e.*, the number of nodes) until the *Large Node* with which it is associated. This distance is equal to $z - i$.

The coding is detailed now. As indicated in [52], the index of a leaf may be stored on 27 bits, that of an internal node on 28 bits. To code the index of any node, we thus need 29 bits, the first one indicating if the next 28 correspond to the index of a leaf or of an internal node.

A leaf F occupies 32 bits coded according to the model B shown below. The third bit ($\downarrow$) indicates whether the sibling of F is *nil*. If this is the case, it is flagged and the next 29 bits code for the suffix link of the father (as explained below), otherwise the next 29 bits code for the sibling of F. We shall come back to the use of the first 2 bits later.



The coding of a *Small Node* is done using $2 * 32$ bits (models A and B above) and the coding of a *Large Node* on 4*32 bits (models A, B, C and D above). We denote by N the coded node, and by i its index in the table of nodes. The first 5 bits of A are zero if N is a *Large Node* and, if N is a *Small Node*, are equal to the *distance* until the *Large Node* with which N is associated. In the case where $distance \geq 32$, a "dummy" *Large Node* is inserted at the index $i + 32$. The first child of N is stored on the last 27 bits of A and the first 2 bits of B. The remaining 30 bits code for the right sibling. If N is a *Small Node*, its coding is finished. Indeed, the suffix link, $head(N)$ and $|path(N)|$ may all be calculated from the *Large Node* whose value is $I_N + distance(N)$ as described above. If N is a *Large Node*, it has 2*32 more bits. If $|path(N)|$ can be coded on 10 bits (*i.e.* its value is strictly less than 1024), then the next 4 bytes are coded according to scheme $C1$: the first bit is flagged, the next 21, the first 5 bits of D and the 2 free bits of the leaf whose value is $head(N)$ are used to code the suffix link of N. If $|path(N)| \geq 1024$, then the next 4 bytes are coded according to scheme $C2$. The 27 last bits code for $|path(N)|$. The suffix link of N is then coded in the rightmost child (coding of B) whose field "right sibling" is zero. Finally, the last 27 bits of D code for $head(N)$.

Coding for the factor tree construction algorithm

Let us examine now how we must change the *illi* coding in order to use it in the implementation of our factor tree construction algorithm. As observed

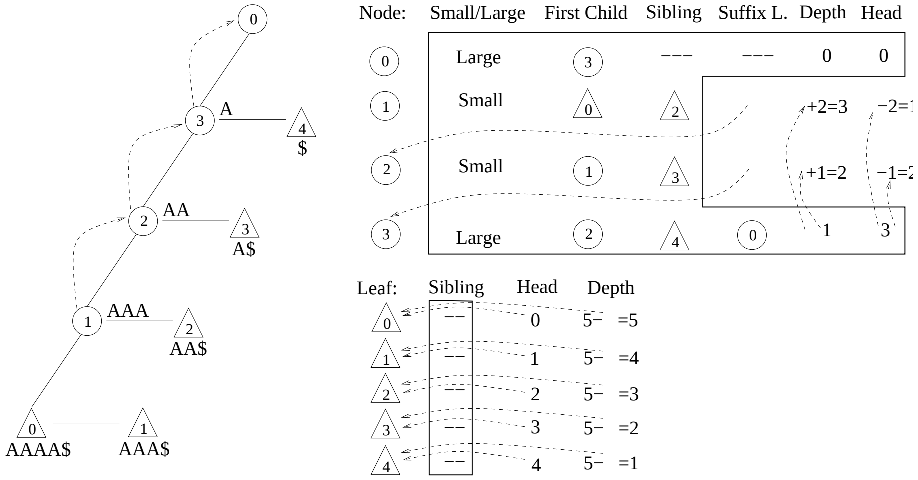


FIG. A.8 – An example of a best case for the *illi* coding, $s = AAAA\$$. In the boxes, what the data that are really coded.

previously, the values $head(F)$ and $|path(F)|$ of a leaf F can be deduced directly from the index of the leaf : the leaf F whose index is i is such that $head(F) = i$ and $|path(F)| = |s| - i$. This is not true anymore in the case of a k-deep factor tree. Indeed, since some leaves are cut, a shift is introduced in the order in which a leaf is created and then inserted in the table for the leaves. We can clearly identify the moment when a leaf is cut and a shift is thus introduced. This happens at line 11 of the algorithm given in Figure A.6. An easy way out would be at each such point to create and insert in the table a dummy leaf as we could then again deduce the values of $head(F)$ and $|path(F)|$ from the index of F. This solution is however not satisfying because memory space is unecessarily lost and, furthermore, we cannot produce the list of all the occurrences associated with a leaf.

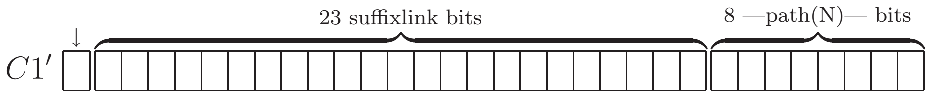
Another way of solving the problem consists in introducing a coding of the occurrences of a leaf in the form of a linked list. At line 11 in the algorithm, if a leaf F is reached at step p, a new cell is created in the table for all the leaves whose index in the table is $p - k$. We then store in the cell the index of F. In the tree, the link to F (field first child of the father or sibling of the left brother) is replaced by a link to the leaf whose index is $p - k$.

This new procedure presents two problems :

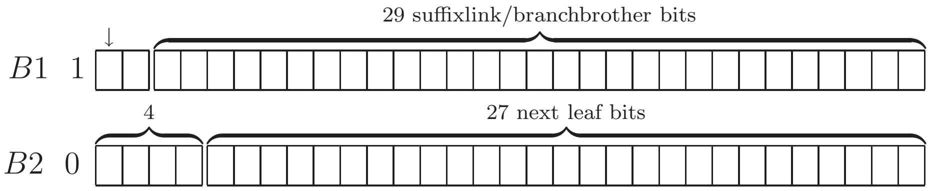
– when one traverses a list of occurrences, an additional bit is required

to indicate that we have reached the end of the list, but, as we have seen above, all the bits of the field have already been used ;
- to obtain the information "right sibling/suffix link of the father", we need to examine all the occurrences of the leaf, which implies that the complexity of the construction algorithm is no longer in $O(|s|)$.

In order to address these problems, the coding of a node has to be modified. In particular, we need to "free" bits in the coding scheme of a leaf. We thus modify the coding of the nodes in the following way. The suffix link of a node is still either to a node which exists already, or to the next node which will be created just after. In the case where the suffix link is to the next node, we have a *Small Node* and the index of the suffix link is not coded but is deduced from the node. In the other case, we have a *Large Node*. Let us denote by i the index of N in the table of the nodes. If $i \leq 2^{26}$, we know that the suffix link can be coded using 26 bits and therefore, we employ the same coding as above without using the 2 bits of the leaf whose index is $head(N)$. If $i > 2^{26}$, we use the coding scheme $C1'$ below. If $|path(N)|$ can be coded on 8 bits (*i.e.*, its index is less than 256), then the 23 bits of $C1'$ plus the 5 bits of D are used to code the suffix link. If $|path(N)| \geq 256$, we follow the coding scheme $C2$ and the suffix link is found in the rightmost child.



We thus have 32 bits per leaf and not 30 anymore. We recall that the index of a leaf requires 27 bits. We also need 1 bit to indicate whether we have reached the end of a linked list. In the case where we are at the end of a list, we remain with 31 bits, 29 to code the value *right sibling/suffix link* and 1 to indicate whether the index of the right sibling is *nil*. If we have not reached the end of a list, then of the 31 remaining bits, 27 are used to code the position of the next occurrence. To guarantee that we can obtain the value *right sibling/suffix link* in constant time, we use the last 4 bits to code part of the index of the last cell. In this way, in the worst case, we need to traverse the first 7 cells of the list before being able to jump directly to the last cell. To improve access time to the last cell, we optimise the code in the following way : if we are in cell c of the list, then we have already read $c * 4$ bits of the index of the last cell. If i is smaller than 2^{c*4}, then the value read is the index of the last cell (because this index is less than c). Another possible optimisation that has not yet been implemented consists in using in cell c, with c coded on j bits, $32 - 1 - j$ bits to code part of the index of the last cell. Indeed, the cell after i has a smaller index and thus can be coded on j bits. In what is shown below, $B1$ represents the model used for the last cell, otherwise it is the model $B2$ that is employed.



The algorithm used to insert a cell in the linked list is the following. Denoting by j the position of the next occurrence (last considered cell in the table for the leaves), by i the index of the leaf to which an occurrence is added and by e the index of the leaf at the end of the list whose first cell is i, we do the following :

- while looking for e, we :
  - check whether the value e has already been coded in the first cells of the list ;
  - determine the index l of the last cell reached before e.
- if e has already been coded, we insert cell j after cell l and i stays at the head of the list ;
- otherwise, we place j at the head of the list. Its coding follows model $B2$. We then need to update the pointers to i in the tree so that they point to j.

An example of the procedure for inserting a cell is given in Figure A.9.

Experimental results

Suffix and factor tree construction time and space occupancy

Our first test uses 43 files from the Canterbury Corpus [4] and the Calgary Corpus [9]. All tests have been done on a PC with a single $1GHz$ processor with $4Go$ of RAM memory under linux Debian/Gnu. Whenever possible, we added a termination symbol at the end of the file. The table presented below contains the following information (in order) :

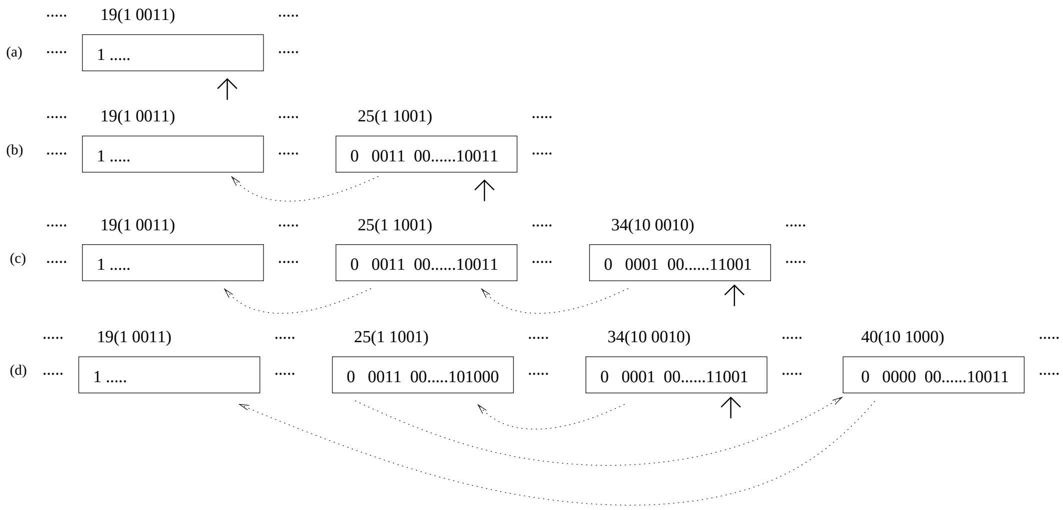


FIG. A.9 – Insertion in a list of occurrences. (a) The list contains only one cell, 19. (b) Insertion of occurrence 25, $i = e = 19$. (c) Insertion of occurrence 34 $i = 25$, $e = 19$ and $l = 25$.(d) Insertion of occurrence 40 $i = 34$, $e = 19$, $l = 25$, 19 is already coded in the cells 34 and 25, 40 is added after 25.

- the name of the file with, in parenthesis, its type : t for a text file, b for a binary file, d for DNA and f for a file in formal language ;
- the length of the text ;
- the size of the alphabet used in the file ;
- the results obtained with a suffix tree using the *illi* coding :
  - execution time in seconds ;
  - space occupied in bytes ;
- the results obtained with a k-deep factor tree using our coding and for $k = 20$ then $k = 10$:
  - execution time in seconds ;
  - the memory gain (in %) in relation to a suffix tree with the *illi* coding ;
- the results obtained with the k-deep factor tree with our coding for a memory gain of at least 15% :
  - execution time in seconds ;
  - value of k.

The last line indicates the total size of the data, followed by the average value (time, space or gain) by column. In the case where there are 2 values in a cell, the second value corresponds to the average of the values in the column except for the files *aaa* and *alphabet*.

Let us start by commenting the space memory used by both data structures. We may observe first that the type of results obtained depends on the size of the file. Indeed, cutting the tree at a depth of 20 when the text is short (for instance, *paper5* which makes 11Kb) produces a weaker space gain than cutting it also at a depth of 20 when the text is long (for instance, *world192* which makes 2,5Mb). However, independently from the type and size of the file, the factor tree offers an often significant gain for $k = 10$ as well as for $k = 20$, except for the files *random* and *kennedy*. The text in these two files present the same characteristic : they are not structured and do not contain long repeats.

The files containing a text written in a formal language are the most structured ones and are among the ones yielding the best results. For example, a file such as *cp* which is in html format contains long repeats that are specific of the language (*e.g.* the sentence `<a href="http://www.` with 20 characters). The structure itself of formal texts favours the existence of deep nodes in the tree and therefore a reasonable cut ($10 \leq k \leq 20$) leads to very good memory gains such as 25.98% for $k = 10$. The files *aaa* and *alphabet* are equally favourable to both a suffix tree with the *illi* coding and a k-factor tree with our coding in the sense that they too contain long repeats. In fact, files such as these two represent the best cases for both approaches. They lead to the best gains for the factor tree relatively to a suffix tree.

file	length	$\lvert\Sigma\rvert$	suffix tree		k=20		k=10		15%	
			time	space	time	gain	time	gain	time	k
alice29 (t)	152090	75	0.50	9.84	0.51	1.95	0.43	16.12	0.47	10
asyoulik (t)	125180	69	0.38	9.77	0.48	1.30	0.38	9.37	0.38	8
bib (t)	111262	82	0.31	9.46	0.31	9.55	0.28	25.65	0.32	15
bible (t)	4047393	64	13.56	9.64	14.17	9.94	12.50	40.60	14.29	17
book1 (t)	768772	83	3.05	9.83	3.33	0.24	2.97	14.36	3.03	9
book2 (t)	610857	97	2.05	9.67	2.09	3.81	2.04	23.36	2.15	12
lcet10 (t)	426755	85	1.35	9.66	1.60	4.33	1.35	22.39	1.54	11
paper1 (t)	53162	96	0.14	9.82	0.17	4.96	0.14	16.12	0.14	10
paper2 (t)	82200	92	0.22	9.82	0.26	1.72	0.25	13.54	0.23	9
paper3 (t)	46527	85	0.15	9.80	0.14	0.79	0.12	9.21	0.15	8
paper4 (t)	13287	81	0.04	9.91	0.04	1.11	0.04	8.12	0.03	7
paper5 (t)	11955	92	0.03	9.80	0.04	1.24	0.03	8.44	0.04	7
paper6 (t)	38106	94	0.11	9.89	0.10	5.11	0.10	16.17	0.11	10
plrabn12 (t)	481862	82	1.75	9.74	2.00	0.34	1.96	11.59	1.88	9
world192 (t)	2473401	95	7.70	9.22	7.96	20.85	7.93	37.03	8.14	27
SUBTOTAL :	9290719		2.20	9.72	2.33	4.66	2.15	18.28	2.31	11.36
aaa (f)	100001	2	0.14	12.26	0.06	67.35	0.04	67.36	0.16	77734
alphabet (f)	100001	27	0.27	12.26	0.08	67.35	0.08	67.35	0.27	77713
cp (f)	24604	87	0.07	9.34	0.07	13.81	0.07	24.43	0.07	18
fields (f)	11151	91	0.01	9.79	0.04	10.69	0.01	25.97	0.03	15
grammar (f)	3722	77	0.02	10.14	0.02	7.83	0.01	21.93	0.01	13
news (f)	377110	99	1.49	9.54	1.43	10.00	1.44	20.16	1.56	12
progc (f)	39612	93	0.11	9.59	0.14	5.44	0.10	17.37	0.09	10
progl (f)	71647	88	0.17	10.23	0.18	19.87	0.16	34.11	0.16	27
progp (f)	49380	90	0.12	10.31	0.12	22.85	0.12	36.13	0.11	41
trans (f)	93696	100	0.23	10.50	0.20	30.73	0.19	42.96	0.25	59
xargs (f)	4228	75	0.01	9.63	0.01	1.68	0.01	10.79	0.01	8
SUBTOTAL :	875152		0.24	10.32	0.21	23.41	0.20	33.50	0.25	14150
				9.90		13.65		25.98		22.55
geo (b)	102401	256	0.85	7.49	0.87	0.17	0.89	0.40	0.60	4
kennedy (b)	1029745	256	10.06	4.64	9.21	0.00	5.99	8.20	1.57	2
obj1 (b)	21505	256	0.10	7.69	0.07	6.65	0.11	10.86	0.10	6
obj2 (b)	246815	256	1.01	9.30	1.08	14.03	1.00	26.03	1.08	18
pi (b)	1000001	11	4.13	10.13	4.43	0.00	4.39	0.00	3.20	6
pic (b)	513217	160	1.20	8.95	0.85	44.01	0.74	47.30	1.16	204
random (b)	100001	65	0.50	7.05	0.49	0.00	0.49	0.00	0.32	3
sum (b)	38241	255	0.14	8.92	0.15	14.55	0.12	21.20	0.13	18
SUBTOTAL :	3051926		2.25	8.02	2.14	9.93	1.72	14.25	1.02	32.62
TOTAL :	13369887		1.53	9.52	1.55	11.89	1.36	22.19	1.29	4490.58
						8.42		19.37		19.78

Text files produce also good memory gains. However, natural languages do not in general contain very long repeats and the gain is thus in general less strong, although there may be exceptions to this. For instance, the *bible* file where sentences like " `And God ...` " favour the presence of nodes at a depth of more than 10. The average gain of 18.28% for a height $k = 10$ indicates that the factor tree is well adapted to this type of file.

The results on binary files are less easy to predict. Indeed, the file *pic*, which codes for an image, contains long repeats due to the repetitition of identical lines and the gain obtained on it is good.

For the other files, the results are disappointing. There are two reasons for this : the alphabet of these files is often big and since the text in the files are not structured, long repeats are rare. These files present therefore some of the less good results with, nevertheless, an average gain of 14.25% for $k = 10$.

file	length	$\|\Sigma\|$	suffix tree		k=20		k=10		15%	
			time	space	time	gain	time	gain	time	k
C14	87164908	5	319.98	12.43	342.30	7.35	243.73	67.34	328.53	15
C22	34553832	5	116.28	12.29	124.82	9.83	92.30	66.20	120.99	15
J03071	66495	4	0.14	12.36	0.13	21.15	0.11	33.05	0.14	30
K02402	38059	4	0.10	12.59	0.09	0.14	0.11	3.69	0.07	8
M64239	94647	4	0.23	12.62	0.25	0.51	0.26	7.44	0.23	9
AE000111	4639221	4	14.84	12.56	16.33	1.04	9.80	59.44	14.11	12
V00636	48502	4	0.11	12.57	0.13	0.00	0.14	3.02	0.10	8
X14112	152261	4	0.44	12.55	0.48	0.88	0.38	18.79	0.39	10
TOTAL :	126757925		56.51	12.49	60.56	5.11	43.35	32.37	58.07	13.37

Let us now examine the results obtained on biological data, in this case, DNA sequences. The sequences considered were chromosomes 14 and 22 of *Homo sapiens* (retrieved from the database of the NCBI [29] and denoted by C14 and C22), and seven sequences retrieved from the EMBL database (the first column indicates their primary access number). The results on chromosomes 14 and 22 of man are encouraging given the size of the tree in memory (more than 1Go), the 15% gain of the factor tree over the suffix tree represents a real gain. Otherwise, although the gain remains important for $k = 10$, results become disappointing for $k = 20$.

Figure A.10 shows the results for different values of k on the sequence with EMBL primary access number *AE000111* which corresponds to the sequence covering the first 400 entries of the complete genome of *Escherichia coli* in the EMBL database. We see that, as expected [1] [82], $\log_4 |s|$ is a good indicator of the value of k starting from which the gain diminishes as this value is increased. The curve, that presents the same behaviour with random sequences (results not shown), thus confirms that the tree is almost complete up to a depth of $\log_{|\Sigma|} |s|$. More details may be found in [1] and [82], where the authors study the internal repeats of a text and average height of a suffix tree. Observe that for DNA, the value of $\log_4 |s|$ goes from 10 for a sequence of 1Mo to 13 for a sequence of 100Mo. However, one must remember that

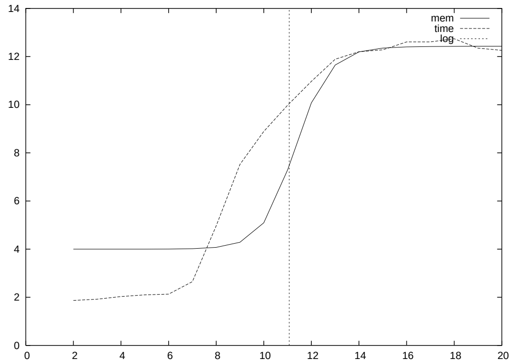


FIG. A.10 – Results obtained with the factor tree on the sequence with EMBL primary access number *AE000111*. The Y-axis indicates the space occupied by the factor tree in bytes per input char and the execution time in seconds, the X-axis plots the values at which the tree is cut. The log curve represents $\log_4 |s|$.

the nature of these sequences may change these results.

Finally, let us consider the construction time required by both structures. In the case of the factor tree, we can see that the tests added by our algorithm during construction lead to a slight increase in the execution time. However, as soon as values of k between 10 and 20 are reached, the factor tree is at least as performing as the suffix tree.

Execution time for pattern matching using a suffix or factor tree

We now present the results concerning the time required to do pattern matching with a suffix tree or a factor tree. More precisely, since the time for simply searching for a pattern (that is, just telling whether a motif is in a text) is the same for both trees, we rather examine the time taken in each case to enumerate all the positions of a pattern in the text. This is expected to be different because factor trees are cut at a depth k and a leaf points now to a list of occurrences instead of a single one as in the case of a suffix tree. We implemented for this a procedure that searches for a pattern and then realises a depth-first traversal of the tree (suffix or factor) until all the positions of the pattern are reached. This traversal is done in an iterative fashion using in both cases a static stack. At each position of pattern x in s, a function is called that does nothing.

Since a single search plus enumeration takes a time that is too small to be measured, for each pattern sought, we repeated the same procedure until we obtained an overall time of the order of a tenth of a second. We did this operation for 1000 patterns randomly taken among the factors of s (the same for the suffix and the factor trees).

The table below shows the results obtained. The columns indicate the following (execution times are in microseconds) :

- name of the file ;
- size of the file ;
- length of the pattern sought ;
- results obtained with the suffix tree using the *illi* coding :
  - minimum execution time of the operation ;
  - average execution time of the operation ;
  - maximum execution time of the operation ;
- results obtained with the factor tree using our coding :
  - value of k ;
  - minimum execution time of the operation ;
  - average execution time of the operation ;
  - maximum execution time of the operation ;

file	length	pattern	suffix tree			factor tree			
		length	min	mean	max	k	min	mean	max
bible(t)	4047393	10	2.30	14.90	368.34	17	3.01	12.28	183.40
bible(t)	4047393	10	2.30	14.90	368.34	10	2.03	7.80	13.75
bible(t)	4047393	15	2.53	8.29	134.81	17	3.50	8.79	17.54
bible(t)	4047393	15	2.53	8.29	134.81	15	1.99	8.66	15.95
world192(t)	2473401	15	1.81	10.72	77.03	20	1.28	8.71	39.07
world192(t)	2473401	25	1.97	8.90	70.02	25	0.0	8.59	18.69
cp(f)	24604	10	0.84	5.50	19.27	18	0.85	5.79	17.08
cp(f)	24604	10	0.84	5.50	19.27	12	0.99	5.08	13.43
trans(f)	93696	20	0.92	5.28	45.79	59	0.46	6.12	26.92
trans(f)	93696	40	1.05	5.28	39.33	40	1.36	6.01	11.82
pic(b)	513217	60	0.0	8423.35	23333.33	204	0.0	4782.99	12999.99
U00096	4639221	10	2.06	3.44	17.55	12	1.84	3.21	4.80
U00096	4639221	12	2.16	3.22	12.68	12	1.97	3.13	4.93

We see that the factor tree has search plus enumeration times that are equivalent and often better than those obtained using the suffix tree with the *illi* coding. It is interesting to observe that, in all cases, the maximal search time observed is greatly improved for a factor tree as against a suffix tree.

A.1.6 Conclusion

We introduced in this paper a useful structure for indexing the at most k-long factors of a string, and only those factors. The structure preserves the properties of a suffix tree, including the on-line characteristic of its construction, and can therefore replace it advantageously in some cases. We also showed that the modifications one must do to Ukkonen's construction in order to obtain a k-factor tree for a string s are easy.

Our experimental results show that this structure leads to often important gains in terms of the time for constructing the tree, memory space and time for some usages of the tree.

Acknowledgment

We would like to thank Maxime Crochemore and Mathieu Raffinot for their help and for interesting discussions.

Bibliographie

[1] Apostolico A. and Szpankowski W. Self-alignement in words and their applications. *J. Algorithms*, 13 :446–467, 1992.

[2] A. L. Oliveira A. M. Carvalho, A. T. Freitas and M-F. Sagot. Efficient extraction of structured motifs using box-links. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE'04)*, 2004.

[3] J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Towards optimally solving the longest common subsequence problem for sequences with nested arc annotations in linear time. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, pages 99–114. Springer-Verlag, 2002.

[4] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the Conference on Data Compression*, page 201. IEEE Computer Society, 1997.

[5] Dorohonceanu B. and Nevill-Manning C.G. Accelerating protein classification using suffix tree. In *Proc. 8th Intl. Conference on Intelligent Systems for Molecular Biology ISMB 2000*, pages 128–133, 2000.

[6] Leontis N. B. and Westhof E. Geometric nomenclature and classification of RNA base pairs. *RNA*, 7 :499–512, 2001.

[7] V. Bafna, S. Muthukrishnan, and R. Ravi. Computing similarity between RNA strings. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 1–16, Espoo, Finland, 1995. Springer-Verlag, Berlin.

[8] Jeffrey E. Barrick, K. A. Corbino, W. C. Winkler, Ali Nahvi, Maumita Mandal, Jennifer Collins, Mark Lee, Adam Roth, Narasimhan Sudarsan, Inbal Jona, J. Kenneth Wickiser, and Ronald R. Breaker. New

RNA motifs suggest an expanded scope for riboswitches in bacterial genetic control. *PNAS*, 101(17) :6421–6426, 2004.

[9] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression*. Prentice-Hall, Inc., 1990.

[10] G. Blin, G. Fertin, I. Rusu, , and C. Sinoquet. RNA sequences and the edit(nested,nested) problem. Technical Report RR-IRIN-03.07, IRIN, Université de Nantes, 2003.

[11] G. Blin, G.Fertin, R. Rizzi, and S. Vialette. Pattern matching in arc-annotated sequences : New results for the aps problem. In *Proceedings of the 5th Journée Ouverte Biologie Informatique Mathématiques (JOBIM'04)*, 2004.

[12] A. Brazma, I. Jonassen, J. Vilo, and E. Ukkonen. Predicting gene regulatory elements *in silico* on a genomic scale. *Genome Research*, 8(11) :1202–1215, 1998.

[13] J. W. Brown. The ribonuclease P database. *Nucleic Acids Research*, 24(1) :314, 1999.

[14] Dock-Bregeon A. C., Chevrier B., Podjarny A., Johnson J., de Bear J. S., Gough G. R., Gilham P. T., and Moras D. Crystallographic structure of an RNA helix : [U(UA)6A]2. *J Mol Biol.*, 209(3) :459–474, 1989.

[15] J.J Cannone, S. Subramanian, M.N. Schnare, J.R. Collett, L.M. D'Souza, Y. Du, B. Feng, N. Lin, L.V. Madabusi, K.M. Muller, N. Pande, Z. Shang, N. Yu, and R.R. Gutell RR. The comparative RNA web (CRW) site : an online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BMC Bioinformatics*, 3(1), 2002.

[16] Jih-H. Chen, Shu-Yun Le, and Jacob V. Maizel. Prediction of common secondary structures of RNAs : a genetic algorithm approach. *Nucl. Acids Res.*, 28(4) :991–999, 2000.

[17] G. D. Collins, S. Le, and K. Zhang. A new algorithm for computing similarity between RNA structures. *Inf. Sci.*, 139(1-2) :59–77, 2001.

[18] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert, 2001.

[19] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, Inc., 1994.

[20] Bouthinon D. and Soldano H. A new method to predict the consensus secondary structure of a set of unaligned RNA sequences. *BioInformatics*, 15(10) :785–798, 1999.

[21] S. Dulucq and L. Tichit. Rna secondary structure comparison : exact analysis of the zhang–shasha tree edit algorithm. *Theor. Comput. Sci.*, 306(1-3) :471–484, 2003.

[22] S. Dulucq and H. Touzet. Analysis of tree edit distance algorithms. In *Proceedings of CPM'03*, volume 2676, pages 83–95. Springer-Verlag, 2003.

[23] P. A. Evans. *Algorithms and Complexity for annoted Sequence Analysis.* Ph. D. Thesis, University of Victoria, 1999.

[24] P. A. Evans. Finding common subsequence with arcs and pseudoknots. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, number 1645 in LNCS, pages 270–280, 1999.

[25] Noller H. F. The structure of ribosomal RNA. *Ann Rev Biochem*, 53 :119–162, 1984.

[26] Tahi F., Gouy M., and Regnier M. Automatic RNA secondary structure prediction with a comparative approach. *Comput Chem.*, 26(5) :521–530, 2002.

[27] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4) :490–505, 1989.

[28] W. Fontana, D. A. M. Konings, P. F. Stadler, and P. Schuster. Statistics of RNA secondary structures. *Biopolymers*, 33 :1389–1404, 1993.

[29] National Center for Biotechnology Information.

[30] Vitreschak A. G., Rodionov D. A., Mironov A. A., and Gelfand M. S. Riboswitches : the oldest mechanism for the regulation of gene expression ? *Trends Genet.*, 20(1) :44–50, 2004.

[31] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner : A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3) :331–353, 1997.

[32] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proceedings of the 3rd International Workshop on Algorithm Engineering*, pages 30–42. Springer-Verlag, 1999.

[33] J Gorodkin, LJ Heyer, and GD Stormo. Finding the most significant common sequence and structure motifs in a set of RNA sequences. *Nucl. Acids Res.*, 25(18) :3724–3732, 1997.

[34] J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. In *Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*, pages 182–193. Springer-Verlag, 2002.

[35] J. Guo. *Exact Algorithms for the LONGEST COMMON SUBSEQUENCE Problem for Arc-Annotated Sequences*. Ph. D. Thesis, Universität Tübingen, 2002.

[36] D. Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge University Press, 1997.

[37] R. R. Gutell, N. Larsen, and C. R. Woese. Lessons from an evolving rRNA : 16s and 23s rRNA structures from a comparative perspective. *Microbiol. Rev.*, 58(1) :10–26, 1994.

[38] Li W. H. *Molecular Evolution*. Sinauer Associates, Sunderland, Massachusetts., 1999.

[39] Touzet H. and Perriquet O. CARNAC : folding families of related RNAs. *Nucleic Acids Res.*, 32, 2004.

[40] K Han and HJ Kim. Prediction of common folding structures of homologous RNAs. *Nucl. Acids Res.*, 21(5) :1251–1257, 1993.

[41] D. S. Hirschberg. *The longest common subsequence problem*. PhD thesis, Princeton University, 1975.

[42] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatsh. Chem.*, 125 :167–188, 1994.

[43] J. Hong and X. Tan. The taxonomy-based learnin : Algorithms and applications. Technical Report UM-CS-1989-065, University of Massachusetts at Amherst, 1989.

[44] M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz. Local similarity in RNA secondary structures. In *Proceedings of the IEEE Computer Society Conference on Bioinformatics*, page 159. IEEE Computer Society, 2003.

[45] M. Höchsmann, B. Voss, and R. Giegerich. Pure multiple RNA secondary structure alignments : A progressive profile approach. *TCCB*, 1(1) :53–62, 2004.

[46] Felsenstein J. *Inferring Phylogenies.* Sinauer Associates, Sunderland, Massachusetts., 2004.

[47] T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures, 2002.

[48] T. Jiang, G. H. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 154–165. Springer-Verlag, 2000.

[49] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 75–86. Springer-Verlag, 1994.

[50] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, pages 91–102. Springer-Verlag, 1998.

[51] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1) :323–350, 1977.

[52] S. Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13) :1149–1171, 1999.

[53] Collins L., Moulton V., and Penny D. Use of RNA secondary structure for studying the evolution of RNase P and RNase MRP. *Journal of Molecular Evolution*, 51 :194–204, 2000.

[54] N. Jesper Larsson. Extended application of suffix trees to data compression. In J. A. Storer and M. Cohn, editors, *ProceedingsData Compression Conference*, pages 190–199, Snowbird, UT, 1996. IEEE Computer Society Press.

[55] N. Jesper Larsson. *Structures of String Matching and Data Compression.* PhD thesis, Department of Computer Science, Lund University, Sweden, September 1999.

[56] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.*, 6 :707–710, 1966.

[57] The GNU Free Documentation License.

[58] The GNU General Public License.

[59] G. Lin, Z. Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. *J. Comput. Syst. Sci.*, 65(3) :465–480, 2002.

[60] G. H. Lin, B. Ma, and K. Zhang. Edit distance between two RNA structures. In *Proceedings of the fifth annual international conference on Computational biology*, pages 211–220. ACM Press, 2001.

[61] Crochemore M. and Rytter W. *Jewels of Stringology*. World Scientific, 2002.

[62] Nei M. and Kumar. S. *Molecular Evolution and Phylogenetics*. Oxford University Press, Oxford., 2000.

[63] Zuker M. On finding all suboptimal foldings of an RNA molecule. *Science*, 244 :48–52, 1989.

[64] Zuker M., Mathews D. H., and Turner D. H. Algorithms and thermodynamics for RNA secondary structure prediction : A practical guide. In J. Barciszewski and B.F.C. Clark, editors, *RNA Biochemistry and Biotechnology*. Kluwer Academic Publishers, 1999.

[65] Zuker M. and Stiegler P. Optimal computer folding of large RNA sequences using thermodynamic and auxiliary information. *Nucl Acid Res*, 9 :133–148, 1981.

[66] B. Ma, L. Wang, and K. Zhang. Computing similarity between RNA structures. *Theor. Comput. Sci.*, 276(1-2) :111–132, 2002.

[67] L. Marsan and M. F. Sagot. Extracting structured motifs using a suffix tree
u2014algorithms and application to promoter consensus identification. In *Proceedings of the fourth annual international conference on Computational molecular biology*, pages 210–219. ACM Press, 2000.

[68] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2) :262–272, 1976.

[69] J. C. Na and K. Park. Data compression with truncated suffix trees. In *Proceedings of the Conference on Data Compression*, page 565. IEEE Computer Society, 2000.

[70] Holbrook S. R., Cheong C., Tinoco I. J.r., and Kim S. H. Crystal structure of an RNA double helix incorporating a track of non-watson-crick base pairs. *Nature*, 353 :579–581, 1991.

[71] Holbrook S. R. and Kim S. H. RNA crystallography. *Biopolymers*, 44(1) :3–21, 1997.

[72] Kumar S., , and Rzhetsky A. Evolutionary relationships among eukaryotic kingdoms. *J. Mol. Evol.*, 42 :183–193, 1996.

[73] Lemieux S. and Major F. RNA canonical and non-canonical base pairing types : a recognition method and complete repertoire. *Nucleic Acids Research*, 30(19) :4250–4263, 2002.

[74] McCaskill J. S. The equilibrium partition function and base pair binding probabilities for RNA secondary structures. *Biopolymers*, 29 :1105–1119, 1990.

[75] M-F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In C. L. Lucchesi and A. V. Moura, editors, *Proceedings of the3rd Latin American Symposium*, pages 374–390, Campinas, Brazil, 1998. Springer-Verlag, Berlin.

[76] D. Sankoff and J. B. Kruskal. *Time warps, string edits, and macromolecules : the theory and practice of sequence comparison*. Addison-Wesley, Reading, MA, 1983.

[77] S. M. Selkow. The tree-to-tree editing problem. *Inform. Process. Lett.*, 6(6) :184–186, 1977.

[78] B. Shapiro. An algorithm for comparing multiple RNA secondary structures. *Comput. Appl. Biosci.*, 4(3) :387–393, 1988.

[79] B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Comput. Appl. Biosci.*, 6(4) :309–318, 1990.

[80] E. Sobel and H. M. Martinez. A multiple sequence alignment program. *Nucl. Acids. Res.*, 14(1) :363–374, 1986.

[81] N. Sudarsan, J.K. Wickiser, S. Nakamura, M.S. Ebert, and R.R. Breaker. An mRNA structure in bacteria that controls gene expression by binding lysine. *Genes Dev.*, 17(21) :2688–2697, 2003.

[82] W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Comput.*, 22(6) :1176–1198, 1993.

[83] K. C. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3) :422–433, 1979.

[84] P. Thebault. *Développement d'un outil générique pour la localisation de motifs structurés dans les textes génomiques et protéiques : Extension du langage, adaptation et amélioration de l'efficacité des algorithmes*. PhD thesis, INRA, Toulouse, 2004.

[85] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3) :249–260, 1995.

[86] Marsan L. Sagot M.-F. V. A. Promoter sequences and algorithmical methods for identifying them. *Research in Microbiology*, 150 :779–799, 1998.

[87] G. Valiente. An efficient bottom-up distance between trees. In *Proc. 8th Int. Symposium on String Processing and Information Retrieval*, pages 212–219. IEEE Computer Science Press, 2001.

[88] S. Vialette. *Aspects algorithmiques de la prédiction des structures secondaires d'ARN.* Ph. D. Thesis, Université Paris VII, 2001.

[89] S. Vialette. Pattern matching problems over 2-interval sets. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, pages 53–63. Springer-Verlag, 2002.

[90] S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theor. Comput. Sci.*, 312(2-3) :223–249, 2004.

[91] Saenger W. Principles of nucleic acid structure. *Springer-Verlag*, 1984.

[92] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and K. M. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(8) :889–895, 1998.

[93] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[94] W. C. Winkler, S. Cohen-Chalamish, and R. R. Breaker. An mRNA structure that controls gene expression by binding FMN. *PNAS*, 99(25) :15908–15913, 2002.

[95] Christina Witwer, Ivo L. Hofacker, and Peter F. Stadler. Prediction of consensus rna secondary structures including pseudoknots. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(2) :66–77, 2004.

[96] C R Woese, L J Magrum, R Gupta, R B Siegel, D A Stahl, J Kop, N Crawford, J Brosius, R Gutell, J J Hogan, and H F Noller. Secondary structure model for bacterial 16s ribosomal RNA : phylogenetic, enzymatic and chemical evidence. *Nucleic Acids Res.*, 8(10) :2275–2293, 1980.

[97] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7) :739–755, 1991.

[98] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6) :1245–1262, 1989.

[99] K. Zhang, L. Wang, and B. Ma. Computing similarity between RNA structures. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, volume 1645, pages 281–293. Springer-Verlag, Berlin, 1999.

[100] M. Zuker and D. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46 :591–621, 1984.