Grammaires S-attribuées multi-bandes et applications à l'analyse automatique de séquences biologiques

Fabrice Lefebvre
LIX
École Polytechnique
91128 Palaiseau CEDEX, FRANCE
lefebvre@lix.polytechnique.fr

30 mai 1997

Introduction

Problématique

Cette thèse destinée en priorité à un public informaticien, mais sans exclure pour autant les biologistes familiers des problèmes de bio-informatique, a trois objectifs :

- montrer que les grammaires formelles, sous la forme de grammaires S-attribuées multi-bandes (MTSAG, pour Multi-Tape S-Attribute Grammars), permettent de décrire élégamment de nombreux problèmes et modèles d'analyse de séquences biologiques,

- introduire un algorithme d'analyse syntaxique adapté aux MTSAG, puis une implémentation particulièrement performante de cet algorithme, afin d'ériger les MTSAG en un formalisme qui ne se contente pas d'être élégant mais qui sache aussi se montrer utile et efficace,

- convaincre le lecteur de l'inefficacité du formalisme classique des équations de programmation dynamique lorsqu'il s'agit de décrire des algorithmes complexes d'analyse de séquences ou de structures d'ARN.

Comme nous le verrons, la description de problèmes d'analyse de séquences à l'aide de méthodes formelles diverses trouvant leurs racines dans la théorie des automates finis ou la théorie des langages n'est pas une nouveauté. Cette thèse unifie sous la forme de MTSAG toutes les méthodes formelles précédemment utilisées dans le contexte de l'analyse de séquences et met en évidence la richesse conceptuelle de cette unification. Sans prétendre à l'exhaustivité, nous appliquerons les MTSAG à une large gamme de problèmes d'analyse de séquences et nous nous intéresserons plus particulièrement aux problèmes de repliement et/ou d'alignement des ARN car ils sont réputés être parmi les plus difficiles. Nous introduirons également plusieurs modèles nouveaux qui auraient sans doute été incompréhensibles, voire inaccessibles, à l'aide des seules équations de programmation dynamique traditionnelles.

Toutes les applications des MTSAG exposées dans cette thèse bénéficient d'un algorithme d'analyse syntaxique spécialement adapté aux MTSAG décrit dans les deux premiers chapitres, et dont l'implémentation performante a représenté une part importante du travail associé à cette thèse.

Chronologie

Le lecteur de tout document présentant la version finale et épurée d'un travail est toujours plus à même de comprendre ce travail s'il a connaissance des motivations initiales et des errements successifs de son auteur. C'est pourquoi j'ai tenu à inclure cette chronologie.

L'impulsion initiale de ce travail fut donnée par Jean-Marc Steyaert en 1993 alors que je terminais un stage de DEA sous sa direction (Lefebvre, 1995a, 1994) et qu'il me proposait de débuter une thèse ayant pour objet la formalisation de divers problèmes d'analyse de séquences. Comme il était fermement convaincu que les langages formels avaient un rôle à jouer dans ce processus, il m'a naturellement demandé d'examiner les problèmes liés à la détermination des structures secondaires d'ARN à l'aide de grammaires.

Après un premier mois passé à approfondir ma bibliographie du sujet, je pensais encore que le problème du repliement d'un ARN isolé était soluble, peut-être à l'aide du modèle thermodynamique, mais j'estimais que les recherches ayant pour seul but d'optimiser les paramètres de ce modèle manquaient de portée.

Les premières évaluations qualitatives montrèrent rapidement que les grammaires non-contextuelles permettent d'exprimer agréablement les contraintes structurales dans les structures secondaires d'ARN (forme en trèfle, appariements forcés, etc.). Il restait encore à vérifier que le modèle thermodynamique et ses règles de calcul d'énergie étaient exprimables à l'aide d'une grammaire, ce qui fut fait grâce à l'introduction d'une première grammaire S-attribuée. Aucun code n'avait encore été écrit, mais j'étais alors fermement persuadé que le formalisme grammatical était condamné à rester un jouet d'informaticien, sans intérêt pour les biologistes, si les analyseurs syntaxiques qu'il nécessite en pratique se révélaient trop lents et trop gourmands en mémoire par rapport aux programmes existants. Je fis donc une étude de faisabilité en écrivant rapidement un premier analyseur syntaxique utilisant une des nombreuses variations de l'algorithme de Earley (Graham *et al.*, 1980) et dans lequel cette première grammaire S-attribuée était codée "en dur". Les premiers tests furent totalement catastrophiques. L'analyseur était plusieurs centaines de fois plus lent et gourmand en mémoire que le programme *Vienna*, qui utilisait quant à lui les classiques équations de programmation dynamique de Zuker. Une opération de chasse aux gaspis n'améliora pas de manière déterminante les performances de ce programme. L'analyse des faiblesses de l'algorithme de Earley et du type d'ambiguïté de la grammaire S-attribuée utilisée conduisirent deux mois plus tard à une ébauche d'analyseur syntaxique utilisant l'algorithme présenté dans cette thèse. Les premier tests furent simplement mauvais et il fallut un peu plus de six mois supplémentaires d'analyse acharnée du comportement de l'analyseur et d'optimisations subséquentes avant d'obtenir un analyseur syntaxique dont les performances étaient comparables à celles du programme *Vienna*. Cet analyseur n'utilisait déjà plus une grammaire codée en dur mais était obtenu grâce à un générateur automatique d'analyseurs syntaxiques (le futur outil MTSAG2C). Je testai également ce générateur sur les grammaires stochastiques introduites entre-temps par l'équipe de Haussler à l'université de Californie Santa Cruz et constatai bientôt que les analyseurs produits étaient plus rapides que ceux de type Cocke–Younger–Kasami utilisés par l'équipe de Haussler (Lefebvre, 1995b).

Il était entre-temps devenu de plus en plus évident pour la communauté scientifique que le problème de la détermination des structures secondaires des ARN ne pourrait pas être résolu de manière satisfaisante sans utiliser des alignements d'ARN homologues, alignements dont la détermination nécessite elle-même une étude de la structure secondaire consensus de ces ARN. Dans ce contexte, les grammaires stochastiques représentaient, et représentent toujours, un formalisme intéressant car il est possible de produire des alignements corrects d'ARN de la même famille à partir d'une grammaire stochastique convenable dont la détermination reste cependant problématique[1]. J'ai donc orienté mes recherches sur les problèmes de construction automatique de grammaires à partir de familles d'ARN bruts, tout en continuant à améliorer le générateur d'analyseurs syntaxiques.

Je considérai d'abord les méthodes de comparaison d'arbres de dérivation dans l'espoir de construire une grammaire à partir d'une intersection d'arbres de dérivation d'ARN de la même famille. Mes premières investigations réussirent à me faire prendre conscience du caractère largement non trivial du problème. De plus, je me heurtais sans cesse à la remarque fondamentale suivante : à quoi bon comparer des arbres en vue de construire une grammaire si les arbres de départ sont faux, ou si aucune preuve de leur justesse n'est disponible ? La comparaison de familles d'arbres de dérivation contenant éventuellement le bon arbre, plutôt que la comparaison d'arbres de dérivation isolés, n'allait évidemment pas dans le sens d'une simplification du problème.

Une relecture simultanée d'un ancien brouillon et d'un exposé de Searls à la conférence ISMB'95 (Searls & Murphy, 1995) me suggéra en décembre 1995 le concept de grammaires multi-bandes et une série d'évaluations me convainquit rapidement de l'intérêt de ce nouveau formalisme. Je m'engageai alors dans la généralisation multi-bandes du générateur d'analyseurs syntaxiques, tout en étendant sans cesse la liste des problèmes d'analyse de séquences auxquels les grammaires multi-bandes pouvaient s'appliquer. Les améliorations apportées précédemment au générateur mono-bande se révélèrent profitables aux analyseurs multi-bandes. Le restant de l'année scolaire 1995–1996 fut consacré aux améliorations (parfois sensibles) du générateur, à l'exploitation des résultats (Lefebvre, 1996a), et à la rédaction de cette thèse.

Structure du document

Cette thèse étant une thèse en informatique, nous avons délibérément pris le parti d'exposer nos résultats à l'intention d'un lecteur informaticien non nécessairement familier des problèmes d'analyse de séquences en biologie moléculaire. Nous n'avons

[1]Citons pour mémoire deux méthodes d'apprentissage de grammaires stochastiques. La première est due à Sakakibara (Sakakibara, 1990) et présuppose l'existence d'une fonction retournant vrai si la grammaire est bonne et faux plus une structure contre-exemple sinon. Une telle fonction, qui s'apparente beaucoup à un oracle, est bien sûr chimérique dans le cas des ARN. La seconde est due à Stolcke et Omohundro (Stolcke & Omohundro, 1994) et est beaucoup trop propice à la généralisation intempestive dans le cadre des ARN.

pas pour autant exclu le lecteur bio-informaticien, ou même biologiste, en détaillant raisonnablement les concepts informatiques les plus fondamentaux et en reportant les parties les plus techniques en fin de thèse. La thèse est divisée en cinq chapitres, plus une annexe et deux glossaires (un glossaire des termes biologiques et un glossaire des termes informatiques) :

Chapitre 1 Présentation des grammaires S-attribuées et introduction d'un algorithme d'analyse syntaxique adapté à ces grammaires : l'algorithme GCP. Le problème de la détermination des structures secondaires des ARN est également exposé dans ce chapitre ;

Chapitre 2 Les grammaires S-attribuées introduites au chapitre précédent sont généralisées en grammaires S-attribuées multi-bandes et l'algorithme GCP est généralisé concomitamment en algorithme GCP multi-bandes ;

Chapitre 3 Un certain nombre d'applications utilisant les grammaires S-attribuées multi-bandes sont présentées, et les performances des analyseurs syntaxiques générés sont comparées, lorsque c'est possible, à celles d'algorithmes précédemment proposés par d'autres auteurs ;

Chapitre 4 Les aspects importants de notre implémentation de l'algorithme GCP multi-bandes introduit au chapitre 2 sont discutés dans ce chapitre ;

Chapitre 5 Les propriétés des grammaires S-attribuées multi-bandes ainsi qu'un certain nombre de généralisations de ces grammaires et de l'algorithme GCP sont examinés dans ce dernier chapitre ;

Annexe A Manuel de référence du générateur automatique d'analyseurs syntaxiques utilisé tout au long de la thèse ;

Glossaire biologique Glossaire des termes de biologie moléculaire principaux utilisés dans cette thèse ;

Glossaire informatique Glossaire des termes d'informatique principaux utilisés dans cette thèse ;

Les deux premiers chapitres, bien que techniques, pourront être raisonnablement lus par un biologiste souhaitant se familiariser avec les techniques et notations utilisés dans le chapitre 3 qui est, sur le plan pratique, le chapitre le plus important de la thèse. Les résultats les plus techniques ont été reportés dans les deux derniers chapitres lorsqu'ils n'étaient pas indispensable à l'écriture et la compréhension des trois premiers chapitres. Le chapitre 3 pourra être lu en parallèle avec l'annexe A car des notions de la syntaxe du générateur d'analyseurs syntaxiques MTSAG2C sont nécessaires à l'approfondissement des exemples proposés ; Le lecteur informaticien sera ici avantagé du fait de la similitude de la syntaxe employée par MTSAG2C avec celle de YACC. Les deux derniers chapitres sont réservés au lecteur informaticien motivé qui souhaite dépasser le cadre des applications des MTSAG à la bio-informatique ou qui souhaite comprendre pourquoi les analyseurs syntaxiques générés par MTSAG2C sont plus performants que d'autres analyseurs syntaxiques.

Chapitre 1

Grammaires et Structure Secondaire des ARN

1.1 Introduction

Ce chapitre a pour but d'introduire un premier modèle de grammaires adaptées à la formulation de problèmes de détermination de structures secondaires des ARN, ainsi qu'un algorithme d'analyse syntaxique pour ces grammaires. Les grammaires, comme l'algorithme, seront généralisés dans le chapitre suivant afin de traiter une classe plus large de problèmes d'analyse de séquences biologiques. Ce chapitre introduit progressivement les concepts fondamentaux utilisés dans tout le reste de cette thèse. Pour bien comprendre l'intérêt et la difficulté du problème de la détermination de structures secondaires des ARN, il nous faut d'abord rappeler ce que sont un ARN et une structure secondaire d'ARN, puis exposer brièvement quelques méthodes habituellement utilisées pour le calcul de ces structures.

Une molécule d'ARN (ou acide ribonucléique) est formée d'une suite de ribonucléotides. Chaque ribonucléotide contient une base choisie parmi essentiellement quatre bases. Ces quatre bases sont l'adénine (A), la cytosine (C), la guanine (G) et l'uracile (U). Les ribonucléotides sont attachés les uns aux autres par une liaison phosphodiester, formant ainsi une chaîne orientée de ribonucléotides partant de l'extrémité 5' et se terminant à l'extrémité 3' (figure 1.1). Il est d'usage de décrire la **structure primaire** d'un ARN par la suite des lettres représentant les bases de ses ribonucléotides. Ainsi, l'ARN de la figure 1.1 est entièrement décrit par le mot `ACGU`.

Mais la connaissance de la structure primaire d'un ARN ne suffit pas pour expliquer l'ensemble des propriétés biologiques de cet ARN. Certains ARN, comme les ARN de transfert ou les ARN ribosomaux, possèdent une activité enzymatique marquée (Gesteland & Atkins, 1993). Cette activité enzymatique est partiellement liée à la **structure tertiaire** de l'ARN, c'est-à-dire à la position de ses ribonucléotides dans l'espace. La structure tertiaire d'un ARN est déterminée par les interactions entre ses différents nucléotides, interactions qui contribuent à lui donner une structure **repliée** non-triviale respectant la contrainte thermodynamique habituelle de minimisation de

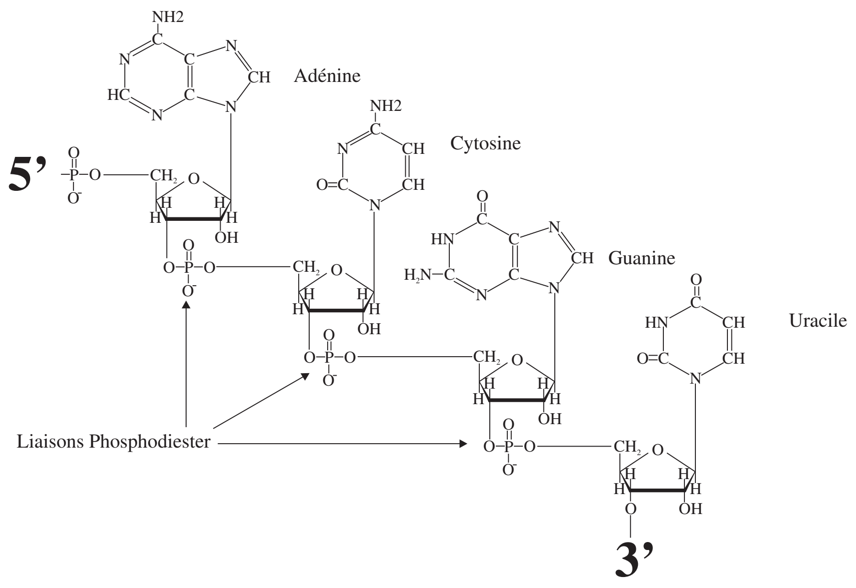


Figure 1.1: Exemple d'ARN. Le premier nucléotide de l'ARN se trouve à l'extrémité 5' et le dernier se trouve à l'extrémité 3'.

l'énergie libre totale. Les interactions favorables au **repliement** sont principalement les liaisons hydrogènes entre bases complémentaires. Un ensemble de liaisons hydrogènes entre deux bases complémentaires forme une **paire de bases**. Les paires les plus communément rencontrées dans les ARN sont les paires `G-C` et `A-U` et, dans une moindre mesure, `G-U` (figure 1.2). D'autres éléments ont un rôle à jouer dans la détermination du repliement d'énergie minimale : contraintes stéréochimiques, torsions des liaisons phosphodiester, forces électrostatiques, effets de solvant, température, présence de molécules particulières dans le solvant, etc.

De nombreuses personnes se sont intéressées à la détermination de la structure tertiaire des ARN dans l'espoir de découvrir leurs propriétés enzymatiques. Les procédés physiques classiques de mesure de structure (diffraction de rayons X dans un cristal de la substance dont la structure doit être déterminée) sont malheureusement quasi-inutilisables avec les ARN : seule la structure tertiaire des ARN de transfert a pu être déterminée de cette manière. La mesure directe étant inaccessible, il faut se tourner vers le calcul. Il est théoriquement possible de construire un modèle mathématique de l'ARN, où les interactions entre nucléotides, les effets de solvant et de température, etc., sont mis en équations. Malheureusement, le problème d'optimisation non-linéaire obtenu n'est pas informatiquement soluble, même pour les ARN les plus simples : il comporte beaucoup trop de variables et de minima locaux.

La structure tertiaire d'un ARN étant trop difficile à obtenir par calcul direct, le problème précédent est simplifié en remarquant que cette structure tertiaire dépend essentiellement des appariements entre bases complémentaires. Tout ensemble d'ap-

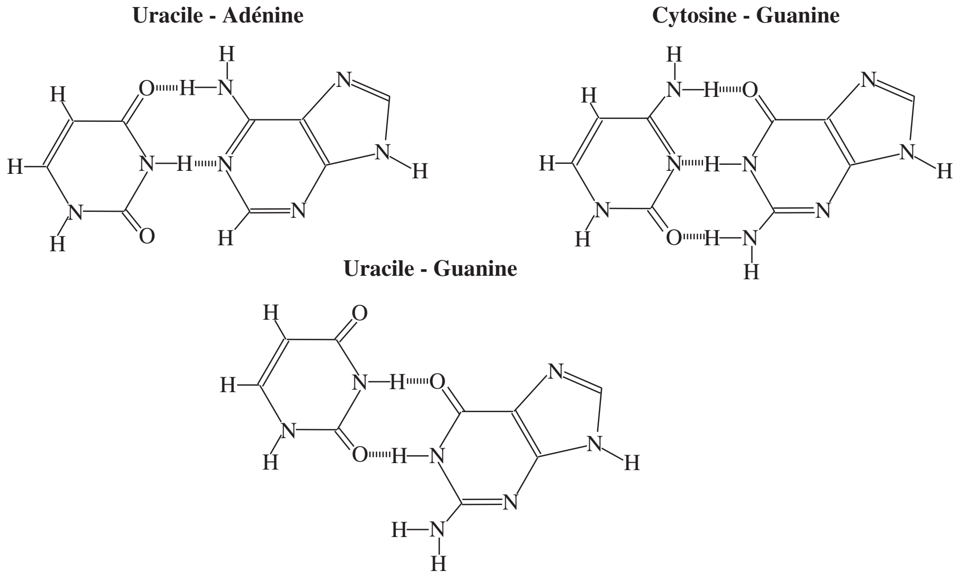


Figure 1.2: Liaisons hydrogènes impliquées dans les paires `U-A`, `C-G` et `U-G`.

pariements de bases d'un ARN est appelé **structure secondaire**. Une[1] structure secondaire est souvent représentée sous forme planaire en accolant les bases de chaque paire (figure 1.3). Une restriction d'ordre purement technique est habituellement imposée aux structures secondaires : elles ne doivent pas contenir de **pseudo-nœud** (figure 1.4). Les pseudo-nœuds sont courants dans les ARN (van Belkum *et al.*, 1985; Le *et al.*, 1991; Wang *et al.*, 1995), et même les petits ARN de transfert en contiennent, mais leur prédiction est souvent considérée comme faisant partie du domaine de l'étude de la structure tertiaire. Des recherches ont cependant montré que les pseudo-nœuds, ainsi que les bases impliquées dans les pseudo-nœuds, peuvent avoir une grande importance fonctionnelle (Pleij, 1993; ten Dam *et al.*, 1992; Pleij, 1995; Florentz *et al.*, 1984). Récemment, une méthode de détermination automatique des pseudo-nœuds communs aux ARN d'une même famille a été publiée (Cary & Stormo, 1995). Son seul inconvénient, et il est de taille, est de nécessiter un excellent alignement multiple de ces ARN. Or un alignement multiple d'ARN ne peut être véritablement excellent sans connaissance préalable des pseudo-nœuds (James *et al.*, 1989; Woese & Pace, 1993).

Par ailleurs, les structures secondaires d'ARN ne doivent pas exclusivement être considérées comme des représentations simplifiées de structures tertiaires, introduites uniquement pour faciliter un éventuel calcul ultérieur de structure tertiaire. La comparaison des structures secondaires d'ARN ayant la même fonction dans des organismes différents (c'est l'analyse phylogénétique) doit permettre de déterminer quels éléments

[1]Nous utiliserons la locution "structure secondaire" avec des articles définis, comme dans "la structure secondaire", pour désigner des structures secondaires réelles, ou cherchées. Nous utiliserons des articles indéfinis, comme dans "une structure secondaire", pour désigner une ou plusieurs structures secondaires parmi un ensemble de structures secondaires possibles.

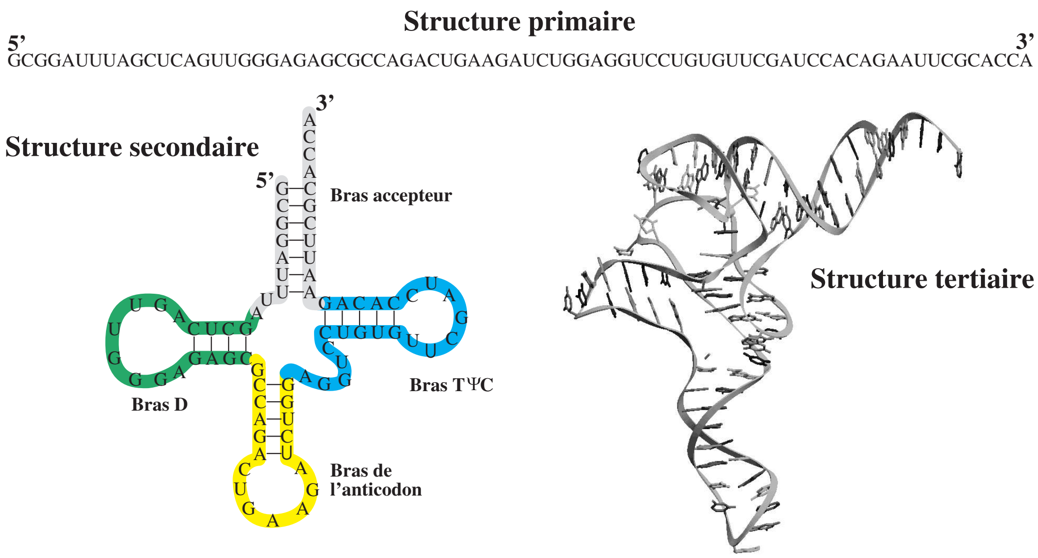


Figure 1.3: Structures primaire, secondaire et tertiaire d'un ARN de transfert de la phénylalanine. La structure secondaire a la forme d'un trèfle, caractéristique d'un ARN de transfert.

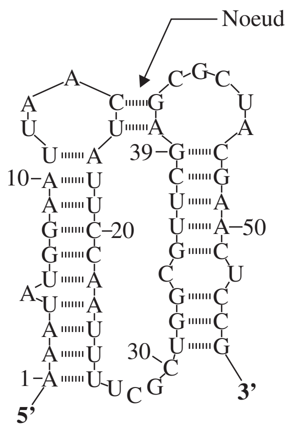


Figure 1.4: Exemple de pseudo-nœud dans une structure secondaire. Les paires 11–17 et 39–47 sont incompatibles avec les paires 15–41 et 16–40 car ces deux dernières paires impliquent des nucléotides de deux boucles différentes.

structuraux sont conservés, et donc de trouver quels sont les groupes fonctionnels importants de ces ARN, sans même faire référence à leur structure tertiaire. La pertinence de ces comparaisons est assurée par une hypothèse fondamentale : un élément structural conservé dans plusieurs organismes, appartenant éventuellement à des espèces éloignées, doit sans doute avoir un rôle fonctionnel. Nous verrons plus loin (cf. section 3.3 p. 95) que la comparaison de structures secondaires est souvent faite par le biais d'alignements de séquences.

Plusieurs méthodes ont été établies pour déterminer la structure secondaire d'un ARN. Historiquement, la première méthode fut l'analyse phylogénétique d'ARN homologues, c'est-à-dire d'ARN ayant la même fonction au sein d'organismes différents. Elle repose sur l'hypothèse citée précédemment de conservation des éléments structurels importants au cours de l'évolution. Il peut sembler à première vue curieux d'utiliser une analyse phylogénétique pour obtenir des structures secondaires alors qu'on espère justement exploiter les structures secondaires pour l'analyse phylogénétique des ARN. En fait, cette dépendance mutuelle se traduit par un aspect itératif des méthodes de résolution : la comparaison des structures secondaires de plusieurs ARN homologues permet de dégager des zones de forte similitude entre ces ARN, zones qui constituent des points d'ancrage permettant d'affiner les structures secondaires des ARN, etc. Les méthodes comparatives sont donc celles fournissant les structures secondaires les plus légitimes car les structures trouvées doivent satisfaire une contrainte très forte de cohérence les unes par rapport aux autres. La recherche et la validation des zones similaires fait souvent appel à l'analyse manuelle (James *et al.*, 1989; Woese & Pace, 1993), mais des résultats récents (Eddy & Durbin, 1994) ainsi que cette thèse (cf. section 3.12 p. 169) laissent espérer la création dans un futur proche de méthodes générales de résolution sans intervention manuelle.

Une autre méthode de repliement des ARN utilise quant à elle un modèle thermodynamique simplifié des énergies de repliement (Turner *et al.*, 1987; Freier *et al.*, 1986; Jaeger *et al.*, 1989). Ce modèle thermodynamique simplifié s'appuie sur une forme additive de l'énergie libre dépendant uniquement des interactions locales entre bases et paires de bases. Essentiellement, l'énergie d'un repliement est approchée en ajoutant les énergies d'empilement des paires de bases (énergies négatives, donc favorables) et les énergies des régions non-appariées formant des boucles (énergies positives, donc défavorables). La pertinence de ce modèle repose sur l'hypothèse suivante : la structure secondaire d'énergie minimale pour la forme approximative de l'énergie libre ainsi définie est peu différente de la structure secondaire vraie, à condition de bien choisir les paramètres thermodynamiques du modèle (Freier *et al.*, 1986; Papanicolaou *et al.*, 1984). L'expérience a montré que cette hypothèse n'est pas réaliste (Han & Kim, 1993) car les structures obtenues d'après le modèle thermodynamique dépendent trop significativement de faibles variations des paramètres du modèle. Ces paramètres sont en général fixés en fonction d'une certaine réalité thermodynamique[2] et de critères moins objectifs discutables (et discutés). Une des méthodes les plus courantes de test des

[2]Les énergies d'empilement des paires de bases ont été très étudiées grâce à des ARN complémentaires synthétisés dont on a mesuré la température de séparation des brins (Freier *et al.*, 1986).

paramètres thermodynamiques consiste à les utiliser pour replier une famille d'ARN dont les structures secondaires vraies sont bien connues (ARN de transfert ou ARN ribosomaux). Si le pourcentage d'ARN correctement repliés est proche de 100%, les paramètres sont considérés bons. Sinon, ils sont modifiés jusqu'à ce que le pourcentage devienne satisfaisant, avec le risque d'une convergence vers un modèle repliant trop bien certains ARN (ceux pour qui les paramètres ont été optimisés) et pas assez bien d'autres. Cette dimension "recette de cuisine", présente dans la méthode classique de calcul des énergies des structures secondaires, n'est pas entièrement satisfaisante, surtout à cause du caractère obscur des règles de modification des paramètres[3].

La popularité du modèle thermodynamique doit sans doute beaucoup à la facilité avec laquelle il a été possible de décrire un algorithme de programmation dynamique trouvant relativement rapidement une structure secondaire plausible (Zuker, 1989b,a; Zuker & Sankoff, 1984). La complexité de l'algorithme de programmation dynamique utilisé avec le modèle thermodynamique est en effet de $O(n^3)$ en temps et $O(n^2)$ en espace, où n est la longueur de l'ARN à replier. Cette complexité cubique en temps contraste avec la complexité exponentielle d'autres méthodes ayant une approche beaucoup plus combinatoire (Gaspin *et al.*, 1995; Dumas & Ninio, 1982). Même si les approches directes du repliement d'un ARN, comme le modèle thermodynamique, semblent passées de mode du fait de leurs insuffisances, ce sont pour l'instant les seules utilisables lorsqu'il est nécessaire d'avoir une idée de la structure secondaire d'un ARN isolé, c'est-à-dire lorsque les méthodes d'analyse phylogénétique (ou analyse comparative) sont inapplicables.

Nous allons voir dans ce chapitre que, grâce à l'introduction des grammaires non-contextuelles S-attribuées (ou SAG, pour S-attribute grammars), il est possible de conserver certains aspects intéressants du modèle thermodynamique (obtenir un repliement "énergétiquement cohérent"), tout en assurant un repliement correct d'une famille d'ARN, comme celle des ARN de transfert, sans manipulation artificielle des paramètres thermodynamiques. En fait, les SAG apportent au modèle thermodynamique la possibilité d'imposer des contraintes structurales aux repliements cherchés (sous la forme de contraintes syntaxiques), ces contraintes étant indépendantes des paramètres du modèle thermodynamique. Nous verrons également que les SAG permettent une réécriture triviale des grammaires non-contextuelles stochastiques (ou SCFG, pour Stochastic Context-Free Grammars), et constituent donc une généralisation des modèles thermodynamiques et stochastiques.

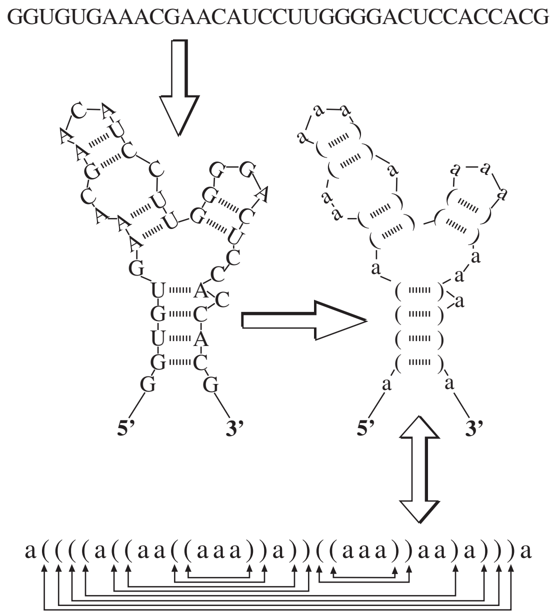
[3]L'efficacité des grammaires non-contextuelles stochastiques repose elle aussi sur des optimisations "par famille" de paramètres de calcul des structures secondaires. Mais les règles d'optimisation sont parfaitement définies et cohérentes, ce qui les distingue des "recettes de cuisine" mal justifiées, et dont la reproductibilité laisse parfois à désirer, du modèle thermodynamique.

1.2 Grammaires non-contextuelles

1.2.1 Pertinence dans le contexte des structures secondaires

Nous allons montrer qu'une recherche d'une structure secondaire sans pseudo-nœud d'un ARN peut en fait être transformée en une recherche d'un arbre de dérivation pour une grammaire non-contextuelle donnée (Searls, 1992; Vauchaussade de Chaumont, 1985). Cette grammaire est fonction du modèle de repliement utilisé (thermodynamique, stochastique, etc.) et des contraintes structurales souhaitées.

Dans un premier temps, et dans un but uniquement pédagogique, nous pouvons considérer qu'une structure secondaire sans pseudo-nœud d'un ARN est la réunion d'un ensemble de bases correctement appariées (i.e. pas de pseudo-nœud) et d'un ensemble de bases non-appariées. Si nous attribuons une parenthèse ouvrante "(" à la première base d'un appariement, une parenthèse fermante ")" à la seconde base d'un appariement, et un "*a*" à une base non-appariée, nous transformons la structure primaire d'un couple *(ARN, structure secondaire de l'ARN)* en une expression correctement parenthésée. C'est ce qu'illustre la figure suivante.



Nous voyons dans cette figure qu'il y a équivalence entre l'écriture linéaire habituelle (ou "structure primaire") d'une expression correctement parenthésée et sa "structure secondaire", dont une représentation est obtenue, comme avec les structures secondaires d'ARN, en rapprochant dans le plan les parenthèses appariées. L'équivalence entre structure primaire et structure secondaire dans les cas des expressions correctement parenthésées est vérifiée parce qu'une parenthèse ouvrante ne peut s'apparier qu'avec une seule parenthèse fermante. L'équivalence n'est évidemment pas vérifiée dans le cas des ARN puisqu'une base donnée peut potentiellement s'apparier avec de nombreuses autres bases de l'ARN. Une structure primaire, ou séquence, d'ARN peut donc se replier en de nombreuses structures secondaires différentes. En fait, le nombre de structures secondaires possibles d'un ARN contenant n bases croît exponentiellement avec n (Waterman, 1978). La bonne structure secondaire parmi toutes ces structures possibles est, par définition, fonction de la structure tertiaire qui est elle-même, comme toute structure moléculaire, sous la contrainte d'une minimisation de l'énergie libre totale. La structure tertiaire exacte étant le plus souvent inaccessible, la "bonne" structure secondaire est estimée (parfois incorrectement) grâce à des critères plus accessibles : minimisation d'une énergie calculée suivant un modèle thermodynamique simple, maximisation d'une probabilité, ressemblance avec d'autres ARN de la même famille, expérimentation, etc.

Les expressions correctement parenthésées de l'exemple précédent sont classiquement décrites par la grammaire non-ambiguë suivante :

$$\begin{aligned} E &\to \epsilon \\ E &\to aE \\ E &\to (E)E \end{aligned}$$

ou, de façon équivalente au niveau de l'écriture :

$$E \to \epsilon \mid aE \mid (E)E$$

Cette grammaire traduit simplement le fait qu'une expression E peut être définie récursivement comme une expression vide, ou une expression précédée d'un a, ou une expression entre parenthèses suivie d'un E. La relation $\to$ traduisant la décomposition d'une expression en composantes plus petites est également appelée *relation de dérivation*. La représentation, sous forme d'arbre, de l'ensemble des dérivations conduisant à une expression donnée est un *arbre de dérivation*. L'arbre de dérivation d'une expression correctement parenthésée pour la grammaire précédente peut alors être considéré comme une autre représentation possible de la structure secondaire d'une expression correctement parenthésée (figure 1.5). Si nous remplaçons dans la grammaire précédente la paire de parenthèses par les différentes paires de bases possibles, puis la lettre a par les différentes bases, nous obtenons une grammaire adaptée à la description des structures secondaires des ARN (les lettres des bases ont été écrites en minuscules pour faciliter la lecture de la grammaire):

$$\begin{aligned} E \to \epsilon \mid & aE \mid cE \mid gE \mid uE \mid \\ & aEuE \mid uEaE \mid gEuE \mid uEgE \mid gEcE \mid cEgE \end{aligned}$$

Cette grammaire associe une unique structure secondaire à chaque arbre de dérivation d'une chaîne d'ARN : les bases appariées sont révélées par les relations utilisant deux bases pour encadrer le symbole E, telle $E \to aEuE$ (figure 1.6). En outre, toute structure secondaire d'ARN peut être obtenue par transformation d'un arbre de dérivation de la grammaire. L'inévitable multiplicité des structures secondaires possibles d'un ARN se traduit donc par une nécessaire multiplicité des arbres de dérivation

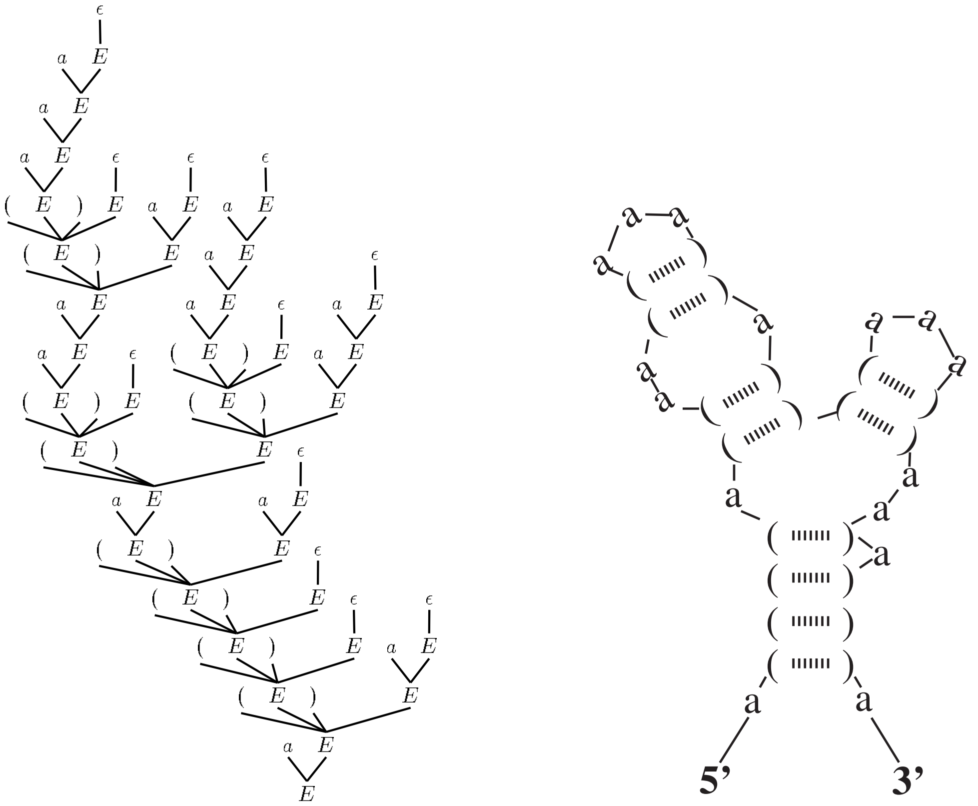


Figure 1.5: Arbre de dérivation de l'expression $a((((a((aa((aaa))a))((aaa))aa)a)))a$ pour la grammaire $E \to \epsilon \mid aE \mid (E)E$ et structure secondaire associée.

d'une même chaîne d'ARN. Les grammaires utilisées pour décrire les structures secondaires seront donc toujours ambiguës, et même à ambiguïté exponentielle. Notons qu'un arbre de dérivation n'a qu'une seule structure secondaire associée[4], alors qu'une structure secondaire est en général associée à plusieurs arbres de dérivation différents du fait de l'ambiguïté de la grammaire de départ (et également du fait de la multiplicité des grammaires utilisables pour décrire des structures secondaires). Ainsi, la grammaire suivante autorise une infinité d'arbres de dérivation pour chaque structure secondaire en raison de la présence des dérivations $E \to EE$ et $E \to \epsilon$, qui autorisent des dérivations circulaires $E \to^+ E$:

$$
\begin{aligned}
E \to \epsilon \; &|aE \mid cE \mid gE \mid uE \mid EE \\
&aEu \mid uEa \mid gEu \mid uEg \mid gEc \mid cEg
\end{aligned}
$$

Cette multiplicité des grammaires décrivant un même ensemble de structures secondaires est commode car elle permet de changer de grammaire en fonction de la

[4]La structure secondaire dépend en fait de l'interprétation donnée aux bases dans les règles de la grammaire. Dans l'exemple donné, tout couple de bases encadrant un non-terminal est qualifié d'appariement, les autres bases appartenant alors à un fragment simple-brin.

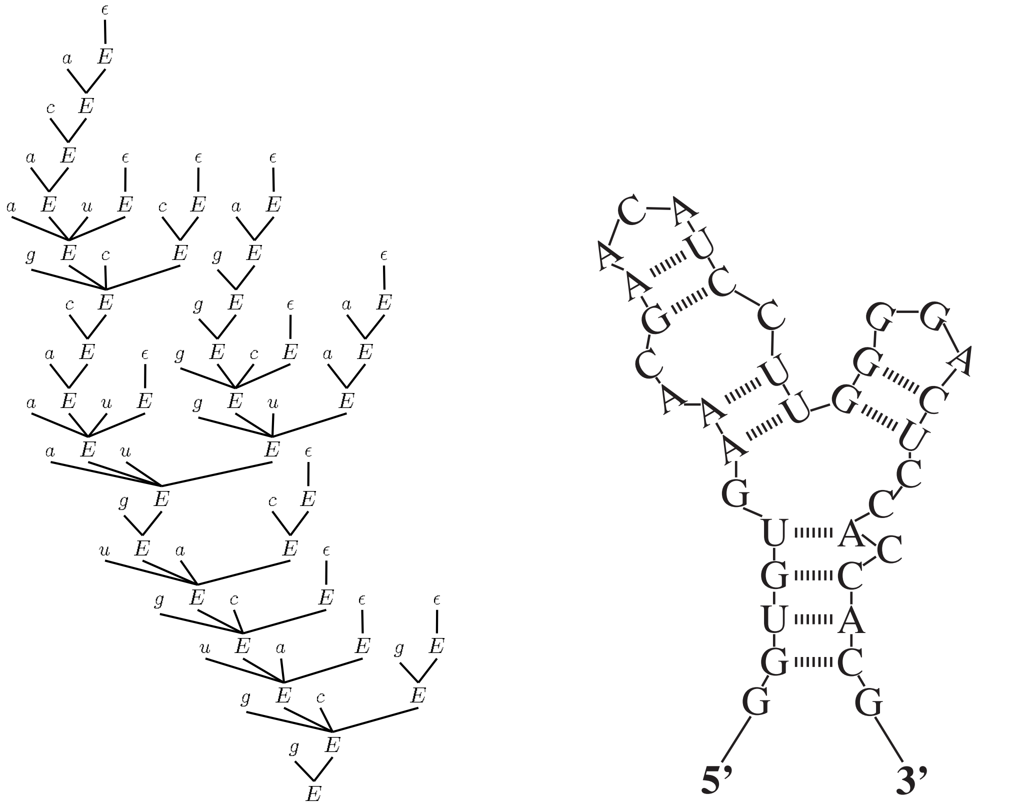


Figure 1.6: Arbre de dérivation de l'ARN ggugugaaacgaacauccuuggggacuccaccacg pour la grammaire $E \rightarrow \epsilon \mid aE \mid cE \mid gE \mid uE \mid aEuE \mid uEaE \mid gEuE \mid uEgE \mid gEcE \mid cEgE$. La structure secondaire associée à cet arbre est représentée à droite. Les minuscules sont employées dans l'arbre de dérivation et les relations pour bien distinguer les bases du symbole E, alors que des majuscules sont utilisées dans la structure secondaire car il s'agit de la notation classique des bases d'un ARN.

manière dont les structures doivent être dérivées des arbres de dérivation. Par exemple, le modèle thermodynamique impose une décomposition d'une structure en boucles de types différents (cf. section 1.7.2.1 p. 51). Ces types de boucles seront agréablement représentés par des non-terminaux différents dans la grammaire. Nous verrons plus loin (section 1.2.3) qu'il est également possible de créer des grammaires ne produisant que des arbres de dérivation d'une certaine forme, et donc des structures secondaires possédant certaines caractéristiques (structure en feuille de trèfle par exemple pour les ARN de transfert).

1.2.2 Définitions

Les définitions suivantes sont classiques en théorie des langages formels (Aho & Ullman, 1972; Salomaa, 1973; Harrison, 1978). Le lecteur informaticien pourra se contenter de parcourir rapidement cette section pour prendre connaissance des notations utilisées dans la thèse.

Définition 1.1 (Grammaire non-contextuelle)
Une grammaire non-contextuelle $G = (V_T, V_N, P, S)$ est composée d'un ensemble fini de terminaux V_T, d'un ensemble fini de non-terminaux V_N tel que $V_N \cap V_T = \emptyset$, d'un ensemble fini P de productions (dites aussi règles de réécriture ou règles de dérivation), et d'un symbole de départ $S \in V_N$ (dit aussi axiome). On note $V = V_N \cup V_T$ le vocabulaire de la grammaire.

Chaque production de P est de la forme $A \to \alpha$, où $A \in V_N$ et $\alpha \in V^*$ (V^* est le monoïde libre engendré par V). Le non-terminal A est la partie gauche de la production et α est sa partie droite. Les productions sont donc aussi des relations sur $V_N \times V^*$.

L'existence d'une production $A \to \alpha$ dans une grammaire G signifie simplement qu'une chaîne $\beta A \gamma \in V^*$ peut être dérivée, ou réécrite, en $\beta\alpha\gamma$. Les productions de G sont donc également appelées règles de réécriture. Les productions de G définissent également une relation $\to$ sur $V \times V^*$. Cette relation est naturellement prolongée en une relation sur $V^* \times V^*$. La clôture transitive de cette relation est notée $\to^+$ et sa clôture réflexive transitive est notée $\to^*$. L'ensemble des chaînes de terminaux dérivables de S est le langage $L(G)$ généré par G : $L(G) = \{u \in V_T^* \mid S \to^* u\}$. Une suite de dérivations de S en une chaîne de V_T^* est classiquement représentée sous la forme d'un arbre de dérivation. Une chaîne vide de V^* sera notée ϵ. Nous aurons également besoin de la notion de grammaire non-contextuelle propre définie ci-dessous.

Définition 1.2 (Grammaire non-contextuelle propre)
Une grammaire non-contextuelle G est dite propre lorsque :

- *G ne contient ni symbole ni production inutile, c'est-à-dire n'apparaissant au cours d'aucune dérivation de S ou ne pouvant pas ultimement générer une chaîne de V_T^*,*
- *Il n'existe aucune production $A \to \epsilon$, ou il existe une unique production $S \to \epsilon$ et S n'apparaît dans aucune partie droite des productions de G. On dit dans ce cas que G est ϵ-free ;*
- *Il n'existe aucune dérivation $A \to^+ A$. Cela revient à dire que la grammaire est non-circulaire.*

La restriction de non-circularité permet de conserver un nombre fini d'arbres de dérivation pour n'importe quelle chaîne de $L(G)$.

Lorsqu'aucune confusion n'est possible sur le type des symboles (les terminaux sont souvent notés en minuscules et les non-terminaux en majuscules), une grammaire est souvent décrite par la simple énumération de ses productions ou règles. Le premier non-terminal de la première règle est alors pris comme symbole de départ des dérivations. Par exemple, l'ensemble de productions suivantes :

$$P = \begin{cases} A \to B & A \to C \\ B \to aSu & B \to uSa \\ C \to gSc & C \to cSg \end{cases}$$

s'écrit aussi :

$$P = \begin{cases} A \to B \mid C \\ B \to aSu \mid uSa \\ C \to gSc \mid cSg \end{cases}$$

et s'intègre dans la grammaire $G = \big(\{a, c, g, u\}, \{A, B, C\}, P, A\big)$.

Remarque 1.1 Toute grammaire non-contextuelle $G = (V_T, V_N, P, S)$ telle que S n'est pas dérivable en ϵ peut être transformée en grammaire non-contextuelle propre (Aho & Ullman, 1972).

Remarque 1.2 Les grammaires non-contextuelles (CFG) que nous venons de définir ne sont pas suffisamment puissantes pour décrire des structures secondaires contenant des pseudo-nœuds. Schématiquement, reconnaître un pseudo-nœud dans une structure secondaire revient à reconnaître le langage $\{a^i b^j c^i d^j \mid i > 0, j > 0\}$ (Brown, 1995). Or ce langage n'est généré par aucune grammaire non-contextuelle (cf. section 3.13 p. 171) (Aho & Ullman, 1972; Bar-Hillel, 1964).

Remarque 1.3 Les grammaires non-contextuelles sont habituellement utilisées pour générer un langage donné (langage de programmation, langage naturel, etc.) avec, de préférence, un petit nombre d'arbres de dérivation possibles pour chaque chaîne du langage généré. Les grammaires que nous utiliserons dans cette thèse génèrent un langage sans intérêt puisqu'il s'agit le plus souvent de V_T^*. Ici, seuls nous intéressent les arbres de dérivation de ces grammaires, qui sont donc ambiguës, et même à ambiguïté exponentielle. Cela a des conséquences importantes sur le choix de l'algorithme d'analyse syntaxique, car cet algorithme aura à prendre en compte un nombre d'arbres croissant exponentiellement avec le nombre de bases des ARN analysés.

Pour éviter de multiplier inutilement les qualificatifs d'appartenance, nous utiliserons dans la suite les symboles et conventions classiques suivants :

- A, B, C pour décrire des éléments de V_N ;
- a, b, c pour décrire des éléments de V_T ;
- u, v, w pour décrire des éléments de V_T^* ;
- X, Y, Z pour décrire des éléments de V ;
- α, β, γ pour décrire des éléments de V^* ;

1.2.3 Exemples de grammaires non-contextuelles

Un des buts de ce chapitre est de montrer que les grammaires permettent d'introduire simplement des contraintes structurales sur l'ensemble des repliements possibles d'un ARN. Il est en effet plus efficace de chercher une structure secondaire dans un ensemble contenant uniquement les structures satisfaisant certaines contraintes, que de chercher cette structure dans un ensemble beaucoup plus vaste contenant toutes les structures possibles, et donc de vérifier à chaque fois la satisfaction des contraintes structurales.

Jusqu'à présent, nous avons vu qu'une grammaire non-contextuelle permet de décrire un ensemble d'arbres de dérivation (et donc un ensemble de structures secondaires). La grammaire simplifiée suivante sera utilisée pour illustrer le mécanisme d'introduction des contraintes structurales dans les ARN.

$$
\begin{aligned}
&S \to E \mid B \mid SS \\
&E \to aSu \mid uSa \mid gSc \mid cSg \mid gSu \mid uSg \\
&B \to a \mid c \mid g \mid u \\
&B \to Ba \mid Bc \mid Bg \mid Bu
\end{aligned}
$$

Dans cette grammaire, le non-terminal E désigne les empilements de paires de bases tandis que B désigne les portions simple brin et S les suites d'empilements et de portions simple brin. Cette grammaire, très générale puisqu'elle peut décrire l'ensemble des structures secondaires possibles, peut facilement être modifiée pour introduire des contraintes sur les structures secondaires cherchées.

Exemple 1.1 (Forcer des appariements connus) *Si l'on sait qu'une base donnée d'un ARN (la base 10 par exemple) s'apparie avec une autre base donnée (la base 75 par exemple), il suffit, lors d'un prétraitement de la chaîne d'ARN, de transformer la première des deux bases données en parenthèse ouvrante "(" et la deuxième base en parenthèse fermante ")", puis d'ajouter une règle, d'appariement uniquement, absorbant ces deux symboles.*

$$
\begin{aligned}
&S \to E \mid B \mid SS \\
&E \to aSu \mid uSa \mid gSc \mid cSg \mid gSu \mid uSg \\
&E \to (\,S\,) \qquad \text{(règle ajoutée)} \\
&B \to a \mid c \mid g \mid u \\
&B \to Ba \mid Bc \mid Bg \mid Bu
\end{aligned}
$$

La règle $E \to (\,S\,)$ est la seule règle dérivant les parenthèses. Donc les deux bases associées à ces parenthèses seront forcément appariées dans une structure secondaire.

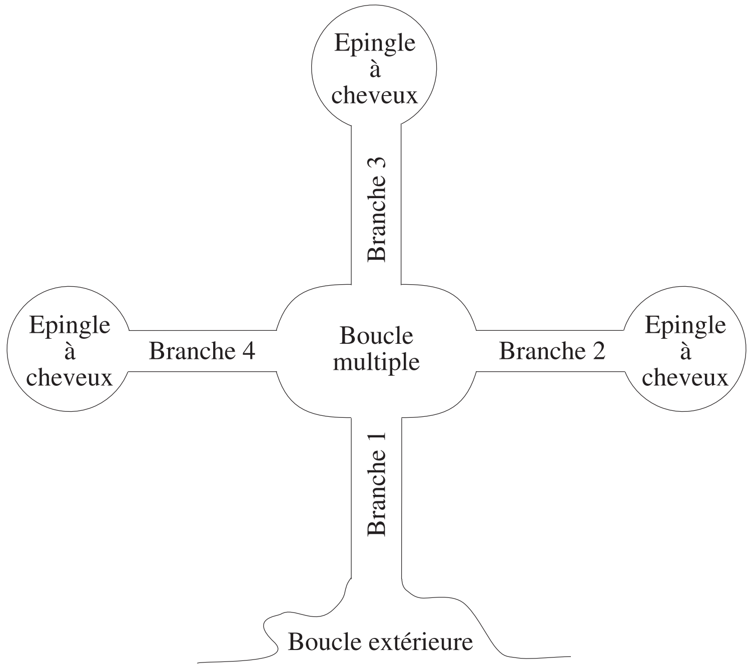


Figure 1.7: Forme stylisée d'une structure secondaire en trèfle.

Exemple 1.2 (Forcer certaines bases à ne pas s'apparier) *Il s'agit là d'une sorte d'inverse du problème précédent. Si les bases 22 à 25 d'un ARN doivent être non-appariées, il suffit de transformer chacune de ces bases en un nouveau terminal (un caractère $\star$ par exemple), puis d'ajouter ce terminal à la liste des terminaux dérivables du non-terminal B codant les portions simple-brin d'une structure secondaire :*

$$
\begin{aligned}
&S \to E \mid B \mid SS \\
&E \to aSu \mid uSa \mid gSc \mid cSg \mid gSu \mid uSg \\
&B \to a \mid c \mid g \mid u \\
&B \to Ba \mid Bc \mid Bg \mid Bu \\
&B \to \star \mid B\ \star \qquad \text{(règles ajoutées)}
\end{aligned}
$$

Exemple 1.3 (Forcer une structure en forme de feuille de trèfle) *Pour qu'une structure secondaire ait une forme de trèfle (figure 1.7), il faut qu'elle contienne un et un seul point de branchement, ce point de branchement séparant la branche principale en trois autres sous-structures contenant des empilements. La seule règle de branchement de la grammaire originale ($S \to SS$) doit être supprimée car elle permet un nombre illimité de branchements à n'importe quel endroit. Il faut la remplacer par une règle séparant la branche principale en trois sous-branches ne pouvant pas à leur tour se séparer. Nous noterons donc S_2 et E_2 les non-terminaux codant les structures dans une des trois sous-branches et $E \to S_2 S_2 S_2$ la règle de séparation. Les règles* *de dérivation de S_2 et E_2 sont déduites des règles de dérivation de S et E, règle de branchement exceptée.*

$$
\begin{aligned}
S &\to E \mid S\,B \mid B\,S \\
E &\to aSu \mid uSa \mid gSc \mid cSg \mid gSu \mid uSg \mid S_2\,S_2\,S_2 \\
S_2 &\to E_2 \mid B \mid S_2\,B \mid B\,S_2 \\
E_2 &\to aS_2u \mid uS_2a \mid gS_2c \mid cS_2g \mid gS_2u \mid uS_2g \\
B &\to a \mid c \mid g \mid u \mid Ba \mid Bc \mid Bg \mid Bu
\end{aligned}
$$

1.3 Analyse syntaxique des grammaires non-contextuelles

Dans la section précédente, nous nous sommes attachés à montrer qu'une grammaire non-contextuelle appropriée peut représenter un ensemble de structures secondaires sans pseudo-nœud d'un ARN, par l'intermédiaire des arbres de dérivation de l'ARN pour cette grammaire. Nous n'avions pas jusqu'alors discuté des moyens permettant d'obtenir ces arbres de dérivation. C'est précisément l'objet de cette section.

1.3.1 Intérêt des algorithmes tabulaires

Habituellement, un algorithme d'analyse syntaxique (ou analyseur syntaxique) est considéré performant lorsque la complexité de l'analyse dépend linéairement en temps et en espace de la longueur de la chaîne à analyser. La complexité d'analyse d'une chaîne suivant une grammaire non-contextuelle quelconque étant généralement non-linéaire, les analyseurs syntaxiques performants imposent des contraintes aux grammaires afin d'atteindre cette complexité linéaire. La contrainte la plus classique est l'appartenance à une classe particulière de grammaires non-ambiguës faciles à manipuler, comme les classes LL(k), LR(k) ou LALR(k). Lorsque la grammaire est ambiguë, des méthodes adaptées sont utilisées pour lever les ambiguïtés apparaissant en cours d'analyse. Dans tous les cas, ces analyseurs performants retournent un seul arbre de dérivation de la chaîne analysée : le premier arbre trouvé. Si la grammaire n'est pas ambiguë, cet arbre est forcément le bon puisqu'il est unique. Si la grammaire est ambiguë, l'utilisateur de l'algorithme d'analyse et l'auteur de la grammaire doivent s'arranger pour que ce premier arbre soit bien celui qu'ils souhaitent, ou pour que l'interprétation donnée à chacun des arbres possibles soit équivalente à celle de l'arbre obtenu. Un outil tel que YACC permet de générer automatiquement un analyseur syntaxique à partir d'une grammaire $LALR(1)$. Cet outil se révèle utilisable pour toute une classe de grammaires incluant les grammaires décrivant les langages usuels de programmation.

Les analyseurs générés par YACC appartiennent à la classe des analyseurs performants dont nous venons de discuter, et ils ne permettent de considérer qu'une unique dérivation, et donc une unique interprétation, de la chaîne analysée. S'il est important pour un langage de programmation de n'autoriser qu'une unique interprétation de chaque programme, ce comportement est à éviter absolument lorsque nous cherchons une structure secondaire parmi un très grand nombre de structures secondaires, et non pas la première structure secondaire venue, qui a très peu de chances d'être la bonne. De plus, qui dit multiplicité des structures secondaires ou multiplicité des arbres de dérivation dit ambiguïté des grammaires. Un algorithme d'analyse utile à la recherche de structures secondaires devra donc obligatoirement explorer les ambiguïtés plutôt que tenter de les éviter, de façon à trouver une structure secondaire satisfaisant un critère donné[5]. Cependant, même si nous devons imposer à notre analyseur de générer *implicitement* n'importe quel arbre de dérivation d'une chaîne suivant n'importe quelle grammaire non-contextuelle propre, nous ne pouvons attendre de lui une génération *explicite* de tous les arbres : le nombre d'arbres codant les structures secondaires d'un ARN augmente exponentiellement avec sa longueur.

Afin de satisfaire ces contraintes, l'analyseur que nous allons décrire appartient à la famille des *analyseurs tabulaires*. Ces analyseurs factorisent la forêt d'arbres de dérivation en partageant les sous-arbres communs, et ils peuvent donc coder l'ensemble des arbres de dérivation d'une chaîne en temps polynômial, même s'il faut toujours un temps exponentiel pour générer explicitement tous les arbres de dérivation de la forêt. Fort heureusement, nous verrons que le partage des sous-arbres communs rend inutile cette génération explicite. Les deux algorithmes tabulaires les plus connus sont celui de Cocke-Younger-Kasami (Aho & Ullman, 1972) et celui de Earley (Earley, 1970; Aho & Ullman, 1972). Dans le cas général, ces algorithmes ont une complexité de $O(n^3)$ en temps et $O(n^2)$ en espace, n étant la longueur de la chaîne analysée. L'algorithme de Cocke-Younger-Kasami (ou CYK) est un algorithme de type bottom-up, alors que l'algorithme de Earley est de type bottom-up avec un filtrage top-down. L'algorithme de Earley, contrairement à celui de Cocke-Younger-Kasami, a deux avantages fondamentaux : il ne nécessite pas de grammaire sous forme normale de Chomsky[6], et sa complexité diminue lorsque la grammaire utilisée possède de bonnes propriétés (cf. section 1.3.6 p. 35). Il est également conceptuellement plus simple que l'algorithme de Tomita (cf. section 5.1 p. 205).

Comme nos grammaires non-contextuelles seront systématiquement étendues en grammaires S-attribuées, il est essentiel d'utiliser un algorithme acceptant une grammaire quelconque, non nécessairement sous forme normale, car l'évaluation des attributs des non-terminaux selon une dérivation ne supporte pas forcément une normalisation explicite de la grammaire. Nous avons exclu l'algorithme CYK car, même s'il est possible de le modifier pour lui faire accepter n'importe quelle grammaire, il évalue au moins autant (et généralement beaucoup plus) de sous-arbres de dérivation que l'algorithme de Earley. Nous verrons dans le chapitre suivant qu'il est essentiel d'utiliser un algorithme exploitant au mieux les caractéristiques d'une grammaire, même pour des applications en biologie où le nombre d'ambiguïtés croît exponentiellement avec la taille des problèmes. L'algorithme que nous allons décrire possède, en plus des bonnes propriétés de l'algorithme de Earley, la propriété de ne jamais évaluer la

[5]Nous verrons plus loin avec les grammaires S-attribuées, ou SAG, que ce critère peut être exprimé comme une contrainte sur les attributs des sommets d'un arbre de dérivation (cf. définition 1.14 p. 42).

[6]Une grammaire est sous forme normale de Chomsky lorsque ses productions sont de la forme $A \to BC$ ou $A \to a$.

même sous-dérivation plus d'une fois. Il constituera une base performante sur laquelle viendra naturellement se greffer une stratégie d'évaluation et de sélection des arbres attribués pour les SAG (section suivante), puis un mécanisme de gestion simultanée (ou alignement) de plusieurs chaînes analysées avec les MTSAG (chapitre suivant). Nous appellerons *algorithme GCP* (pour *Generalized Common Prefix*) l'algorithme d'analyse syntaxique introduit dans la présente section, et *analyse GCP* toute analyse syntaxique effectuée avec cet algorithme. Certains des concepts utilisés par l'algorithme GCP peuvent être retrouvés dans d'autres algorithmes d'analyse syntaxique, parfois sous une forme sensiblement différente (Kruseman Aretz, 1989; Graham *et al.*, 1980) (cf. section 5.1 p. 205). Les algorithmes s'approchant le plus de l'algorithme GCP sont l'algorithme proposé par Voisin (Voisin, 1988) ou celui proposé par Nederhof (Nederhof, 1993).

1.3.2 Algorithme de Earley

Avant de présenter l'algorithme GCP, il est souhaitable de rappeler l'algorithme de Earley, dont l'algorithme GCP est inspiré, ainsi que certaines de ses propriétés essentielles.

Soit $G = (V_T, V_N, P, S)$ une grammaire non-contextuelle **propre**[7], et soit $a_1 \dots a_n$ appartenant à V_T^* la chaîne à analyser (ou chaîne d'entrée). Un objet de la forme $[A \to X_1 \dots X_k \bullet X_{k+1} \dots X_m, i]$ est appelé *état d'analyse* d'une chaîne $a_1 \dots a_n$ si $A \to X_1 \dots X_m$ et si l'entier i est compris entre 0 et n. Le point $\bullet$ est un métasymbole n'appartenant pas à V. L'entier k peut être n'importe quel nombre compris entre 0 (dans ce cas, $\bullet$ n'est précédé d'aucun symbole) et m (dans ce cas, $\bullet$ n'est suivi d'aucun symbole).

Pour chaque entier j tel que $0 \leqslant j \leqslant n$, nous allons construire une liste T_j d'états d'analyse telle que $[A \to \alpha \bullet \beta, i] \in T_j$ pour $0 \leqslant i \leqslant j$ si et seulement si l'état $[A \to \alpha \bullet \beta, i]$ est *a priori* utile :

Définition 1.3 (Utilité a priori d'un état d'analyse)
Un état $[A \to \alpha \bullet \beta, i]$ de T_j est a priori utile lorsqu'il existe γ tel que

$$S \to^* a_1 \dots a_i A\gamma \to a_1 \dots a_i \alpha\beta\gamma \to^* a_1 \dots a_j \beta\gamma.$$

La condition d'utilité a priori assure que tout état d'analyse de T_j est *utilisable* dans au moins une chaîne coïncidant avec $a_1 \dots a_n$ jusqu'à la position j. La condition d'utilité a priori d'un état d'analyse est plus faible que la condition d'utilité *a posteriori* qui assure que tout état d'analyse de T_j est *effectivement utilisé* dans une dérivation de $a_1 \dots a_n$:

Définition 1.4 (Utilité a posteriori d'un état d'analyse)
Un état $[A \to \alpha \bullet \beta, i]$ de T_j est a posteriori utile lorsqu'il existe γ tel que

$$S \to^* a_1 \dots a_i A\gamma \to a_1 \dots a_i \alpha\beta\gamma \to^* a_1 \dots a_j \beta\gamma \to^* a_1 \dots a_n.$$

[7] L'algorithme de Earley ne nécessite pas une CFG propre, mais sa présentation est plus simple lorsque la grammaire est propre, et l'algorithme GCP sera présenté dans le contexte des CFG propres.

Cette condition d'utilité a posteriori est impossible à satisfaire dans le cas général pour un algorithme d'analyse syntaxique lisant la chaîne de la gauche vers la droite. En ce sens, et dans le cas général, la condition d'utilité a priori satisfaite par l'algorithme de Earley est optimale. Notons que dans tous les cas, l'entier i d'un état $[A \to \alpha \,\centerdot\, \beta, i]$ et l'index j de la liste T_j à laquelle appartient cet état encadrent la portion $a_{i+1} \dots a_j$ de la chaîne analysée.

Définition 1.5 (position gauche, position droite)
Lorsqu'un non-terminal A est dérivable en une portion $a_{i+1} \dots a_j$ d'une chaîne analysée $a_1 \dots a_n$, alors l'entier i est appelé la **position gauche** *de A et j sa* **position droite**.

Les listes $T_0, \dots, T_n$ sont appelées les *listes d'analyse* pour la chaîne $a_1 \dots a_n$.

Nous pouvons maintenant décrire l'algorithme de Earley, qui génère les listes d'analyse pour n'importe quelle chaîne d'entrée.

Algorithme 1.1 (Earley)
Soient $G = (V_T, V_N, P, S)$ une grammaire non-contextuelle propre et $a_1 \dots a_n \in V_T^*$ une chaîne à analyser. L'algorithme construit les listes d'analyse $T_0, \dots T_n$, initialement vides, de la manière suivante.

1. *Si $S \to \alpha$, alors ajouter $[S \to \centerdot\alpha, 0]$ à T_0.*
2. *Soit $[A \to \centerdot B\beta, 0]$ un état de T_0. Pour toutes les productions $B \to \gamma$ de P, ajouter $[B \to \centerdot\gamma, 0]$ à T_0.*
3. *Répéter l'étape précédente tant que T_0 n'est pas stationnaire.*
4. *Soit $j = 1$.*
5. *Ajouter $[B \to \alpha a \,\centerdot\, \beta, i]$ à T_j s'il ne s'y trouve pas déjà, pour tous les états d'analyse $[B \to \alpha \,\centerdot\, a\beta, i]$ de T_{j-1} tels que $a = a_j$.*
6. *Soit $[A \to \gamma\centerdot, i]$ un état de T_j. Pour chaque état d'analyse $[B \to \alpha \,\centerdot\, A\beta, h]$ trouvé dans T_i, ajouter $[B \to \alpha A \,\centerdot\, \beta, h]$ à T_j s'il ne s'y trouve pas déjà.*
7. *Soit $[A \to \alpha \,\centerdot\, B\beta, i]$ un état de T_j. Pour toutes les productions $B \to \gamma$ de P, ajouter $[B \to \centerdot\gamma, j]$ à T_j s'il ne s'y trouve pas déjà.*
8. *Répéter les étapes 6 à 8 tant que T_j n'est pas stationnaire.*
9. *Incrémenter j et répéter les étapes 5 à 9 tant que j n'est pas égal à n.*

La chaîne $a_1 \dots a_n$ appartient à $L(G)$ si et seulement si il existe un état d'analyse $[S \to \alpha\centerdot, 0]$ dans T_n.

Nous ne montrerons pas ici que les états d'analyse construits par l'algorithme précédent sont effectivement a priori utiles. Le lecteur intéressé pourra consulter (Aho & Ullman, 1972) ou la démonstration du théorème 1.1 p. 28. L'algorithme ainsi présenté est un algorithme de reconnaissance qui ne construit pas directement d'arbre d'analyse.

Proposition 1.1 (Complexités de l'algorithme de Earley)
L'algorithme de Earley a en général une complexité de reconnaissance de $O(n^3)$ en temps et $O(n^2)$ en espace. Lorsque la grammaire est non-ambiguë, les complexités tombent à $O(n^2)$ en temps et en espace. Lorsque la grammaire est $LR(k)$ non-récursive à droite, alors les complexité sont de $O(n)$ en temps et en espace.

Ces résultats sont démontrés dans (Aho & Ullman, 1972) et (Earley, 1968). Nous verrons plus loin que le cas des grammaires $LR(k)$ n'est pas à négliger, même si ces grammaires semblent dans un premier temps inapplicables à nos problèmes puisqu'étant non-ambiguës. Il existe une transformation de l'algorithme de Earley permettant une exécution en temps linéaire pour n'importe quelle grammaire $LR(k)$ sans utiliser de "lookahead" (Leo, 1991). Mais cette transformation n'est pas applicable à nos problèmes car elle ne permet pas l'exécution de fonctions d'évaluation d'attributs au fur et à mesure de l'analyse syntaxique (cf. section 5.1 p. 205).

1.3.3 Version ensembliste de l'algorithme GCP

Dans cette section, nous allons montrer comment factoriser les états d'analyse de l'algorithme de Earley afin de diminuer le nombre d'états manipulés par l'algorithme d'analyse. Nous appellerons algorithme GCP (pour Generalized Common Prefix) l'algorithme utilisant ces états factorisés.

Les états d'analyse générés par l'algorithme de Earley sont séparables en deux catégories :

- les états dits "du noyau"[8], de la forme $[A \to \alpha \centerdot \beta, i]$ avec $\alpha \neq \epsilon$,
- les états n'appartenant pas au noyau, de la forme $[A \to \centerdot\beta, i]$.

Intuitivement, tous les états n'appartenant pas au noyau sont immédiatement déductibles des états du noyau par une étape dite de clôture. Cela explique la terminologie d'états du noyau utilisée pour qualifier les états $[A \to \alpha \centerdot \beta, i]$ avec $\alpha \neq \epsilon$ car aucun état du noyau ne peut être supprimé sans perte d'information, alors que ce n'est pas le cas pour les états n'appartenant pas au noyau. Dans le cas présent, les états n'appartenant pas au noyau sont générés par les étapes 2 et 7 de l'algorithme de Earley donné ci-dessus.

Si un non-terminal A apparaît dans un état $[A \to \centerdot\alpha, j]$ de T_j, alors toutes les règles de G ayant A pour non-terminal à gauche auront été utilisées pour générer des états de la forme $[A \to \centerdot\alpha, j]$, y compris les règles mutuellement exclusives comme $A \to a\beta$ et $A \to a'\beta'$ (la chaîne analysée ne peut contenir à la fois a et a' en position $j+1$). En fait, les états $[A \to \centerdot\alpha, j]$ sont ajoutés à T_j uniquement pour signifier aux étapes suivantes de l'algorithme de Earley que le non-terminal A est attendu à l'étape j pour faire progresser l'analyse. Pour diminuer sensiblement le nombre d'états à stocker dans les listes d'analyse, il suffit de ne générer que les états du noyau et de noter les non-terminaux attendus plutôt que de générer les états n'appartenant pas au noyau.

Une autre manière de diminuer le nombre d'états d'analyse stockés dans les listes d'analyse consiste à remplacer tous les états $[A_1 \to \alpha \centerdot \beta_1, i], \ldots, [A_k \to \alpha \centerdot \beta_k, i]$ appartenant à T_j, et partageant le même préfixe α et le même index i, par un seul état $\big[\{A_1, \ldots A_k\} \rightarrowtail \alpha, i]$ (la relation $\to$ a été remplacée par $\rightarrowtail$ pour indiquer que α n'est qu'un facteur gauche d'un ensemble de parties droites de règles de G).

[8]Voir (Aho *et al.*, 1988) p.223 pour une introduction de la notion de "kernel item".

La formalisation des deux remarques précédentes nous mène à une première version de l'algorithme GCP.

Définition 1.6 (État d'analyse)
Soit $G = (V_T, V_N, P, S)$ une grammaire non-contextuelle propre. Nous appellerons état d'analyse tout objet de la forme $\big[\{A_1, \dots, A_k\} \rightarrowtail \alpha, i\big]$ tel que $k > 0$ et tel que pour tout A_j $(1 \leqslant j \leqslant k)$, il existe β tel que $A_j \to \alpha\beta$.

On utilisera le symbole Δ (ainsi que ses dérivés) pour désigner tout ensemble de non-terminaux. Un état d'analyse pourra donc être noté $[\Delta \rightarrowtail \alpha, i]$. Les états d'analyse seront plus simplement appelés états lorsqu'aucune confusion ne sera possible. On appellera également état d'analyse, ou état, tout objet $[\Delta \rightarrowtail \alpha]$ résultant de la suppression de l'index i d'un état d'analyse $[\Delta \rightarrowtail \alpha, i]$.

Définition 1.7 (Liste d'analyse)
Soit $a_1 \dots a_n \in V_T^*$ une chaîne à analyser. L'algorithme GCP construit n listes d'analyse $T_1, \dots, T_n$ pour cette chaîne.

Une liste d'analyse T_j est une liste d'états d'analyse $[\Delta \rightarrowtail \alpha, i]$ construits de manière à vérifier au minimum la condition d'utilité a priori suivante : pour tout état $[\Delta \rightarrowtail \alpha, i] \in T_j$ et pour tout $A \in \Delta$, il existe β et γ tels que

$$S \to^* a_1 \dots a_i A\gamma \to a_1 \dots a_i \alpha\beta\gamma \to^* a_1 \dots a_j \beta\gamma.$$

La chaîne $a_1 \dots a_n$ appartient à $L(G)$ si et seulement si il existe $[\Delta \rightarrowtail \alpha, 0] \in T_n$ tel que $S \in \Delta$ et $S \to \alpha$.

Comme pour l'algorithme de Earley, la condition d'utilité a priori précédente signifie qu'un état d'analyse n'est jamais ajouté à une liste d'analyse T_j s'il est impossible à utiliser dans un arbre de dérivation d'une chaîne dont la partie déjà analysée est $a_1 \dots a_j$. Cette propriété est la clef des diminutions des complexités d'analyse constatées lorsque les grammaires vérifient de bonnes propriétés.

Pour savoir quels non-terminaux A de l'ensemble Δ d'un état $[\Delta \rightarrowtail \alpha]$ forment avec α une production complète $A \to \alpha$, nous définissons une fonction Reduce.

Définition 1.8 (Reduce)
Nous noterons Reduce la fonction retournant l'ensemble des non-terminaux réductibles d'un état $[\Delta \rightarrowtail \alpha]$:

$$\mathsf{Reduce}([\Delta \rightarrowtail \alpha]) = \Big\{ A \in \Delta \;\Big|\; A \to \alpha \Big\}.$$

Pour supprimer le calcul des états n'appartenant pas au noyau dans l'algorithme de Earley, nous calculerons, à l'aide de la fonction Expect suivante, l'ensemble des non-terminaux attendus par chaque état de l'algorithme GCP, et nous effectuerons l'union Δ_j de ces ensembles pour tous les états d'analyse de T_j.

Définition 1.9 ($\mathsf{Expect}, \Delta_j$)
Nous noterons Expect *la fonction retournant l'ensemble des non-terminaux attendus par un état* $[\Delta \rightarrowtail \alpha]$*:*

$$\mathsf{Expect}\big([\Delta \to \alpha]\big) = \Big\{ D \in V_N \;\Big|\; \exists B, \beta, \gamma, \exists A \in \Delta, A \to \alpha B \beta \;\wedge\; B \to^* D\gamma \Big\}.$$

Une fois T_j *construit, l'ensemble des non-terminaux attendus par au moins un état de* T_j *sera noté* Δ_j *ou* $\mathsf{Expect}(T_j)$*. Cet ensemble est défini pour* $j \geqslant 1$ *par :*

$$\Delta_j = \mathsf{Expect}(T_j) = \bigcup_{[\Delta \to \alpha, i] \in T_j} \mathsf{Expect}\big([\Delta \to \alpha]\big).$$

Notons que, dans l'algorithme de Earley, T_0 *ne contient que des états n'appartenant pas au noyau.* Δ_0 *est donc calculé de la manière suivante :*

$$\Delta_0 = \big\{ A \in V_N \mid \exists \beta, S \to^* A\beta \big\}.$$

L'ensemble Δ_j peut éventuellement être vide, mais Δ_0 n'est jamais vide et contient au moins S.

Il ne faut pas confondre l'ensemble de non-terminaux retourné par Expect et l'ensemble de terminaux retourné par la fonction Follow, classique en analyse syntaxique (Aho & Ullman, 1972).

Dans l'algorithme de Earley, la prolongation d'états (i.e. le déplacement vers la droite du symbole **.**) par des terminaux ou des non-terminaux se fait soit à partir d'états du noyau, soit à partir d'états n'appartenant pas au noyau. Nous noterons Goto la fonction effectuant la prolongation des états d'analyses de l'algorithme GCP, sachant que ces états sont équivalents à une factorisation des états d'analyse du noyau qui auraient été générés par l'algorithme de Earley.

Définition 1.10 (Goto)
L'état d'analyse résultant de l'extension (ou de la prolongation) d'un état d'analyse $[\Delta \rightarrowtail \alpha]$ *par un symbole* X *de la grammaire est obtenu à l'aide de la fonction* Goto *définie comme suit :*

$$\mathsf{Goto}\big([\Delta \rightarrowtail \alpha], X\big) = \begin{cases} [\Delta' \rightarrowtail \alpha X] & \textit{si } \Delta' = \{A \in \Delta \mid \exists \beta, A \to \alpha X \beta\} \textit{ est non vide,} \\ \textit{non-définie} & \textit{sinon.} \end{cases}$$

Enfin, la prolongation des états n'appartenant pas au noyau est remplacée par une création d'états d'analyse à partir d'un ensemble de non-terminaux attendus. Nous utiliserons à cet effet la fonction $\mathsf{Initial}$.

Définition 1.11 ($\mathsf{Initial}$)
L'état d'analyse résultant de l'extension d'un ensemble de non-terminaux attendus Δ *par un symbole* X *de la grammaire est obtenu à l'aide de la fonction* $\mathsf{Initial}$ *définie comme suit :*

$$\mathsf{Initial}(\Delta, X) = \begin{cases} [\Delta' \rightarrowtail X] & \textit{si } \Delta' = \{A \in \Delta \mid \exists \beta, A \to X \beta\} \textit{ est non vide,} \\ \textit{non-définie} & \textit{sinon.} \end{cases}$$

Dans son principe, l'algorithme GCP peut être considéré comme une adaptation de l'algorithme de Earley, où les états n'appartenant pas au noyau ont été remplacés par des ensembles de non-terminaux attendus, et où les états du noyaux ont été factorisés suivant la chaîne précédant le '$\bullet$'. Nous obtenons alors l'algorithme suivant :

Algorithme 1.2 (Algorithme GCP ensembliste)
Soient $G = (V_T, V_N, P, S)$ une grammaire non-contextuelle propre et $a_1 \dots a_n \in V_T^*$ une chaîne à analyser. Soit $\Delta_0 = \{A \in V_N \mid \exists \beta, S \to^* A\beta\}$. Au départ, les listes d'analyse T_j $(1 \leqslant j \leqslant n)$ sont vides. Pour j allant de 1 à n, exécuter les étapes suivantes :

1. *Création des états ne comportant qu'un terminal à droite :*
   $$T_j := T_j \cup \Big\{[\Delta \rightarrowtail a_j, j-1] \;\Big|\; [\Delta \rightarrowtail a_j] = \mathsf{Initial}(\Delta_{j-1}, a_j)\Big\}$$
2. *Prolongement d'états par un terminal :*
   $$T_j := T_j \cup \Big\{[\Delta \rightarrowtail \alpha a_j, i] \;\Big|\; \exists [\Delta' \rightarrowtail \alpha, i] \in T_{j-1}, [\Delta \rightarrowtail \alpha a_j] = \mathsf{Goto}([\Delta' \rightarrowtail \alpha], a_j)\Big\}$$
3. *Création des états ne comportant qu'un non-terminal à droite :*
   $$T_j := T_j \cup \Big\{[\Delta \rightarrowtail A, i] \;\Big|\; \exists [\Delta'' \rightarrowtail \beta, i] \in T_j, A \in \mathsf{Reduce}([\Delta'' \rightarrowtail \beta]) \;\wedge\; [\Delta \rightarrowtail A] = \mathsf{Initial}(\Delta_i, A)\Big\}$$
4. *Prolongement d'états par un non-terminal :*
   $$T_j := T_j \cup \Big\{[\Delta \rightarrowtail \alpha A, h] \;\Big|\; \exists [\Delta' \rightarrowtail \alpha, h] \in T_i,\ \exists [\Delta'' \rightarrowtail \beta, i] \in T_j, A \in \mathsf{Reduce}([\Delta'' \rightarrowtail \beta]) \;\wedge\; [\Delta \rightarrowtail \alpha A] = \mathsf{Goto}([\Delta' \rightarrowtail \alpha], A)\Big\}$$
5. *Répéter les étapes 3 à 5 tant que T_j n'est pas stationnaire ;*
6. *Calculer $\Delta_j := \mathsf{Expect}(T_j)$.*

La chaîne $a_1 \dots a_n$ appartient à $L(G)$ si et seulement si il existe un état d'analyse $[\Delta \rightarrowtail \alpha, 0] \in T_n$ tel que $S \in \mathsf{Reduce}([\Delta \rightarrowtail \alpha])$. On dit alors que la chaîne a été reconnue par l'algorithme.

Théorème 1.1 (Utilité a priori des états générés par l'algorithme 1.2)
Lorsque les listes d'analyse T_j sont construites comme décrit dans l'algorithme 1.2, les deux propositions suivantes sont équivalentes :

1. $[\Delta \rightarrowtail \alpha, i] \in T_j$
2. $\forall A \in \Delta, \exists (\beta, \gamma), S \to^* a_1 \dots a_i A\gamma \to a_1 \dots a_i \alpha\beta\gamma \to^* a_1 \dots a_j \beta\gamma$

Démonstration : la démonstration s'inspire de celle de l'algorithme de Earley donnée dans (Aho & Ullman, 1972).

Nous allons démontrer que $1 \Longrightarrow 2$ par récurrence sur les états ajoutés. Remarquons tout d'abord que, par construction de $[\Delta \rightarrowtail \alpha]$, $\forall A \in \Delta, \exists \beta, A \to \alpha\beta$. Au départ, les listes d'analyse sont vides et satisfont l'hypothèse de récurrence. Supposons maintenant que tous les états déjà ajoutés par l'algorithme 1.2 satisfont l'hypothèse de récurrence. Un nouvel état ajouté à T_j se présente donc sous une des quatre formes décrites dans les étapes de l'algorithme :

1. $[\Delta \rightarrowtail a_j, j-1]$. Si $A \in \Delta$, alors par construction, $A \in \Delta_{j-1}$ et, si $j > 1$, il existe $[\Delta' \rightarrowtail \alpha, h] \in T_{j-1}$ tel que $A \in \mathsf{Expect}([\Delta' \rightarrowtail \alpha])$. Donc, par hypothèse de récurrence sur $[\Delta' \rightarrowtail \alpha]$ et par définition de Expect, il existe γ tel que $S \to^* a_1 \dots a_{j-1} A\gamma$. Comme on sait déjà qu'il exite β tel que $A \to a_j\beta$, l'hypothèse de récurrence est vérifiée par $[\Delta \rightarrowtail a_j, j-1]$ dans le cas où $j > 1$. Le cas $j = 1$ se traite de manière équivalente à partir de la définition de Δ_0 ;

2. $[\Delta \rightarrowtail \alpha a_j, i]$. Par construction, il existe $[\Delta' \rightarrowtail \alpha, i] \in T_{j-1}$ tel que $A \in \Delta' \supset \Delta$. Donc il existe β et γ tels que $S \to^* a_1 \dots a_i A\gamma \to a_1 \dots a_i \alpha a_j \beta\gamma \to^* a_1 \dots a_j \beta\gamma$. L'hypothèse de récurrence est vérifiée par $[\Delta \rightarrowtail \alpha a_j, i]$;

3. $[\Delta \rightarrowtail A, i]$. Par construction, il existe $[\Delta'' \rightarrowtail \beta, i] \in T_j$ tel que $A \in \mathsf{Reduce}([\Delta'' \rightarrowtail \beta])$, donc $A \to^* a_{i+1} \dots a_j$ par hypothèse de récurrence. Soit $B \in \Delta$. Alors $B \in \Delta_i$ et, si $i > 0$ il existe $[\Delta' \rightarrowtail \alpha, h] \in T_i$ tel que $B \in \mathsf{Expect}([\Delta' \rightarrowtail \alpha])$. Donc il existe γ tel que $S \to^* a_1 \dots a_i B\gamma$. Le cas $i = 0$ se traite de manière équivalente en revenant à la définition de Δ_0. L'hypothèse de récurrence est vérifiée par $[\Delta \rightarrowtail A, i]$;

4. $[\Delta \rightarrowtail \alpha A, h]$. Par construction, il existe $[\Delta' \rightarrowtail \alpha, h] \in T_i$ tel que $\forall B \in \Delta, B \in \Delta'$. Donc il existe β et γ tels que $S \to^* a_1 \dots a_h B\gamma \to a_1 \dots a_h \alpha A\beta\gamma \to^* a_1 \dots a_i A\beta\gamma$. À l'aide du même raisonnement que dans le cas précédent, on montre que $A \to^* a_{i+1} \dots a_j$. Donc l'hypothèse de récurrence est vérifiée par $[\Delta \rightarrowtail \alpha A, h]$.

Pour démontrer que $2 \Longrightarrow 1$, nous procédons également par récurrence. Nous introduisons à cet effet un multiplet (A, α, β, i, j) extrait des termes de la proposition 2 du théorème. Un multiplet (A, α, β, i, j) existe lorsque $\exists\gamma, S \to^* a_1 \dots a_i A\gamma \to a_1 \dots a_i \alpha\beta\gamma \to^* a_1 \dots a_j \beta\gamma$. Pour démontrer que $2 \Longrightarrow 1$, il suffit de démontrer par récurrence sur les multiplets que

$$(A, \alpha, \beta, i, j) \text{ existe } \Longrightarrow \exists[\Delta \rightarrowtail \alpha, i] \in T_j, A \in \Delta$$

Pour pouvoir appliquer une démonstration par récurrence, il faut introduire une relation d'ordre sur les multiplets. Nous dirons que $(A, \alpha, \beta, i, j) < (A', \alpha', \beta', i', j')$ si

- $j < j'$ ou
- $j = j'$ et $i' < i$ ou
- $j = j'$ et $i' = i$ et la longueur de la plus courte dérivation $\alpha \to^* a_{i+1} \dots a_j$ est strictement inférieure à la longueur de la plus courte dérivation $\alpha' \to^* a_{i+1} \dots a_j$ ou

- on utilisera un critère arbitraire[9] sur β et A pour départager les multiplets égaux selon les critères précédents.

Nous pouvons maintenant écrire une démonstration par récurrence sur les multiplets. Nous devons distinguer 4 formes de multiplets :

1. (A, a_j, β, i, j). On a forcément $i = j - 1$. Si $j = 1$, cela signifie que $A \in \Delta_0$, donc il existe $[\Delta \rightarrowtail a_1] \in T_1$ tel que $A \in \Delta$ (étape 1 de l'algorithme).

   Si $j > 1$, il existe $\alpha, \alpha', B, D, \beta, \beta', \beta''$ avec $\alpha' \to^* a_{k+1} \dots a_{j-1} \neq \epsilon$ tels que :

   $$S \to^* \alpha B \beta \to \alpha\alpha' D \beta' \beta \to^* \alpha\alpha' A \beta'' \beta' \beta \to^* a_1 \dots a_k a_{k+1} \dots a_{j-1} A \beta'' \beta' \beta$$

   Donc $(B, \alpha', D\beta', k, j-1)$ existe et précède (A, a_j, β, i, j). Donc il existe $[\Delta' \rightarrowtail \alpha', k] \in T_{j-1}$ tel que $B \in \Delta'$ et $A \in \mathsf{Expect}([\Delta' \rightarrowtail \alpha'])$. Donc $A \in \Delta_{j-1}$. Donc il existe $[\Delta \rightarrowtail a_j] \in T_j$ tel que $A \in \Delta$ (étape 1 de l'algorithme).

2. $(A, \alpha a_j, \beta, i, j)$. Le multiplet $(A, \alpha, a_j\beta, i, j-1)$ existe donc et précède le multiplet $(A, \alpha a_j, \beta, i, j)$. Donc il existe $[\Delta' \rightarrowtail \alpha, i] \in T_{j-1}$ tel que $A \in \Delta'$. Donc il existe $[\Delta \rightarrowtail \alpha a_j, i] \in T_j$ tel que $A \in \Delta$ (étape 2 de l'algorithme).

3. (B, A, β, i, j). Par un raisonnement similaire à celui utilisé pour le multiplet (A, a_j, β, i, j), on montre aisément que $B \in \Delta_i$. Soit γ tel que la dérivation $A \to \gamma \to^* a_{i+1} \dots a_j$ soit la plus courte possible. Comme $\exists \beta', B \to A\beta'$, on en déduit que $(A, \gamma, \epsilon, i, j)$ existe et précède (B, A, β, i, j). Donc il existe $[\Delta'' \rightarrowtail \gamma, i] \in T_j$ avec $A \in \mathsf{Reduce}([\Delta'' \rightarrowtail \gamma])$. Donc il existe $[\Delta \rightarrowtail A, i] \in T_j$ tel que $B \in \Delta$ (étape 3 de l'algorithme).

4. $(B, \alpha A, \beta, i, j)$. On remarque facilement qu'il existe k $(i < k < j)$ tel que $(B, \alpha, A\beta, i, k)$ existe et précède $(B, \alpha A, \beta, i, j)$. Donc il existe $[\Delta' \rightarrowtail \alpha, i] \in T_k$ prolongeable par A tel que $B \in \Delta'$, donc $A \in \Delta_k$. Soit α' tel que $A \to \alpha' \to^* a_{k+1} \dots a_j$. Le multiplet $(A, \alpha', \epsilon, k, j)$ existe et précède $(B, \alpha A, \beta, \gamma, i, j)$. Donc il existe $[\Delta'' \rightarrowtail \alpha', k] \in T_j$ tel que $A \in \Delta''$ et $A \in \mathsf{Reduce}([\Delta'' \rightarrowtail \alpha'])$. Donc (étape 4 de l'algorithme), il existe $[\Delta \rightarrowtail \alpha A, i] \in T_j$ tel que $B \in \Delta$.

L'implication $2 \Longrightarrow 1$ est démontrée. □

La correction de l'algorithme 1.2 est une conséquence du théorème 1.1 : toute dérivation de la chaîne initiale se met sous la forme donnée dans la deuxième proposition du théorème. Cette dérivation est alors représentée par un état d'analyse d'une liste d'analyse. Le théorème montre également que si $[\Delta \rightarrowtail \alpha, 0]$ appartient à T_n et si $S \in \mathsf{Reduce}([\Delta \rightarrowtail \alpha])$, alors $S \to \alpha \to^* a_1 \dots a_n$, c'est-à-dire $a_1 \dots a_n \in L(G)$. Il suffit maintenant de constater que l'algorithme 1.2 se termine obligatoirement (du fait du nombre fini d'états $[\Delta \rightarrowtail \alpha, i]$ avec $i \leqslant j$ pouvant être ajoutés à T_j) pour achever la démonstration de la correction de l'algorithme.

[9]On pourra par exemple définir une relation d'ordre total sur les symboles de la grammaire et utiliser l'ordre lexicographique sur βA et $\beta' A'$ pour les départager.

1.3.4 L'algorithme GCP en pratique

La description de l'algorithme 1.2 n'est pas très directive puisqu'elle comporte quatre étapes à répéter *tant que T_j n'est pas stationnaire*. De plus, chaque étape est écrite dans une syntaxe ensembliste préjudiciable à une évaluation rapide. Il est souhaitable, lors d'une mise en œuvre réelle de l'algorithme, d'éviter de perdre du temps à envisager des étapes s'avérant finalement inutiles. Nous allons donc réécrire l'algorithme 1.2 sous une forme moins ensembliste plus adaptée à la programmation. Notons que ce désir d'une approche plus impérative n'est pas nouveau : bien que la description de l'algorithme de Earley donnée p. 24 utilise elle aussi une syntaxe ensembliste, il en existe d'autres présentations beaucoup plus propices à une programmation efficace (Graham *et al.*, 1980; Kruseman Aretz, 1989) (voir également la section 5.1 p. 205). Tout d'abord, quelques remarques s'imposent :

- les étapes 1 et 3, respectivement 2 et 4, ont une forme très semblable,
- les étapes 1 et 2 ne sont effectuées qu'une seule fois puisqu'elles dépendent de a_j, Δ_{j-1} et T_{j-1}, trois valeurs connues et constantes lorsque j est fixé,
- l'étape 3 est à effectuer pour tous les **états d'analyse réductibles** de T_j, c'est-à-dire tous les états $[\Delta'' \rightarrowtail \beta, i]$ tels que $\mathsf{Reduce}\big([\Delta'' \rightarrowtail \beta]\big) \neq \emptyset$. Idéalement, on doit appliquer l'étape 3 à tout état réductible venant d'être ajouté à T_j par une des quatre étapes ;
- l'étape 4 est à considérer uniquement si un état réductible $[\Delta'' \rightarrowtail \beta, i]$ a été ajouté à T_j. Dans un tel cas, tous les non-terminaux issus de la réduction de cet état doivent être utilisés pour tenter de prolonger un état de T_i ;
- un état $[\Delta \rightarrowtail \alpha, i]$ ajouté à T_j ne dépend des états situés dans T_k ($k \leqslant i$) que par l'intermédiaire de Δ_i. L'ajout des états dans T_j peut donc se faire dans l'ordre des i décroissants.

Les cinq remarques précédentes nous conduisent à introduire une version itérative de l'algorithme 1.2, les itérations étant déterminées par les symboles créant ou prolongeant les états. Il suffit pour cela de prendre en compte les états réductibles $[\Delta \rightarrowtail \alpha, i]$ dès qu'ils sont ajoutés à T_j, en insérant dans une file, que nous appellerons Q, les paires (A, i) telles que $A \in \mathsf{Reduce}\big([\Delta \rightarrowtail \alpha]\big)$. La file Q contient initialement la paire $(a_j, j-1)$. Une paire (X, i) stockée dans Q pendant le calcul de T_j symbolise l'ensemble des dérivations $X \rightarrow^* a_{i+1} \ldots a_j$. L'argument de l'itération est le suivant: tant que Q n'est pas vide, retirer une paire de la liste puis l'utiliser pour créer ou prolonger des états d'analyse. Afin d'éviter d'ajouter à Q une paire (X, i) qui aurait été extraite précédemment (au cours du calcul du même T_j), une relation d'ordre est introduite sur les paires :

- si (X, i) et (X', i') appartiennent à Q, et si $i' < i$, alors la paire (X, i) doit être extraite de Q avant la paire (X', i'), car (X, i) peut entraîner la création d'une paire (X', i'), l'inverse étant impossible,
- si (X, i) et (X', i) appartiennent à Q, et si $X' \rightarrow^* X$, alors la paire (X, i) doit être extraite de Q avant la paire (X', i'), pour la même raison que précédemment.

La structure en file de priorité de Q n'est pas essentielle au bon fonctionnement de l'algorithme, même si elle facilite la preuve d'arrêt. Une simple liste de type premier entré, premier sorti, aurait été suffisante. Mais nous verrons après l'introduction des arbres attribués que la structure de Q est essentielle au bon fonctionnement du mécanisme de sélection des arbres de dérivation (cf. discussion p. 44). La définition de la file de priorité Q est donc la suivante.

Définition 1.12 (File de priorité sur les paires (X, i))
Soit Q un ensemble de paires $(X, i) \in V \times \{1, \ldots, n\}$. Soient (X, i) et (X', i') deux éléments de Q. Nous dirons que $(X', i') < (X, i)$, ou que l'extraction de (X, i) est prioritaire sur celle de (X', i') si $i' < i$ ou si $i' = i$ et $X' \to^* X$. Nous noterons $\mathsf{Extract}$ *la fonction utilisée pour simultanément retourner et supprimer une paire maximum (X, i) de Q. Nous noterons cette extraction $(X, i) := \mathsf{Extract}(Q)$.*

Remarque 1.4 la grammaire G étant non-circulaire, la relation $<$ définie précédemment est bien une relation d'ordre. Telle que la relation est définie, il ne s'agit que d'un ordre partiel. Mais cet ordre est bien sûr aisément prolongeable en ordre total.

Une présentation plus formelle des discussions précédentes conduit à la version suivante de notre algorithme d'analyse syntaxique. Cette version sera la version de référence lorsque nous parlerons d'algorithme GCP.

Algorithme 1.3 (Algorithme GCP avec file de priorité)
Soit Q la file de priorité qui vient d'être définie, avec sa fonction $\mathsf{Extract}$. *Au départ, les listes d'analyses T_j sont vides. Soit $\Delta_0 = \{A \in V_N \mid \exists\beta, S \to^* A\beta\}$. Pour j allant de 1 à n, exécuter les étapes suivantes :*

1. *Initialiser la file Q :* $\quad Q := \{(a_j, j-1)\}$
2. *Extraction de la paire à traiter en priorité :*
   $(X, i) := \mathsf{Extract}(Q)$
3. *Création des états d'analyse commençant par X :*
   $T_j := T_j \cup \Big\{[\Delta \rightarrowtail X, i] \Bigm| [\Delta \rightarrowtail X] = \mathsf{Initial}(\Delta_i, X)\Big\}$
4. *Prolongement d'états d'analyse par X :*
   $T_j := T_j \cup \Big\{[\Delta \rightarrowtail \alpha X, h] \Bigm| \exists[\Delta' \rightarrowtail \alpha, h] \in T_i, [\Delta \rightarrowtail \alpha X] = \mathsf{Goto}\big([\Delta' \rightarrowtail \alpha], X\big)\Big\}$
5. *Pour chaque état d'analyse $[\Delta \rightarrowtail \alpha, h]$ ajouté à T_j par une des étapes 3 et 4 ci-dessus, et pour chaque $A \in \mathsf{Reduce}\big([\Delta \rightarrowtail \alpha]\big)$, ajouter la paire (A, h) à Q si elle n'est pas déjà dans Q.*
6. *Répéter les étapes 2 à 6 tant que Q n'est pas vide ;*
7. *Calculer $\Delta_j := \mathsf{Expect}(T_j)$.*

La chaîne $a_1 \ldots a_n$ appartient à $L(G)$ si et seulement si la paire $(S, 0)$ a été extraite de Q pendant le calcul de T_n. On dit alors que la chaîne a été reconnue par l'algorithme.

L'algorithme précédent est correct car tout état d'analyse ajouté à T_j aurait également été ajouté par l'algorithme 1.2, et réciproquement. L'algorithme se termine car la relation d'ordre introduite sur les triplets extraits de Q empêche l'ajout d'une paire de priorité supérieure à celles déjà extraites.

Nous aurions pu donner à l'algorithme 1.3 la même condition de reconnaissance d'une chaîne de $L(G)$ que dans l'algorithme 1.2 ($a_1 \dots a_n \in L(G) \Longleftrightarrow \exists [\Delta \rightarrowtail \alpha, 0] \in T_n, S \in \mathsf{Reduce}([\Delta \rightarrowtail \alpha])$). Mais dans la perspective de l'introduction des attributs, il est préférable de transformer cette condition comme nous l'avons fait, c'est-à-dire en condition sur l'extraction des paires.

1.3.5 Reconstruction de l'arbre d'analyse

L'algorithme GCP est un **algorithme de reconnaissance**, c'est-à-dire qu'il permet de savoir si une chaîne donnée appartient au langage généré par la grammaire; il ne donne pas directement d'arbre d'une dérivation $S \rightarrow^* a_1 \dots a_n$. Cependant, les listes d'analyse contiennent suffisamment d'information pour permettre la reconstruction de n'importe quel arbre de dérivation de la chaîne d'entrée[10].

Nous allons montrer ici qu'il est en fait possible de reconstruire un arbre d'analyse uniquement à partir de la mémorisation de l'ensemble des paires (X, i) extraites de Q au cours de la reconnaissance. Lorsqu'une paire (X, i) est extraite de Q pendant le calcul de T_j, son apparition est mémorisée dans une table particulière $(\mathfrak{T}_{i,j})_{1 \leqslant i < j \leqslant n}$, initialement vide :

$$\mathfrak{T}_{i,j} := \mathfrak{T}_{i,j} \cup \{X\}$$

Le coût de stockage de cette table et de ses éléments est $O(n^2)$ a priori[11], mais il est possible d'implémenter la table de façon à ramener le coût à $O(n)$ lorsque $O(n)$ états d'analyse ont été créés et donc $O(n)$ réductions faites, ce qui est le cas avec certaines grammaires (cf. section 1.3.6).

Notons $\mathfrak{R}$ la fonction qui à un triplet (A, i, j) associe un arbre de dérivation de $A \rightarrow^* a_{i+1} \dots a_j$. Cette fonction se calcule comme suit :

1. Recherche d'une dérivation valide $A \rightarrow u_{-1} B_0 u_0 \dots B_m u_m \rightarrow^* a_{i+1} \dots a_j$, avec $B_k \rightarrow^* a_{s_k+1} \dots a_{e_k}$. Cette dérivation détermine la racine de l'arbre et ses fils.

2. Si $m \neq -1$ (donc si $u_{-1} \neq a_{i+1} \dots a_j$), appeler récursivement $\mathfrak{R}(B_k, s_k, e_k)$ ($0 \leqslant k \leqslant m$) pour connaître le reste de l'arbre.

---

[10] Nous pourrions rendre la construction d'arbres de dérivation beaucoup plus simple en ajoutant à chaque état d'analyse une liste de pointeurs indiquant de quels états d'analyse il provient. En pratique, un tel ajout de pointeurs augmente très significativement la place prise en mémoire par l'algorithme GCP sans améliorer notablement les performances d'analyse puisque la quasi totalité du temps d'analyse est consacrée à la construction des listes d'analyse, et non pas à la reconstruction d'un quelconque arbre de dérivation.

[11] En fait, la table $\mathfrak{T}_{i,j}$ s'apparente aux tables d'analyse construites par l'algorithme CYK.

Proposition 1.2 (Complexité de reconstruction de l'arbre d'analyse) *Soit r le nombre maximum de non-terminaux apparaissant dans les parties droites α des règles $A \to \alpha$ de G. Alors $\mathfrak{R}(S, 0, n)$ retourne un arbre de dérivation de $a_1 \dots a_n$ en temps $O(n^r)$.*

Démonstration : Nous allons montrer que

$$\text{Temps}\big(\mathfrak{R}(A, i, j)\big) = O\big((j-i)^{r-1}\big) + \sum_{k=0}^{m} \text{Temps}\big(\mathfrak{R}(B_k, s_k, e_k)\big),$$

ce qui est suffisant pour assurer $\text{Temps}\big(\mathfrak{R}(A, i, j)\big) = O\big((j-i)^r\big)$.

Soit $A \to u_{-1}B_0u_0 \dots B_m u_m$ une règle de grammaire ayant A pour non-terminal à gauche. Remarquons que :

- il y a un nombre fini de telles règles,
- $m \leqslant r - 1$;
- il y a $O\Big(\big(j - i - (|u_{-1}| + \cdots + |u_m|)\big)^m\Big)$ manières de décomposer cette règle entre a_{i+1} et a_j (On note $|u_i|$ le nombre de terminaux de u_i). En effet, le système :

$$\begin{array}{ll} B_0 \to^* a_{s_0+1} \dots a_{e_0} & s_0 = i + |u_{-1}| \\ B_1 \to^* a_{s_1+1} \dots a_{e_1} & s_1 = e_0 + |u_0| \\ \vdots & \vdots \\ B_m \to^* a_{s_m+1} \dots a_{e_m} & s_m = e_{m-1} + |u_{m-1}|, \quad e_m = j - |u_m| \end{array}$$

  ne possède que m degrés de liberté sur un intervalle de longueur $j - i - (|u_{-1}| + \cdots + |u_m|)$.
- une décomposition est valide lorsque :

  $$\forall k, 1 \leqslant k \leqslant m, (B_k, s_k) \in \mathfrak{T}_{s_k, e_k}.$$

  Cette vérification s'effectue en m étapes.

La recherche d'une décomposition valide de A peut donc se faire en temps $O\big((j - i)^{r-1}\big)$. Une fois la décomposition fixée, il suffit d'appeler $\mathfrak{R}$ pour les non-terminaux B_k de la décomposition. □

Remarque 1.5 La complexité donnée dans le théorème précédent est une borne de complexité au pire qui, en pratique, est rarement atteinte. La plupart des dérivations contenant plus d'un non-terminal tendent en effet à décomposer la chaîne en portions de tailles semblables, ce qui fait diminuer la complexité. Par exemple, la complexité moyenne de reconstruction d'un arbre pour la grammaire $S \to (S)S \mid ()$ est $O(n^{3/2})$ (longueur de cheminement moyenne dans les arbres binaires en statistique de type Catalan) pour une complexité au pire de $O(n^2)$.

Remarque 1.6 La complexité de construction d'un arbre de dérivation peut fortement varier d'une grammaire à l'autre, même pour des arbres de dérivation quasi-identiques. Ainsi, la complexité de reconstruction d'un arbre avec la grammaire $S \to Sa \mid a$ n'est que de $O(n)$, alors qu'elle est de $O(n^2)$ avec la grammaire équivalente $S \to SB \mid a, B \to a$.

Remarque 1.7 En pratique, compte tenu des constantes cachées, le temps de construction d'un arbre de dérivation est négligeable devant le temps d'exécution de l'algorithme GCP tenant compte des attributs (cf. théorème 1.6 p. 47).

1.3.6 Complexités de l'algorithme GCP

Les résultats donnés ci-dessous montrent que les complexités en temps et en espace de l'algorithme GCP sont les mêmes que celles de l'algorithme de Earley. En particulier, l'algorithme GCP, comme l'algorithme de Earley, est asymptotiquement plus rapide et moins gourmand en mémoire lorsque les grammaires sur lesquelles il est appliqué sont non-ambiguës ou $LR(k)$ non-récursives à droite.

La diminution des complexités pour des grammaires non-ambiguës peut sembler sans intérêt dans le contexte d'une recherche de structures secondaires d'ARN, puisque nous avons déjà montré que toutes les grammaires décrivant des structures secondaires d'ARN sont ambiguës. Il faut cependant souligner qu'une grammaire ambiguë peut présenter des non-ambiguïtés locales (i.e. certains non-terminaux peuvent engendrer des dérivations non-ambiguës) exploitables par l'algorithme GCP. D'autre part, nous verrons lors de la généralisation de l'algorithme GCP a plusieurs bandes que la diminution des complexités en cas de grammaire non-ambiguë est d'une grande importance pratique (cf. théorème 2.2 p. 83 et chapitre 3).

Théorème 1.2
Soit $|V_N|$ le nombre de non-terminaux d'une grammaire $G = (V_T, V_N, P, S)$ propre, et $|P|$ la somme des tailles des parties droites des productions de P. Soit $a_1 \dots a_n$ une chaîne analysée par l'algorithme GCP. Dans le cas général, la complexité de l'algorithme GCP est $O(|V_N||P|n^3)$ en temps et $O(|P|n^2)$ en espace.

Démonstration : Par définition de Q, on vérifie immédiatement que, à j fixé, chaque appel à la fonction `Extract` dans l'algorithme GCP retourne une paire (X, i) de priorité strictement inférieure à celle de la paire précédente. La file de priorité pouvant contenir au plus $O\big(|V_N|j\big)$ éléments différents, les étapes 3 à 6 sont donc exécutées $O\big(|V_N|j\big)$ fois, et l'étape 2 d'extraction des éléments de la file de priorité peut être exécutée en temps total $O\big(|V_N|j\big)$ avec une implémentation triviale des files de priorité (table indexée par j et X). L'étape 3 ne créant qu'un seul état d'analyse, elle ne demande que $O(1)$ opérations. La durée d'exécution de l'étape 4 dépend du nombre d'états d'analyse sur le point d'être ajoutés à T_j (certains de ces états, déjà présents dans T_j, ne seront finalement pas ajoutés). Or il y a $O\big(|P|i\big)$ états d'analyse dans T_i susceptibles d'être prolongés en $[\Delta \rightarrowtail \alpha X, h]$. L'étape 4 peut donc être exécutée en temps $O\big(|P|i\big)$. Remarquons au passage que s'il y a $O\big(|P|i\big)$ états dans T_i, il y a $O\big(|P|n^2\big)$ états dans l'ensemble des listes d'analyse. Le temps d'exécution de l'étape 5 peut être séparé en deux composantes. La première composante provient des $O\big(|P|i\big)$ états à réduire à chaque passage dans l'étape 5, soit $O(|V_N||P|j^2)$ réductions faites au cours du calcul de T_j. La deuxième composante provient des $O\big(|V_N|j\big)$ paires différentes issues des $O(|V_N||P|j^2)$ réductions et devant être insérées dans Q. Le coût total d'insertion est de $O\big(|V_N|j\big)$ en utilisant l'implémentation triviale de Q déjà citée (table indexée par j et X). Enfin, Δ_j peut être calculé en temps $O\big(|P|j\big)$. L'algorithme GCP s'exécute donc en temps $O(|V_N||P|n^3)$. □

Remarque 1.8 La dépendance en $|P|$ dans les complexités du théorème est une borne pessimiste. En effet, cette borne est essentiellement fonction du nombre d'états d'analyse de la forme $[\Delta \rightarrowtail \alpha]$. En pratique, le nombre d'états d'analyse est plus proche du nombre de productions (et même parfois inférieur à ce nombre) que de la somme des tailles des productions.

Remarque 1.9 La grammaire étant une donnée fixe du processus d'analyse, les dépendances en $|V_N|$ et $|P|$ seront désormais intégrées à la notation $O()$.

Remarque 1.10 La démonstration précédente utilise une implémentation triviale de la file de priorité Q. En pratique, des implémentations plus sophistiquées ayant théoriquement des complexités au pire plus importantes sont considérablement plus efficaces (cf. section 4.4 p. 193). Cela signifie que la complexité au pire de la gestion de Q n'est pas une bonne mesure dans le cadre de l'algorithme GCP. Une étude théorique de la distribution de la profondeur de Q en cours d'analyse syntaxique en fonction de l'implémentation de Q pourrait se révéler intéressante. Dans tous les cas, la complexité $O(n^3)$ en temps de l'algorithme GCP est une borne supérieure non améliorable dans le cas général.

Nous aurons besoin du lemme suivant dans la démonstration des prochains théorèmes.

Lemme 1.1
Si la grammaire G est non-ambiguë alors l'algorithme GCP ne tente d'ajouter un état d'analyse $[\Delta \rightarrowtail \alpha, i]$ à T_j qu'une seule fois pour $a_1 \ldots a_n \in V_T^*$ donné.

Démonstration : Un état d'analyse de la forme $[\Delta \rightarrowtail \alpha a, i]$ n'est effectivement ajouté qu'une seule fois puisqu'un terminal n'apparaît dans la paire retournée par $\mathsf{Extract}(Q)$ que lors de la première extraction.

Supposons maintenant qu'un état d'analyse de la forme $[\Delta \rightarrowtail \alpha A, i]$ apparaisse deux fois lors de la construction de T_j. Cela signifie que le non-terminal A a été extrait deux fois de Q, sous la forme (A, k) une première fois, puis sous la forme (A, l) une deuxième fois. Donc il existe $[\Delta' \rightarrowtail \alpha', k] \in T_j$ tel que $A \in \mathsf{Reduce}\big([\Delta' \rightarrowtail \alpha']\big)$ et il existe $[\Delta'' \rightarrowtail \alpha'', l] \in T_j$ tel que $A \in \mathsf{Reduce}\big([\Delta'' \rightarrowtail \alpha'']\big)$. De plus, nous savons depuis le théorème 1.1 que la présence de $[\Delta \rightarrowtail \alpha A, i]$ dans T_j signifie qu'il existe $B \in \Delta$ et

γ tels que $B \to \alpha A\gamma \to^* a_{i+1} \dots a_j\gamma$. Donc :

$$B \to^* \alpha\alpha'\gamma \to^* a_{i+1} \dots a_j\gamma$$
$$B \to^* \alpha\alpha''\gamma \to^* a_{i+1} \dots a_j\gamma$$

Mais nous savons aussi que $\alpha' \to^* a_{k+1} \dots a_j$ et $\alpha'' \to^* a_{l+1} \dots a_j$. Si $k \neq l$, αA possède alors deux dérivations distinctes vers la même chaîne terminale, ce qui est bien sûr impossible par hypothèse. Si $k = l$, alors $\alpha' \neq \alpha''$ et nous avons deux dérivations de A vers la même chaîne terminale, ce qui est une fois de plus impossible. □

Remarque 1.11 Nous ne pouvons, par définition de Q, extraire deux fois la même paire. Le cas $k = l$ de la démonstration du lemme précédent semble donc impossible. Mais il est impossible uniquement parce que toutes les apparitions potentielles de (A, k) ont été rassemblées en une seule lors de la construction de Q. La démonstration de l'impossibilité du cas $k = l$ lorsque la grammaire est non-ambiguë montre en fait qu'on n'essaiera jamais d'ajouter deux fois la même paire à Q.

Théorème 1.3
Si la grammaire G est non-ambiguë, alors l'algorithme GCP a des complexités en temps et en espace identiques. Sa complexité en temps est donc $O(n^2)$ pour analyser une chaîne $a_1 \dots a_n \in V_T^*$ donnée.

Démonstration : Nous avons vu dans la démonstration du théorème 1.2 que, pour un j donné, les étapes 2 à 6 sont exécutées en un nombre d'étapes proportionnel au nombre de tentatives d'ajout d'états, plus un terme $O\big(|T_j|\big)$ pour la gestion de la file de priorité et un terme $O\big(|T_j|\big)$ pour le calcul de Δ_j. Or, d'après le théorème précédent, on ne tente d'ajouter à T_j un état $[\Delta \rightarrowtail \alpha, i]$ qu'une seule fois lorsque G est non-ambiguë. Le nombre total de tentatives d'ajout d'états est donc égal au nombre d'états réellement ajoutés. La complexité en temps est donc proportionnelle à la complexité en espace. Comme la complexité en espace est $O(n^2)$ au pire, la complexité en temps est $O(n^2)$ au pire dans le cas d'une grammaire non-ambiguë. □

Théorème 1.4
Quelle que soit la grammaire G, l'algorithme GCP a les mêmes ordres de grandeur de complexités en temps et en espace que l'algorithme de Earley.

Démonstration : Tout état $\big[\{A_1, \dots A_k\} \rightarrowtail \alpha, i\big]$ ajouté à T_j par l'algorithme GCP correspond en fait à l'ajout d'au moins k états $\big[A_m \to \alpha \centerdot \beta_m, i\big]$ à T_j par l'algorithme de Earley. Donc les complexités en temps et en espace de l'algorithme GCP sont dominées par celles de l'algorithme de Earley.

Tous les états $\big[A_p \to \alpha \centerdot \beta_p, i\big]$ ajoutés à T_j par l'algorithme de Earley correspondent à l'ajout d'un unique état $[\Delta \rightarrowtail \alpha, i]$ à T_j par l'algorithme GCP. Donc T_j, dans l'algorithme GCP, contient au moins le nombre d'états de l'algorithme de Earley divisé par le nombre de productions de la grammaire. Donc les ordres de grandeur des complexités en temps et en espace de l'algorithme GCP dominent ceux de l'algorithme de Earley. □

Type de grammaire	**Complexité en temps**	**Complexité en espace**
non-contextuelle	$O(n^3)$	$O(n^2)$
non-ambiguë	$O(n^2)$	$O(n^2)$
ambiguïté bornée	$O(n^2)$	$O(n^2)$
métalinéaire	$O(n^2)$	$O(n^2)$
$LR(k)$ non-récursive à droite	$O(n)$	$O(n)$
listes d'analyse bornées	$O(n)$	$O(n)$

Table 1.1: Quelques complexités de l'algorithme GCP.

Théorème 1.5
Si la grammaire G est à ambiguïté bornée, ou linéaire, ou métalinéaire[12]*, alors l'algorithme GCP a des complexités en temps et en espace de $O(n^2)$.*

Si la grammaire G est $LR(k)$ et non-récursive à droite, alors l'algorithme GCP a des complexités en temps et en espace égales à $O(n)$ pour analyser une chaîne $a_1 \dots a_n$ de V_T^* donnée.

Démonstration : c'est une conséquence immédiate du théorème précédent et de (Earley, 1970). □

Notons que dans le cas où G est $LR(k)$ et non-récursive à droite, alors les listes d'analyse T_j contiennent un nombre borné d'états d'analyse. Les grammaires vérifiant cette propriété (qui assure des complexités linéaires en temps et en espace de l'algorithme GCP) sont appelées grammaires à listes d'analyse bornées (d'après *bounded state grammars* dans (Earley, 1970)). Les grammaires $LR(k)$ non-récursives à droites ne sont pas les seules grammaires à listes d'analyse bornées. La table 1.1 récapitule les résultats obtenus.

L'intérêt principal de l'algorithme GCP, comparé à l'algorithme de Earley, provient de la diminution des constantes masquées dans la notation $O(...)$. La factorisation d'une multitude d'états $[A_i \to \alpha \centerdot \beta_i]$ sous la forme d'un unique état $[\Delta \rightarrowtail \alpha]$ diminue, parfois dans des proportions importantes, le nombre d'états à ajouter aux listes d'analyses. Les gains en temps de calcul des listes d'analyse et en place occupée en mémoire par ces listes sont évidents. Ces gains peuvent être importants pour certaines grammaires (cf. section 1.7.3 p. 64). Le remplacement des états d'analyse n'appartenant pas au noyau par un simple ensemble de non-terminaux attendus présente les mêmes avantages que la factorisation des états d'analyse. Il faut garder à l'esprit que l'analyse d'une chaîne d'ARN pour une grammaire ambiguë donnée peut entraîner l'utilisation de la totalité des ressources d'une machine. Une simple division par deux de la place occupée en mémoire peut faire la différence entre une chaîne d'ARN impossible à analyser en temps raisonnable (i.e. la machine utilise trop le "swap") et une chaîne analysable.

[12] Une grammaire G est métalinéaire lorsque S possède une dérivation de la forme $S \to A_1 \dots A_n$ et lorsque les dérivations des A_i sont linéaires.

1.4 Grammaires S-attribuées

Nous avons montré qu'une grammaire non-contextuelle permet de décrire un ensemble de structures secondaires d'un ARN par l'intermédiaire d'un ensemble d'arbres de dérivation de cet ARN. Mais, du point de vue du biologiste, nous ne semblons pas beaucoup plus avancés qu'auparavant. Rappelons en effet que notre but n'est pas simplement de construire des structures secondaires, mais aussi de trouver une structure secondaire acceptable selon certains critères objectifs (énergie minimale pour le modèle thermodynamique, probabilité maximale pour le modèle stochastique, etc.).

Nous devons donc trouver un moyen d'ajouter des critères de sélection d'arbres de dérivation à l'algorithme GCP si nous voulons continuer à utiliser le formalisme grammatical pour replier les ARN. Il est bien sûr hors de question d'examiner un par un chaque arbre de dérivation pour savoir s'il satisfait le critère de sélection choisi, sous peine d'y passer un temps croissant exponentiellement avec la longueur des ARN.

Fort heureusement, la totalité des méthodes de calcul de structure secondaire couramment utilisées reposent sur l'optimisation (minimisation ou maximisation) récursive d'une valeur. Ce schéma récursif de calcul donne en général lieu à des implémentations explicitement référencées comme étant de type “programmation dynamique”, que ce soit pour le modèle thermodynamique (Zuker, 1989b) ou pour le modèle stochastique (Sakakibara *et al.*, 1993). Or, nous allons voir que ce schéma récursif de calcul se prête bien à l'optimisation récursive des attributs synthétisés des arbres de dérivation pour une grammaire S-attribuée appropriée. Les grammaires S-attribuées Lewis *et al.* (1974) sont un sous-ensemble des grammaires attribuées introduites par Knuth (Knuth, 1968). Contrairement aux grammaires attribuées, les grammaires S-attribuées (ou SAG) n'utilisent pas à la fois des attributs hérités et des attributs synthétisés, mais uniquement des attributs synthétisés (d'où le S de S-attribué).

Avant d'illustrer la notion de grammaire S-attribuée par un exemple, nous en présentons une définition adaptée à notre contexte d'utilisation.

Définition 1.13 (Grammaire S-attribuée)
Une grammaire S-attribuée notée $G = (V_T, V_N, P, S, \mathcal{A}, S_{\mathcal{A}}, F_P)$ est une extension de la grammaire non-contextuelle $G = (V_T, V_N, P, S)$, où l'on adjoint à chaque $X \in V$ un attribut x appartenant à un ensemble d'attributs $\mathcal{A}$, et à chaque élément $\alpha \in V^*$ une chaîne d'attributs λ de $\mathcal{A}^*$. $S_{\mathcal{A}}$ est une fonction de V_T dans $\mathcal{A}$ affectant un attribut à chaque terminal. F_P est un ensemble de fonctions de $\mathcal{A}^*$ dans $\mathcal{A}$, notées $f_{A\to\alpha}$ dont le rôle est de calculer l'attribut de chaque non-terminal présent dans une dérivation. Une fonction $f_{A\to\alpha}$ appartient à F_P si et seulement si $A \to \alpha$ appartient à P.

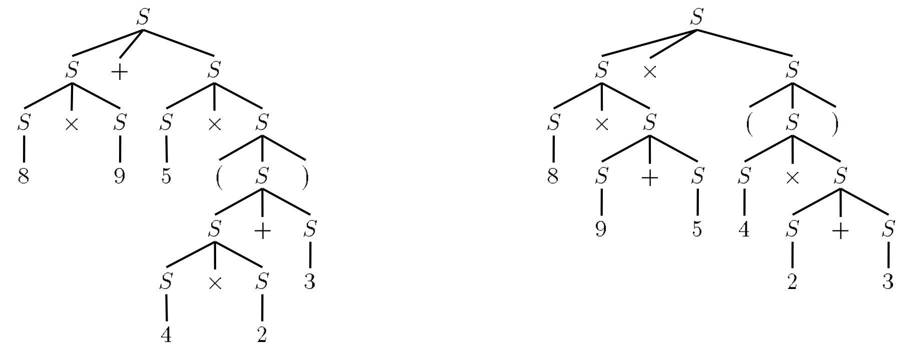
La chaîne d'attributs λ d'une chaîne α est la concaténation formelle des attributs des symboles de α. Lorsqu'une fonction $f_{A\to\alpha}$ est appliquée à une chaîne d'attributs λ d'une chaîne α dérivée de A, elle retourne l'attribut x de A. Par conséquent, les fonctions de F_P sont responsables du calcul des attributs synthétisés des non-terminaux. L'attribut du non-terminal S pour une dérivation $S \to \alpha_1 \to \ldots \to \alpha_k \to u$ est calculé à partir des attributs des terminaux de u, calculés grâce à $S_{\mathcal{A}}$, en remontant peu à peu la chaîne des dérivations (c'est un calcul dit bottom-up). Les attributs synthétisés sont particulièrement simples à calculer par un algorithme d'analyse syntaxique de type bottom-up et ils ne souffrent pas des problèmes de circularité rencontrés dans les grammaires attribuées. Dans la suite de cette thèse, nous utiliserons les conventions suivantes pour éviter de multipler inutilement les qualificatifs d'appartenance :

- x, y, z pour décrire des éléments de $\mathcal{A}$;
- λ, μ, ν pour décrire des éléments de $\mathcal{A}^*$;

Comme promis, nous allons illustrer la définition des grammaires S-attribuées par un exemple simple et classique. La grammaire suivante est une grammaire non-contextuelle ambiguë décrivant des expressions arithmétiques.

$$\begin{aligned}
&S \rightarrow S + S\\
&S \rightarrow S \times S\\
&S \rightarrow (S)\\
&S \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

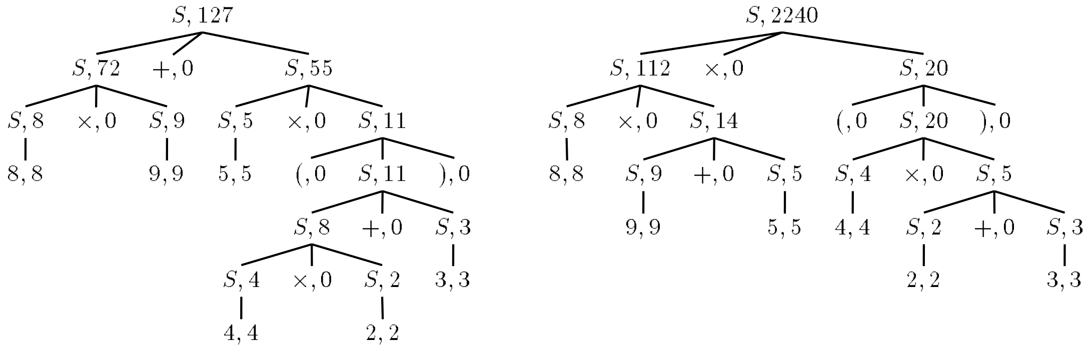
Pour cette grammaire, l'expression $8 \times 9 + 5 \times (4 \times 2 + 3)$ est dérivable de plusieurs manières différentes. Deux des arbres de dérivation possibles sont :



L'arbre de gauche correspond à un opérateur $\times$ prioritaire sur $+$ (priorité usuelle). L'arbre de droite correspond à des priorités inversées. Nous pouvons transformer la grammaire précédente en une grammaire S-attribuée ayant pour but d'affecter à chaque non-terminal d'une dérivation un attribut entier égal au résultat du calcul représenté par cet arbre. Les éléments à ajouter sont les suivants :

$$\mathcal{A} = \mathbb{N} \qquad \begin{array}{c} S_{\mathcal{A}}(0) = 0 \\ \vdots \\ S_{\mathcal{A}}(9) = 9 \\ S_{\mathcal{A}}(+, \times, (,)) = 0 \end{array} \qquad F_P = \left\{ \begin{array}{cl} f_{S \rightarrow S+S}(xyz) & = x + z \\ f_{S \rightarrow S \times S}(xyz) & = x \times z \\ f_{S \rightarrow (S)}(xyz) & = y \\ f_{S \rightarrow a \in V_T}(x) & = x \end{array} \right\}$$

Grâce à la grammaire S-attribuée obtenue, les sommets des deux arbres précédents peuvent être décorés d'attributs.



Observons que l'attribut de chaque sommet des deux arbres est calculable uniquement à partir d'informations présentes aux sommets des sous-arbres. Dans cet exemple, nous avons choisi à dessein une grammaire ambiguë conduisant à des valeurs multiples de l'expression $8 \times 9 + 5 \times (4 \times 2 + 3)$, ce qui n'est pas souhaitable pour une expression arithmétique. Pour obtenir une valeur unique, il faut utiliser une grammaire non-ambiguë respectant la priorité des opérations $+$ et $\times$. La grammaire non moins classique suivante pourra convenir :

$$
\begin{aligned}
&S \to S + T \mid T \\
&T \to T \times E \mid E \\
&E \to (S) \\
&E \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

À la fin de ce chapitre, nous verrons deux exemples de grammaires S-attribuées appliquées, l'une au modèle thermodynamique, l'autre au modèle stochastique.

1.5 Maîtrise de l'explosion combinatoire

Soit G une grammaire S-attribuée dont l'ensemble d'attributs $\mathcal{A}$ est égal au codage sous forme de chaîne de caractères de l'ensemble des séquences de dérivations possibles dans la grammaire. Le code utilisé peut être un code sous forme d'expression préfixe parenthésée avec lequel, par exemple, la séquence $S \to AB \to ACD \to aCD \to acD \to acd$ est représentée par la chaîne $(S(A(a))(B(C(c))(D(d))))$. Pour associer un attribut de $\mathcal{A}$ à chaque symbole de la dérivation, il suffit de définir $S_{\mathcal{A}}(a) = (a)$ et $f_{A \to \alpha}(\lambda) = (A\lambda)$.

Si la grammaire G ainsi définie est à ambiguïté exponentielle, alors le non-terminal de départ S possède un nombre d'attributs différents augmentant exponentiellement avec la taille d'une chaîne analysée, puisque ce nombre d'attributs est précisément égal au nombre de dérivations possibles de cette chaîne. Pour cette grammaire, tout algorithme effectuant le calcul des attributs en parallèle avec l'analyse syntaxique verra ses complexités d'analyse devenir exponentielles. Pour éviter un tel phénomène d'explosion combinatoire, nous devons contraindre l'algorithme à n'utiliser qu'un unique attribut[13] pour toute dérivation $A \rightarrow^* a_{i+1} \ldots a_j$, et donc un unique arbre si le codage précédent est utilisé. Cette stratégie d'élagage est tout à fait réaliste dans le cadre des applications envisagées (minimisation d'énergie, maximisation de probabilité, etc.). Nous introduisons donc des contraintes extrinséques de G, dites *contraintes d'optimisation*, dont le rôle est précisément de choisir une dérivation parmi un grand nombre de dérivations. Les contraintes d'optimisation sont introduites sous la forme d'une fonction $\mathcal{C}_A$ associée à chaque non-terminal A.

Définition 1.14 (Contrainte d'optimisation)
Soient G une grammaire S-attribuée et $a_1 \ldots a_n$ une chaîne analysée. Soit x l'attribut d'une dérivation $A \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow a_{i+1} \ldots a_j$ et soit x' l'attribut d'une autre dérivation $A \rightarrow \alpha'_1 \rightarrow \cdots \rightarrow \alpha'_{k'} \rightarrow a_{i+1} \ldots a_j$.

La contrainte d'optimisation $\mathcal{C}_A$ associée au non-terminal A prend en paramètres i, j, x et x' et retourne un nouvel attribut x'' déduit de ces quatre paramètres. On écrira $x'' = \mathcal{C}_A(i, j, x, x')$.

Les quatre paramètres d'une contrainte d'optimisation suffisent à caractériser la sous-dérivation considérée. Un sous-arbre de sommet A est optimal lorsque son attribut est optimal au sens de la contrainte $\mathcal{C}_A$.

Ainsi, si les attributs sont des réels représentant des énergies et si un arbre d'énergie minimale est recherché, la contrainte de chaque non-terminal doit retourner la plus petite des deux valeurs x et x'. À l'inverse, si les attributs sont des réels représentant des probabilités et si un arbre de probabilité maximale est recherché, la contrainte de chaque non-terminal doit retourner la plus grande des deux valeurs x et x'.

Remarque 1.12 Pour que les contraintes d'optimisation soient cohérentes avec le but à atteindre il faut que les sous-arbres de tout arbre de dérivation optimal soient eux-mêmes optimaux. La contrainte souhaitée doit donc pouvoir être récursivement évaluée. Il existe des contraintes ne se prêtant pas à une évaluation récursive. Pourtant, la plupart des contraintes "naturelles" sont naturellement récursives.

[13]Ou un nombre borné d'attributs. Mais, modulo un codage approprié, un nombre borné d'attributs est équivalent à un unique attribut

1.6 Analyse syntaxique des grammaires S-attribuées

1.6.1 Algorithme GCP attribué avec contraintes d'optimisation

Dans la section 1.3, nous avons décrit un algorithme d'analyse syntaxique pour les grammaires non-contextuelles. Dans la section 1.4, les grammaires non-contextuelles ont été transformées en grammaires S-attribuées grâce à l'addition d'un ensemble d'attributs $\mathcal{A}$, d'une fonction $S_{\mathcal{A}}$ retournant les attributs des terminaux, et d'un ensemble F_P de fonctions $f_{A\to\alpha}$ d'évaluation des attributs des non-terminaux. Enfin, nous avons montré dans la section 1.5 qu'une stratégie d'élagage est nécessaire à l'évaluation d'un attribut d'un arbre de dérivation et que les contraintes d'optimisation de la définition 1.14 sont suffisantes à cet effet. Tous ces éléments peuvent aisément être pris en compte dans une version modifiée de l'algorithme GCP grâce aux extensions suivantes.

Définition 1.15 (État attribué)
Soit $[\Delta \rightarrowtail \alpha, i]$ un état d'analyse. Soit λ une chaîne d'attributs de α. L'association dans une même structure de l'état d'analyse $[\Delta \rightarrowtail \alpha, i]$ et de λ est appelée état attribué. Cette association est notée $[\Delta \rightarrowtail \alpha, i, \lambda]$.

Les états attribués seront également appelés états lorsqu'aucune confusion ne sera possible.

La réduction des états d'analyse dans l'algorithme GCP donne lieu à la création de paires (A, i) ajoutées à une file de priorité Q. Cette fois, les fonctions d'évaluation des attributs des non-terminaux $f_{A\to\alpha}$ doivent être utilisées au moment de la réduction des états attribués $[\Delta \rightarrowtail \alpha, i, \lambda]$. Les paires (A, i) sont donc remplacées par des triplets (A, i, x), où x est l'attribut associé à A renvoyé par $f_{A\to\alpha}(\lambda)$.

Les contraintes d'optimisation permettent de ne sélectionner qu'un seul[14] sous-arbre de dérivation pour toute dérivation $A \to^* a_{i+1} \dots a_j$. Elles doivent donc être utilisées juste après la réduction des états attribués, au moment de l'ajout des triplets (A, i, x) à la file de priorité, pour empêcher la file de priorité de contenir deux triplets (A, i, x) et (A, i, x'). Notons que si pour tout A et pour tout i la file de priorité contient au plus un triplet (A, i, x), l'ordre sur les triplets (A, i, x) dans la file de priorité est induit par l'ordre sur les paires (A, i).

Définition 1.16 (File de priorité sur les triplets (X, i, x))
Soit Q un ensemble de triplets $(X, i, x) \in V \times \{1, \dots, n\} \times \mathcal{A}$. Soient (X, i, x) et (X', i', x') deux éléments de Q. Nous dirons que $(X', i', x') < (X, i, x)$, ou que l'extraction de (X, i, x) est prioritaire sur celle de (X', i', x'), si $i' < i$ ou si $i' = i$ et $X' \to^+ X$. Nous noterons $\mathsf{Extract}$ la fonction utilisée pour simultanément retourner et supprimer de Q le triplet maximum (X, i, x) de Q. Nous noterons cette extraction $(X, i, x) := \mathsf{Extract}(Q)$.

La version modifiée de l'algorithme GCP tenant compte des attributs et des contraintes d'optimisation est décrite ci-dessous.

[14] Ou, comme nous l'avons établi précédemment, un petit nombre fini modulo un codage approprié de l'espace $\mathcal{A}$ et une modification de l'algorithme de reconstruction d'arbres.

Algorithme 1.4
Soit Q la file de triplets (X, i, x) qui vient d'être définie avec sa fonction $\mathsf{Extract}$. *Au départ, les listes d'analyse T_j sont vides. Soit $\Delta_0 = \{A \in V_N \mid \exists \beta, S \rightarrow^* A\beta\}$. Pour j allant de 1 à n, exécuter les étapes suivantes :*

1. *Initialiser la file Q:* $\quad Q := \Big\{\big(a_j, j-1, S_{\mathcal{A}}(a_j)\big)\Big\}$
2. *Extraction du triplet à traiter en priorité :*

   $(X, i, x) := \mathsf{Extract}(Q)$
3. *Création des états attribués commençant par X :*

   $T_j := T_j \cup \Big\{[\Delta \rightarrowtail X, i, x] \Bigm| [\Delta \rightarrowtail X] = \mathsf{Initial}(\Delta_i, X)\Big\}$
4. *Prolongement d'états attribués par X :*

   $$T_j := T_j \cup \Big\{[\Delta \rightarrowtail \alpha X, h, \lambda x] \Bigm| \exists [\Delta' \rightarrowtail \alpha, h, \lambda] \in T_i,\ [\Delta \rightarrowtail \alpha X] = \mathsf{Goto}\big([\Delta' \rightarrowtail \alpha], X\big)\Big\}$$
5. *Pour chaque état attribué $[\Delta \rightarrowtail \alpha, i, \lambda]$ ajouté à T_j par une des étapes 3 et 4 ci-dessus, et pour chaque $A \in \mathsf{Reduce}\big([\Delta \rightarrowtail \alpha]\big)$, calculer le triplet $(A, i, x) := \big(A, i, f_{A\rightarrow\alpha}(\lambda)\big)$.*

   S'il existe déjà un triplet (A, i, x') appartenant à Q, remplacer l'attribut x' de ce triplet par $\mathcal{C}_A(i, j, x, x')$.

   Si aucun triplet de ce type n'existe dans Q, ajouter (A, i, x) à Q.
6. *Répéter les étapes 2 à 6 tant que Q n'est pas vide;*
7. *Calculer $\Delta_j := \mathsf{Expect}(T_j)$.*

La chaîne $a_1 \ldots a_n$ appartient à $L(G)$ si et seulement si le triplet $(S, 0, x)$ a été extrait de Q pendant le calcul de T_n. On dit alors que la chaîne a été reconnue par l'algorithme et qu'elle a un attribut optimal x.

L'algorithme 1.4 reconnaît les chaînes de $L(G)$ car l'ajout des attributs et des contraintes ne modifie pas le principe de création des états défini dans l'algorithme GCP.

Pour montrer que l'algorithme 1.4 est correct, il suffit donc de montrer qu'il calcule effectivement un attribut optimal d'une chaîne de $L(G)$. Or ce résultat est une conséquence directe de la relation d'ordre sur les triplets utilisée dans la file de priorité, dont l'usage se trouve donc justifié. En effet, pour que l'attribut optimal d'une chaîne soit retourné, il suffit que l'attribut de toute sous-chaîne de la chaîne de départ calculé par l'algorithme soit optimal. Donc il suffit que tout triplet (A, i, x) extrait de la file de priorité lors du calcul de T_j ait un attribut x optimal pour les sous-dérivations $A \rightarrow^* a_{i+1} \ldots a_j$. L'attribut x est optimal si la contrainte $\mathcal{C}_A$ a pu être appliquée sur toutes les sous-dérivations de A avant l'extraction de (A, i, x), c'est-à-dire si aucune réduction d'un triplet (A, i, x') n'est effectuée après l'extraction de (A, i, x). Or, par définition de l'ordre sur les éléments de Q, une telle réduction ultérieure est impossible.

1.6.2 Contraintes de sélection

Dans certains cas, il est utile d'éliminer très tôt certaines sous-dérivations de la chaîne analysée si l'attribut de ces dérivations ne satisfait pas un critère donné. Ainsi, pour une grammaire cherchant à minimiser l'énergie totale d'un arbre afin de trouver une structure secondaire plausible (cf. p. 49 pour une introduction au modèle thermodynamique), il est intéressant de supprimer d'office les sous-dérivations ayant une énergie trop élevée et donc n'ayant aucune chance de participer à la structure finale. Grâce à cette suppression anticipée, l'analyseur syntaxique peut dans une certaine mesure éviter d'effectuer des calculs et des allocations mémoire inutiles. La proportion de temps et d'espace ainsi gagnés dépendent bien sûr du critère d'élagage des sous-arbres dans la forêt de sous-arbres de dérivation.

L'étape 5 de l'algorithme GCP est le meilleur endroit pour supprimer les sous-dérivations : juste après la réduction d'une règle, l'attribut x d'un non-terminal A est disponible et il est possible à ce niveau de décider si le triplet (A, i, x) doit être sélectionné pour un ajout à Q ou une comparaison avec un autre triplet (A, i, x') de Q.

Définition 1.17 (Contrainte de sélection)
Soit $(A, i, x) := \big(A, i, f_{A\to\alpha(\lambda)}\big)$ un triplet obtenu pendant le calcul de T_j par réduction d'une règle $A \to \alpha$ dans l'étape 5 de l'algorithme GCP.

La contrainte de sélection $\mathcal{C}_A^{\mathcal{S}}$ associée au non-terminal A est un prédicat prenant en paramètre i, j et x et retournant vrai *si le triplet (A, i, x) doit être sélectionné, et* faux *sinon.*

L'absence de contrainte de sélection est équivalente à $\mathcal{C}_A^{\mathcal{S}}(i, j, x)$ vrai quels que soient A, i, j, x. Si le triplet (A, i, x) est sélectionné, alors la suite de l'étape 5 peut avoir lieu, sinon l'algorithme passe directement à l'étape 6. Grâce aux contraintes de sélection, l'étape 5 de l'algorithme est réécrite de la manière suivante :

5. *Pour chaque état attribué $[\Delta \rightarrowtail \alpha, i, \lambda]$ ajouté à T_j par une des étapes 3 et 4 ci-dessus, et pour chaque $A \in \mathsf{Reduce}\big([\Delta \rightarrowtail \alpha]\big)$, calculer le triplet $(A, i, x) := \big(A, i, f_{A\to\alpha}(\lambda)\big)$.*

   Si $\mathcal{C}_A^{\mathcal{S}}(i, j, x)$ est faux alors passer directement à l'étape 6.

   S'il existe déjà un triplet (A, i, x') appartenant à Q, remplacer l'attribut x' de ce triplet par $\mathcal{C}_A(i, j, x, x')$.

   Si aucun triplet de ce type n'existe dans Q, ajouter (A, i, x) à Q.

Remarque 1.13 S'il existe A, i, j tels qu'un triplet (A, i, x) soit réduit à plusieurs reprises au cours du calcul de T_j, mais tels qu'aucun triplet (A, i, x) ne soit ajouté à Q, alors la contrainte de sélection $\mathcal{C}_A^{\mathcal{S}}$ a pour effet de modifier la structure de la forêt d'arbres de dérivation de $a_1 \dots a_n$. En effet, la dérivation $A \to a_{i+1} \dots a_j$ est une sous-dérivation valide puisque le triplet (A, i, x) a été calculé. Si (A, i, x) n'est pas ajouté à Q à cause de la contrainte de sélection alors la sous-dérivation ne sera incluse dans aucun arbre de dérivation plus important, comme si elle était en fait invalide.

Le langage généré par G peut même être appauvri par les contraintes de sélection si $A \to^* a_{i+1} \ldots a_j$ est la seule manière de dériver $a_{i+1} \ldots a_j$.

Le comportement de la contrainte de sélection $\mathcal{C}_A^{\mathcal{S}}$ est à distinguer de celui de la contrainte d'optimisation $\mathcal{C}_A$, qui sélectionne les triplets une fois qu'un triplet a déjà été enregistré dans la file de priorité Q, et qui ne peut donc pas changer le langage généré par la grammaire.

Plusieurs exemples de contraintes de sélection sont donnés au chapitre 3, en particulier dans la section 3.2. La syntaxe des contraintes de sélection et d'optimisation est donnée dans l'annexe A p. 245.

1.6.3 Reconstruction de l'arbre d'analyse optimal

Pour tout non-terminal A et pour tout index i, l'algorithme GCP attribué extrait de Q au plus un triplet (A, i, x) correspondant à une sous-dérivation optimale. Donc le principe de reconstruction des arbres de dérivation déjà donné reste utilisable (cf. p. 33). Nous en reproduisons ci-dessous les grandes lignes, modifiées comme il convient pour tenir compte des attributs.

Lorsqu'un triplet (X, i, x) est extrait de Q pendant le calcul de T_j, son apparition est mémorisée dans une table particulière $(\mathfrak{T}_{i,j})_{1 \leqslant i < j \leqslant n}$ initialement vide :

$$\mathfrak{T}_{i,j} := \mathfrak{T}_{i,j} \cup \big\{(X, x)\big\}.$$

Le coût de stockage de cette table et de ses éléments est de $O(n^2)$ dans le cas général, mais il est possible d'implémenter cette table de façon à ramener le coût à $O(n)$ seulement lorsque $O(n)$ états d'analyse ont été créés.

Notons $\mathfrak{R}$ la fonction qui à un quadruplet (A, i, j, x) associe un arbre de dérivation de $A \to^* a_{i+1} \ldots a_j$ possédant un attribut x. Cette fonction se calcule comme suit :

1. Recherche, parmi l'ensemble des dérivations valides de A, c'est-à-dire des dérivations telles que $A \to u_{-1} B_0 u_0 \ldots B_m u_m \to^* a_{i+1} \ldots a_j$, $B_k \to^* a_{s_k+1} \ldots a_{e_k}$ a un attribut optimal x_k, et λ_k est la chaîne d'attributs associée à u_k par application de $S_{\mathcal{A}}$ à ses éléments individuels, de la première dérivation valide[15] ayant un attribut $f_{A \to u_{-1} B_0 u_0 \ldots B_m u_m}(\lambda_{-1} x_0 \lambda_0 \ldots x_m \lambda_m)$ égal à x. Cette dérivation détermine la racine de l'arbre et ses fils.

2. Si $m \neq -1$ (i.e. si $u_{-1} \neq a_{i+1} \ldots a_j$), appeler récursivement $\mathfrak{R}(B_k, s_k, e_k, x_k)$ $(0 \leqslant k \leqslant m)$ pour connaître le reste de l'arbre.

Proposition 1.3 (Complexité de reconstruction de l'arbre optimal)
Soit r le nombre maximum de non-terminaux apparaissant dans les parties droites α des règles $A \to \alpha$ de G. Soit x l'attribut optimal d'une dérivation $S \to^* a_1 \ldots a_n$. Alors, dans l'hypothèse où les fonction $f_{A \to \alpha}$ de F_P s'exécutent en $O(1)$, $\mathfrak{R}(S, 0, n, x)$ retourne l'arbre de dérivation associé à l'attribut x en temps $O(n^r)$.

[15] Grâce au langage C, il est possible de définir des contraintes d'optimisation ayant des effets de bords importants. La recherche d'une dérivation peut alors s'avérer infructueuse. Dans un tel cas, il est préférable de mémoriser l'arbre optimal en même temps qu'il est calculé (cf. section A.5 p. 262).

Démonstration: identique à celle de la proposition 1.2. □

Les remarques faites à la suite de la proposition 1.2 à propos de la complexité de reconstruction d'un arbre d'analyse restent valides lorsqu'il s'agit de reconstruire un arbre optimal.

1.6.4 Complexités de l'algorithme GCP attribué

Bien que très semblable à l'algorithme GCP, l'algorithme GCP attribué a des complexités différentes dans le cas général, du fait de l'ajout de chaînes d'attributs aux états d'analyse.

Théorème 1.6
Soit r le nombre maximum de non-terminaux apparaissant dans les parties droites α des règles $A \to \alpha$ de la grammaire G. Dans le cas général, la complexité de l'algorithme GCP attribué est $O(n^{r+1})$ en temps et $O(n^r)$ en espace.

Démonstration : pour l'essentiel, la structure de la démonstration est identique à celle du théorème 1.2.

La file de priorité contenant toujours au plus $O(j)$ éléments différents, les étapes 2 à 6 sont exécutées $O(j)$ fois. Les étapes 2 et 3 sont toujours exécutées en temps $O(j)$ au total, et la complexité d'exécution de l'étape 4 dépend encore du nombre d'états sur le point d'être ajoutés à T_i. Elle est donc bornée par le nombre d'états appartenant à T_i.

Mais, alors que dans l'algorithme GCP il y avait au plus $O(i)$ états $[\Delta' \rightarrowtail \alpha, h]$ dans T_i susceptibles d'être prolongés en $[\Delta \rightarrowtail \alpha X]$, il n'en va pas de même avec l'algorithme GCP attribué. En effet, les états attribués de l'algorithme GCP attribué étant de la forme $[\Delta' \rightarrowtail \alpha, h, \lambda]$, il faut tenir compte de la variabilité de λ. Par exemple, si $\alpha = u_1 B_0 u_0 \ldots B_m u_m$, alors on montre comme dans la proposition 1.2 p. 34 que λ peut prendre $O\big((i-h)^m\big)$ valeurs. Comme $m \leqslant r-1$, il y a en théorie au plus $O(i^r)$ états dans T_i. Mais, en pratique, il n'est jamais nécessaire de stocker les états au delà de $m \leqslant r-2$ (cf. section 4.5.1 p. 201). Donc il y a au plus $O(i^{r-1})$ états dans T_i (soit $O(n^r)$ états au total dans l'ensemble des listes d'analyse). L'étape 5 a donc une complexité totale de $O(j^r)$ en ce qui concerne la réduction des états et de $O(j)$ au total en ce qui concerne l'ajout de triplets à la file de priorité. L'algorithme GCP attribué s'exécute donc en temps $O(n^{r+1})$.

Ces complexités restent valables lorsque $r = 1$, puisque la grammaire devient linéaire. □

Remarque 1.14 En pratique, les grammaires S-attribuées que nous avons utilisées ont été écrites de façon à conserver $r \leqslant 2$. Nous avons donc retrouvé les complexités habituelles d'analyse syntaxique sans attribut : $O(n^3)$ en temps et $O(n^2)$ en espace.

Remarque 1.15 Une grammaire non-contextuelle peut toujours se mettre sous forme normale de Chomsky. Donc, dans le cas des grammaires non-contextuelles, il est toujours possible de se ramener au cas où $r = 2$. Mais il est faux d'en déduire que les complexités d'analyse pour une grammaire S-attribuée avec contraintes peuvent toujours

Type de grammaire	Complexité en temps	Complexité en espace
non-contextuelle	$O(n^{r+1})$	$O(n^r)$
non-ambiguë	$O(n^2)$	$O(n^2)$
ambiguïté bornée	$O(n^2)$	$O(n^2)$
métalinéaire	$O(n^2)$	$O(n^2)$
$LR(k)$ non-récursive à droite	$O(n)$	$O(n)$
listes d'analyse bornées	$O(n)$	$O(n)$

Table 1.2: Quelques complexités de l'algorithme GCP attribué.

être ramenées à $O(n^3)$ en temps et $O(n^2)$ en espace. En effet, le processus de normalisation d'une grammaire impose un découpage des productions et donc l'introduction de nouveaux non-terminaux sur lesquels il n'est pas forcément possible d'appliquer une contrainte. L'ambiguïté sur les chaînes d'attributs λ des états attribués, responsable des complexités $O(n^{r+1})$ en temps et $O(n^r)$ en espace dans l'algorithme GCP attribué, est alors déplacée vers la file de priorité Q pour les non-terminaux ne possédant pas de contrainte, et les complexités données dans le théorème 1.6 sont retrouvées.

Théorème 1.7
Si la grammaire G est non-ambiguë, alors l'algorithme GCP attribué a des complexités en temps et en espace identiques. Sa complexité en temps est donc $O(n^2)$ pour analyser une chaîne $a_1 \ldots a_n \in V_T^*$ donnée.

Démonstration : comme on ne tente d'ajouter un état attribué $[\Delta \rightarrowtail \alpha, i, \lambda]$ à T_j qu'au plus une fois, chaque état d'analyse $[\Delta \rightarrowtail \alpha, i]$ ne possède qu'un seul état attribué équivalent (i.e. comme il n'y a qu'un seul arbre de dérivation, il est forcément optimal, et tous les attributs calculés sont uniques). Donc la démonstration du théorème 1.3 est immédiatement transposable.

Théorème 1.8
Si la grammaire G est $LR(k)$ et non-récursive à droite, alors l'algorithme GCP attribué a des complexités en temps et en espace égales à $O(n)$ pour analyser une chaîne $a_1 \ldots a_n \in V_T^*$ donnée.

Démonstration : c'est une conséquence immédiate de la démonstration précédente et du théorème 1.5. □

La table 1.2 récapitule les complexités obtenues.

1.7 Applications des grammaires S-attribuées

1.7.1 Comparaison entre YACC et l'algorithme GCP

Afin de pouvoir aisément mettre en pratique l'algorithme GCP sur de nombreuses grammaires différentes sans avoir à calculer manuellement les tables d'états d'analyse et les tables de non-terminaux attendus d'une grammaire S-attribuée (SAG), nous avons créé un outil appelé MTSAG2C générant automatiquement un analyseur syntaxique à partir de la donnée d'une SAG. Pour pouvoir être lues par MTSAG2C, les grammaires doivent être écrites selon une syntaxe proche de celle de YACC : le programme YACC est un outil standard sous UNIX qui, étant donnée une description d'une grammaire $LALR(1)$, produit un analyseur syntaxique pour cette grammaire.

Les programmes MTSAG2C et YACC ont donc des fonctions semblables[16] pour des classes différentes de grammaires. Si l'on restreint l'application de MTSAG2C aux grammaires $LALR(1)$, il devient possible, et même naturel, de comparer les performances des analyseurs syntaxiques générés par MTSAG2C et YACC. Pour effectuer les comparaisons, nous avons en fait utilisé BISON (Donnelly & Stallman, 1990), un proche parent de YACC.

La grammaire $LALR(1)$ utilisée dans les tests suivants est celle du langage C (Lee, 1985). Les résultats obtenus avec cette grammaire $LALR(1)$ sur un Pentium 120 en fonction de la taille de la chaîne analysée sont présentés en figure 1.8. L'algorithme GCP ayant été introduit pour gérer les grammaires ambiguës utilisées avec les structures secondaires d'ARN, les analyseurs générés par MTSAG2C sont optimisés pour ces grammaires. Donc l'algorithme GCP est beaucoup plus lent (environ 5 fois) avec les grammaires $LALR(1)$ qu'un analyseur BISON optimisé pour ces grammaires.

Les autres graphes montrent que l'algorithme GCP a bien une complexité linéaire en temps et en espace en fonction de la taille des chaînes analysées. Bien qu'étant linéaire, la complexité en espace de l'analyse est importante : près de 40 Moctets pour un fichier source C de 50 000 terminaux (représentant 200 Koctets de code C), contre 200 Koctets pour l'analyseur généré par BISON. Cette complexité importante est due aux nombreuses structures introduites pour gérer les ambiguïtés, structures finalement inutiles avec une grammaire $LALR(1)$.

Remarquons que l'analyseur MTSAG2C ne construit pas beaucoup plus d'états d'analyse que l'analyseur BISON. Le comportement de l'algorithme GCP est donc bon pour cette grammaire $LALR(1)$.

1.7.2 Application des SAG au modèle thermodynamique

Le modèle thermodynamique des structures secondaires d'ARN s'appuie sur une forme additive de l'énergie libre des structures, dépendant uniquement des interactions locales entre bases. Plus précisément, toute structure secondaire sans pseudo-nœud est

[16] MTSAG2C doit cependant gérer des contraintes de sélection et des contraintes d'optimisation, inutiles lorsque toute chaîne ne possède qu'un seul arbre de dérivation, comme c'est le cas avec les grammaires $LALR(1)$. MTSAG2C possède également plusieurs mécanismes syntaxiques supplémentaires facilitant l'écriture de grammaires adaptées à l'analyse de chaînes biologiques.

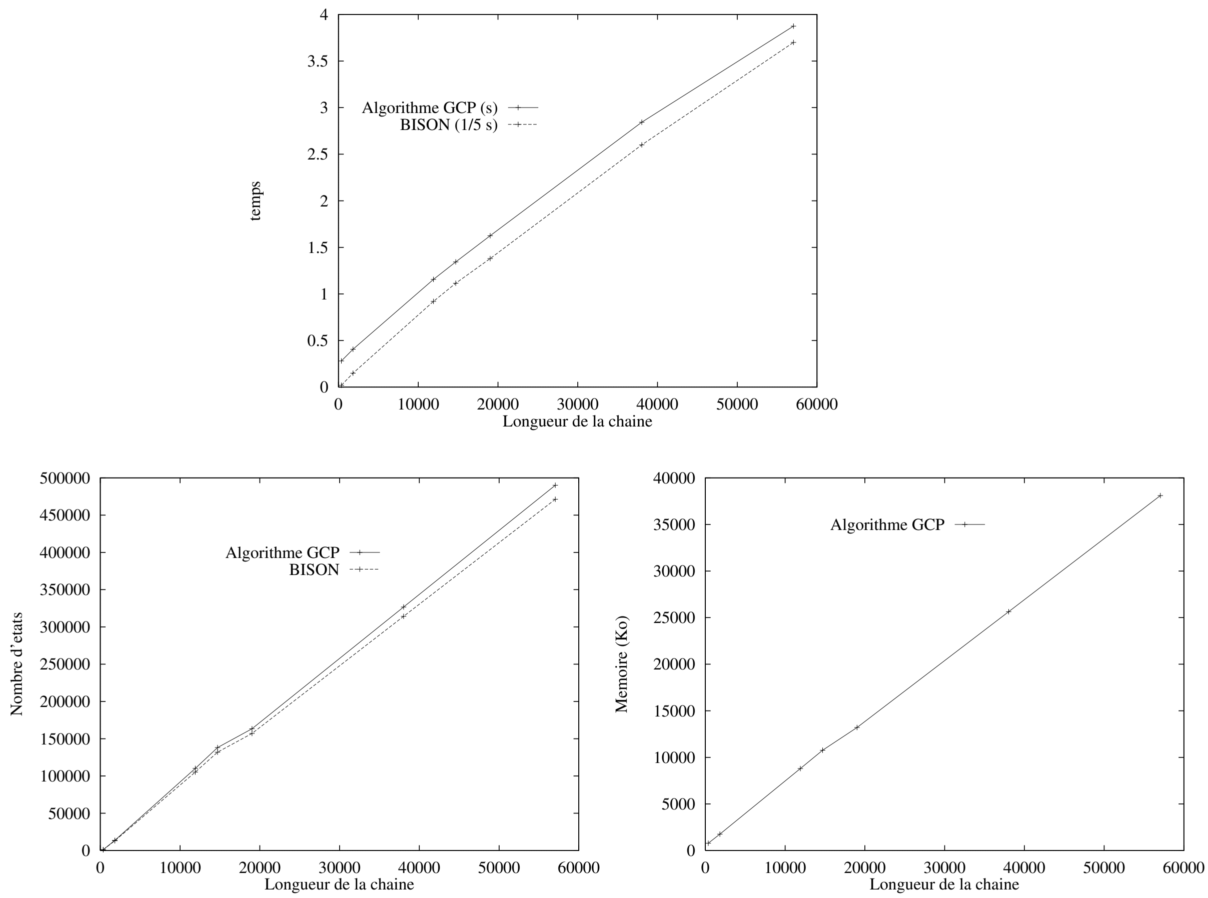


Figure 1.8: Le graphe du haut compare les temps d'exécution de l'algorithme GCP et de l'analyseur généré par BISON en fonction de la longueur de la chaîne analysée. Les deux graphes du bas donnent respectivement le nombre d'états utilisés par les deux algorithmes et la mémoire allouée par l'algorithme GCP, toujours en fonction de la longueur de la chaîne analysée. La mémoire allouée par l'analyseur de BISON est restée inférieure à 200 Koctets pour tous nos tests car nous n'avons pas demandé à l'analyseur de construire l'arbre de dérivation trouvé alors que l'algorithme GCP le fait implicitement. Mais même en demandant à l'analyseur de construire l'arbre de dérivation, la quantité de mémoire allouée, bien que proportionnelle à la longueur de la chaîne analysée, serait restée nettement inférieure à celle de l'algorithme GCP.

décomposable en sous-structures particulières appelées *boucles*. La somme des énergies de chacune de ces sous-structures est censée fournir l'énergie de la structure entière (Zuker, 1989b).

Nous donnerons d'abord la définition formelle des boucles d'une structure secondaire, en suivant pour l'essentiel les conventions adoptées par Zuker dans (Zuker, 1989b), puis nous donnerons un exemple de calcul d'énergie de ces boucles. La méthode de programmation dynamique classique sera ensuite rappelée, puis une comparaison entre cette méthode et l'analyse syntaxique sera effectuée.

1.7.2.1 Présentation du modèle thermodynamique

Définition 1.18 (Structure primaire)
La structure primaire d'une molécule d'ARN est un mot $a_1a_2\dots a_n$ de longueur n, dont les lettres a_i appartiennent à un alphabet contenant au moins quatre lettres A, C, G, U, représentant les bases habituelles.

Définition 1.19 (Structure secondaire)
Une **structure secondaire** *sans pseudo-nœud S d'un ARN $a_1\dots a_n$ est un ensemble de paires d'entiers $(i\cdot j)$ vérifiant :*

1. $i<j$;
2. $(i\cdot j)\in S \iff a_i$ *et* a_j *sont appariés;*
3. *si* $(i_1\cdot j_1)\in S$ *et* $(i_2\cdot j_2)\in S$ *alors* $i_1=i_2 \iff j_1=j_2$;
4. *quels que soient* $(i_1\cdot j_1)\in S$ *et* $(i_2\cdot j_2)\in S$, *on n'a jamais* $i_1<i_2<j_1<j_2$.

$S_{i,j}$ *désigne l'ensemble des paires d'entiers $(i'\cdot j')$ de S telles que $i\leqslant i'<j'\leqslant j$.*

La condition 3 exprime le fait qu'un nucléotide ne peut s'apparier qu'avec un seul autre nucléotide. La condition 4 exclut les pseudo-nœuds des structures secondaires.

Définition 1.20 (Boucle, paire fermante, index et taille d'une boucle)
Soient $(i\cdot j)\in S$ et h un entier tel que $i\leqslant h\leqslant j$. On dit que h est accessible de $(i\cdot j)$ s'il n'existe pas de paire $(k\cdot l)$ telle que $i<k<h<l<j$.

L'ensemble des entiers accessibles de $(i\cdot j)$ forme une **boucle** *notée $L_{i,j}$. La paire $(i\cdot j)$ est appelée* **paire fermante** *de la boucle. Les paires $(k\cdot l)\neq(i\cdot j)$ telles que $k\in L_{i,j}$ (donc $l\in L_{i,j}$ puisque les pseudo-nœuds sont interdits) sont appelées* **paires intérieures**. *Il est impossible de parler de boucle $L_{i,j}$ si $(i\cdot j)\notin S$. L'ensemble des paires intérieures et fermante d'une boucle sont appelées* **paires accessibles**. *L'ensemble des entiers qui ne sont accessibles d'aucune paire est appelé* **boucle extérieure** *et est noté L_{ext}. Les bases d'une boucle extérieure sont appelées les bases extérieures.*

*L'***index** *d'une boucle $L_{i,j}$ est le cardinal de l'ensemble des paires accessibles de la boucle.*

La **taille** *d'une boucle $L_{i,j}$ est égale au cardinal de $L_{i,j}$ diminué du double de l'index. C'est aussi le nombre de bases non-appariées (simple brin) de la boucle.*

La figure 1.9(a) illustre cette définition.

Définition 1.21 (Classification des boucles)
Les boucles $L_{i,j}$ peuvent être classifiées comme suit :

- *une boucle d'index 1 est appelée une* **épingle à cheveux**,
- *une boucle d'index 2 définit deux paires notées $(i \cdot j)$ pour la paire fermante et $(k \cdot l)$ pour la paire intérieure.*
  - *Si $k = i+1$ et $l = j-1$ (taille nulle) alors $(i \cdot j)$ et $(k \cdot l)$ forment un* **empilement**. *Par extension, on appelle empilement une suite de boucles de la forme $L_{i,j}, L_{i+1,j-1}, \ldots, L_{i+k,j-k}$ avec $k \geqslant 1$;*
  - *Si $k > i+1$* **et** *$l < j-1$, la boucle est une* **boucle intérieure**;
  - *Si $k > i+1$* **ou** *$l < j-1$, la boucle est une* **boucle proéminente**;
- *une boucle d'index $\geqslant 3$ est une* **boucle multiple**.

La figure 1.9(b) illustre cette définition.

Définition 1.22 (Énergie d'une structure)
L'ensemble des boucles d'une structure secondaire réalise une décomposition de cette structure. L'énergie $e(S)$ est définie comme somme des énergies de chaque boucle :

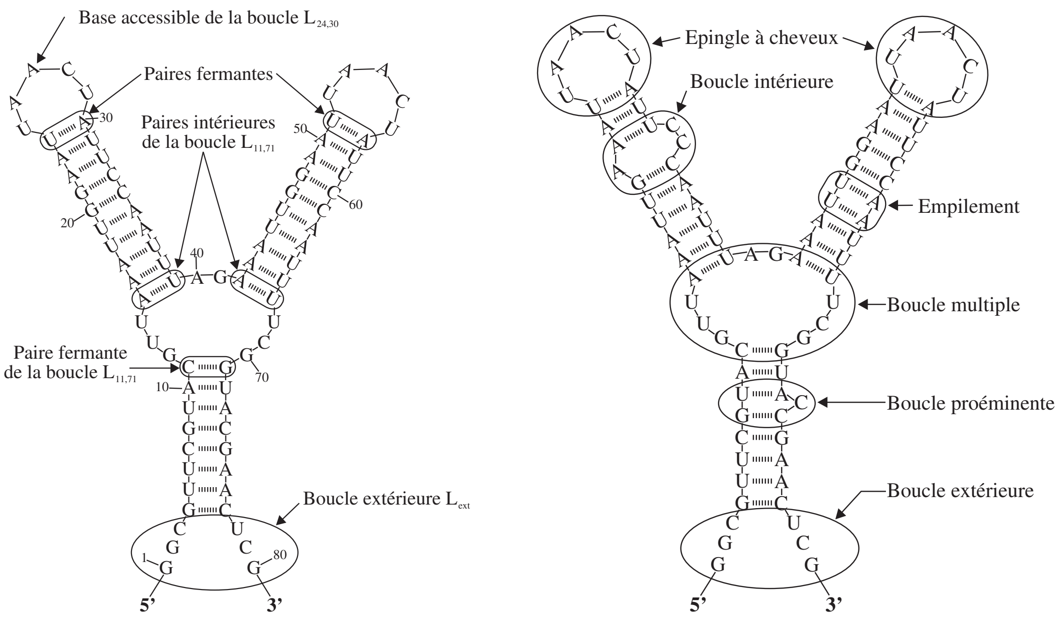
$$e(S) = e\big(L_{\text{ext}}\big) + \sum_{(i \cdot j) \in S} e\big(L_{i,j}\big).$$

Nous ferons l'hypothèse que les énergies des boucles dépendent uniquement de la taille des boucles, des paires délimitant ces boucles et, éventuellement, d'un nombre fini de nucléotides dans ces boucles. Autrement dit, le calcul de $e\big(L_{i,j}\big)$ ou $e\big(L_{\text{ext}}\big)$ est toujours réalisable en $O(1)$. Sans cette hypothèse, il est impossible de calculer une structure secondaire en temps $O(n^3)$, que ce soit avec l'algorithme de programmation dynamique ou avec nos grammaires S-attribuées. Nous reproduisons dans la définition 1.23 ci-dessous quelques valeurs d'énergies de boucles couramment utilisées et issues de (Papanicolaou *et al.*, 1984; Turner *et al.*, 1987; Freier *et al.*, 1986; Jaeger *et al.*, 1989).

Définition 1.23 (Énergie des différents types de boucles)

- **l'énergie d'un empilement** *$L_{i,j}$ est attribuée en fonction des paires $(i \cdot j)$ et $(k \cdot l) = (i+1 \cdot j-1)$ (i.e. $k = i+1$, $l = j-1$) de l'empilement :*

$(i \cdot j)$ \ $(k \cdot l)$	AU	CG	GC	GU	UA	UG
AU	-0,9	-2,1	-1,6	-0,6	-0,9	-0,7
CG	-1,9	-3,0	-2,0	-1,5	-1,6	-1,5
GC	-2,1	-3,4	-3,0	-1,5	-2,1	-1,9
GU	-0,5	-1,9	-1,5	-0,5	-0,7	-0,5
UA	-1,1	-2,3	-1,9	-0,7	-0,9	-0,5
UG	-1,1	-1,3	-1,5	-0,6	-0,5	-0,5



(a) Illustration de la définition des boucles et de leurs éléments associés. La boucle $L_{24,30}$ est d'index 1 et de taille 5. La boucle $L_{11,71}$ est d'index 3 et de taille 8.

(b) Représentation des différents types de boucles.

Figure 1.9: Les structures représentées dans ces deux figures ont la forme d'un `Y`, la séparation en deux branches s'effectuant au niveau d'une boucle multiple. Plus généralement, tous les points de branchement dans une structure secondaire se situent au niveau des boucles multiples.

- **l'énergie d'une boucle proéminente** *est égale à la somme de :*
  - *l'énergie d'empilement des deux paires délimitant la boucle (table précédente) si la boucle est de taille 1, ou 0 sinon ;*
  - *l'énergie de déstabilisation due à la boucle et dépendant de la taille de cette boucle :*

taille de la boucle	0	1	2	3	4	5	6	7	8	9	10
$\Delta G(kcal/mole)$	$+\infty$	3,9	3,1	3,5	4,2	4,8	5,0	5,1	5,3	5,4	5,5

Dans cette table, comme dans les tables suivantes, la valeur $+\infty$ donnée à l'énergie d'une boucle de taille donnée signifie que cette boucle est interdite dans les structures secondaires.

- **l'énergie de mauvais appariement terminal** *d'une paire* $(i \cdot j)$ *ou* $(j \cdot i)$ *d'une boucle dépend de la paire et des deux bases adjacentes accessibles dans la boucle, c'est-à-dire des bases* $k = i+1$ *et* $l = j-1$. *Cette énergie est donnée par la table suivante :*

k,l / i,j	*AA*	*AC*	*AG*	*AU*	*CA*	*CC*	*CG*	*CU*	*GA*	*GC*	*GG*	*GU*	*UA*	*UC*	*UG*	*UU*
AU	-0,8	-1,0	-1,0	-1,0	-0,7	-0,7	-0,7	-0,7	-0,8	-1,0	-1,0	-1,0	-0,8	-0,8	-0,8	-0,8
CG	-1,9	-2,0	-1,9	-1,9	-1,0	-1,1	-1,0	-0,8	-1,9	-1,9	-1,9	-1,9	-1,4	-1,5	-1,4	-1,2
GC	-1,1	-1,3	-1,3	-1,3	-1,1	-0,6	-0,6	-0,5	-1,6	-1,5	-1,4	-1,5	-0,8	-0,8	-0,8	-0,7
GU	-0,8	-1,0	-1,0	-1,0	-0,7	-0,7	-0,7	-0,7	-0,8	-1,0	-1,0	-1,0	-0,8	-0,8	-0,8	-0,8
UA	-1,0	-0,8	-1,1	-0,9	-0,7	-0,6	-0,3	-0,5	-1,1	-0,9	-1,2	-0,9	-0,3	-0,6	-0,3	-0,5
UG	-1,5	-1,4	-1,5	-1,4	-0,9	-0,9	-0,7	-0,7	-1,5	-1,4	-1,6	-1,4	-0,9	-1,1	-0,9	-0,9

- **l'énergie d'une épingle à cheveux** *est égale à la somme de :*
  - *l'énergie de déstabilisation due à la boucle et dépendant de la taille de cette boucle :*

taille de la boucle	0	1	2	3	4	5	6	7	8	9	10
$\Delta G(kcal/mole)$	$+\infty$	$+\infty$	$+\infty$	4,5	5,5	4,9	5,1	5,2	5,5	5,8	5,9

  - *l'énergie de mauvais appariement terminal de la paire fermante* $(i \cdot j)$*;*
  - *un bonus de -2,0 kcal/mole si la chaîne simple brin de la boucle est égale à l'une des chaînes suivantes :* `GAGA, GUGA, GCGA, GAAA, GGAA, GCAA, AUCG, AACG.`

- **l'énergie d'une boucle interne** *est égale à la somme de*
  - *l'énergie de déstabilisation due à la boucle et dépendant de la taille de cette boucle :*

taille de la boucle	0	1	2	3	4	5	6	7	8	9	10
$\Delta G(kcal/mole)$	$+\infty$	$+\infty$	4,1	4,5	4,9	5,3	5,7	5,8	6,0	6,1	6,2

  - *les énergies de mauvais appariement terminal de la paire fermante* $(i \cdot j)$ *et de la paire intérieure* $(i' \cdot j')$ *;*
  - *une correction dépendant de l'assymétrie de la boucle :*

    $$\min\Big(3.0, |i' - i + j' - j| \times f\big(\min(4, i' - i - 1, j - j' - 1)\big)\Big)$$

    avec $f(0) = 0$*;* $f(1) =$*0,4;* $f(2) =$*0,3;* $f(3) =$*0,2;* $f(4) =$*0,1.*

- **l'énergie d'une boucle multiple** *est égale à*

  $$4,5 + 0{,}4 \times (\textit{taille de la boucle}) + 0{,}1 \times (\textit{index de la boucle})$$

- *finalement,* **l'énergie d'une boucle extérieure** *est choisie égale à zéro.*

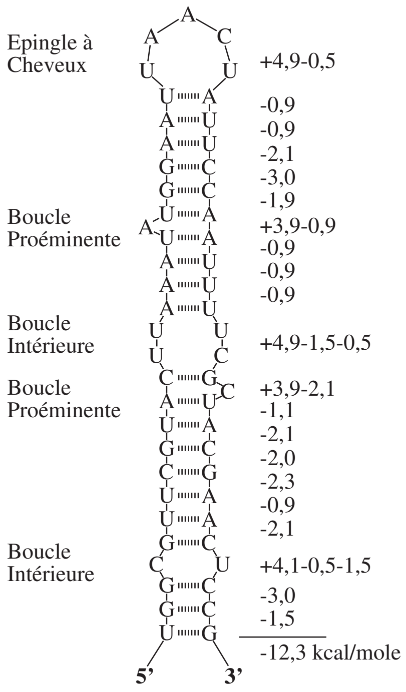


Figure 1.10: Calcul de l'énergie libre d'une structure secondaire d'un fragment d'ARN du virus R17.

Lorsque les tailles des boucles dépassent la taille des tables (10 ici), l'énergie est interpolée en utilisant la relation

$$E_{interpolée} = E_{boucle\ taille\ 10} + 1,07856 \log\left(\frac{taille\ boucle}{10}\right).$$

La figure 1.10 donne un exemple de calcul d'énergie d'une structure secondaire.

1.7.2.2 Programmation dynamique et modèle thermodynamique

Le but du modèle thermodynamique est de permettre le calcul d'une structure S telle que $e(S)$ soit minimale. L'usage de la programmation dynamique pour le calcul d'une structure secondaire d'énergie minimale est rendu possible grâce à la décomposition de l'énergie d'une structure en la somme des énergies de ses boucles (définition 1.22) :

$$e(S) = e\big(L_{\text{ext}}\big) + \sum_{(i \cdot j) \in S} e\big(L_{i,j}\big).$$

En effet, la programmation dynamique est une stratégie utilisée pour résoudre un problème de taille n lorsque sa solution est facilement déductible d'une décomposition de ce problème en sous-problèmes de tailles inférieures auxquels le même principe de résolution est applicable. Le calcul de $\min e(S)$ va donc être transformé en une minimisation récursive d'énergies calculées sur des sous-ensembles $S_{i,j}$ de S. Pour exposer la méthode utilisée, nous utiliserons une fois de plus les notations de Zuker (Zuker, 1989b).

Définition 1.24 (Identification de $W_{i,j}$ et $V_{i,j}$)
On note $W_{i,j} = \min e(S_{i,j})$ l'énergie minimale d'une structure secondaire non vide $S_{i,j}$ sur $a_i \dots a_j$, contenant éventuellement *la paire de nucléotides $(i \cdot j)$. Si $S_{i,j} = \emptyset$, la convention $W_{i,j} = 0$ est adoptée.*

On note $V_{i,j}$ l'énergie minimale possible pour une structure secondaire $S_{i,j}$ contenant obligatoirement *la paire de nucléotides $(i \cdot j)$. Lorsque les nucléotides i et j ne peuvent pas s'apparier, on fixe $V_{i,j} = +\infty$.*

On a toujours $V_{i,j} \geqslant W_{i,j}$ puisque la définition de $V_{i,j}$ est celle de $W_{i,j}$ avec la containte supplémentaire d'un appariement $(i \cdot j)$ obligatoire. L'énergie cherchée, $e(S)$, est égale à $W_{1,n}$.

Proposition 1.4 (Décomposition du calcul de $W_{i,j}$ et $V_{i,j}$)
$W_{i,j}$ et $V_{i,j}$ vérifient :

$$W_{i,j} = \min_{\substack{i \leqslant i_1 < j_1 < \cdots < i_k < j_k \leqslant j \\ k > 0}} \Big(e\big(L_{\text{ext}}\big) + \sum_{l=1}^{k} V_{i_l, j_l} \Big) \tag{1.1}$$

$$V_{i,j} = \min_{\substack{i < i_1 < j_1 < \cdots < i_k < j_k < j \\ k \geqslant 0}} \Big(e\big(L_{i,j}\big) + \sum_{l=1}^{k} V_{i_l, j_l} \Big) \tag{1.2}$$

La boucle L_{ext} apparaissant dans (1.1) est formée des bases $i \dots (i_1 - 1)$ et $(j_k + 1) \dots j$.

Les équations (1.1) et (1.2) sont obtenues en remarquant que les paires fermantes et intérieures d'une boucle réalisent une partition de la structure secondaire d'un ARN comprise entre les bases i et j. $W_{i,j}$ n'intervient pas dans le calcul de $V_{i,j}$. $W_{1,n}$ peut donc théoriquement être calculé au dernier moment.

Une application directe des équations (1.1) et (1.2) nécessite la génération de toutes les paires $(i_1 \cdot j_1), \dots, (i_k \cdot j_k)$ $(k \geqslant 0)$ possibles, soit $O(2^{j-i})$ paires pour i et j fixés, ou $O(2^n)$ paires au total. Cela rend ces équations de programmation dynamique inutilisables en l'état.

Proposition 1.5 (Autre calcul de $W_{i,j}$)
Si nous supposons tout d'abord que $e\big(L_{\text{ext}}\big) = 0$, $W_{i,j}$ peut s'écrire :

$$W_{i,j} = \min\Big(W_{i+1,j}, W_{i,j-1}, V_{i,j}, \min_{i \leqslant k < j} \big(W_{i,k} + W_{k+1,j} \big) \Big). \tag{1.3}$$

ou encore, puisque $W_{i,i} = 0$:

$$W_{i,j} = \min\Big(V_{i,j}, \min_{i \leqslant k < j} \big(W_{i,k} + W_{k+1,j} \big) \Big). \tag{1.4}$$

En effet, les bases i et j peuvent :

- être appariées sous la forme $(i \cdot j)$ (terme $V_{i,j}$);
- être appariées sous la forme $(i \cdot k)$ ou $(k \cdot j)$ (terme $W_{i,k} + W_{k+1,j}$);
- ne pas être appariées (terme $W_{i,k} + W_{k+1,j}$ également).

Avec l'équation (1.4), le temps de calcul de $W_{i,j}$ est ramené à $O(j-i)$ (à cause de la minimisation sur l'indice k), et celui de $W_{1,n}$ à $O(n^3)$. Mais cette fois, il n'est pas possible de calculer $W_{1,n}$ au dernier moment et tous les $W_{i,j}$ doivent être calculés et mémorisés, soit un coût en mémoire de $O(n^2)$.

Il nous reste maintenant à réécrire l'équation (1.2) pour ramener la complexité du calcul de $V_{i,j}$ à un niveau raisonnable. Nous allons pour cela décomposer cette équation en fonction de l'index de la boucle $L_{i,j}$.

Proposition 1.6 (Décomposition du calcul de $V_{i,j}$)
Soit $V_{i,j}^{k}$ l'énergie minimale des structures secondaires $S_{i,j}$ dont la boucle $L_{i,j}$ est d'index k, et $V_{i,j}^{\geqslant k}$ l'énergie minimale des structures dont la boucle $L_{i,j}$ est d'index au moins k. $V_{i,j}$ vérifie l'équation suivante :

$$V_{i,j} = \min\left(V_{i,j}^{1}, V_{i,j}^{2}, V_{i,j}^{\geqslant 3}\right).$$

Il reste maintenant à calculer les énergies $V_{i,j}^{1}$, $V_{i,j}^{2}$, $V_{i,j}^{\geqslant 3}$.

- *$V_{i,j}^{1} = e(L_{i,j})$. Cette énergie est donc calculée en temps $O(1)$ pour chaque paire $(i \cdot j)$, ou $O(n^2)$ au total.*
- *$V_{i,j}^{2} = \min\limits_{i<i'<j'<j}\left(e\left(L_{i,j,i',j'}\right) + V_{i',j'}\right)$, où $L_{i,j,i',j'}$ représente une boucle d'index 2, dont la paire fermante est $(i \cdot j)$ et la paire intérieure $(i' \cdot j')$. Cette énergie est donc calculée en temps $O\left((j-i)^2\right)$ pour chaque paire $(i \cdot j)$, ou $O(n^4)$ au total.*
- *Pour maintenir la complexité du calcul de $V_{i,j}^{\geqslant 3}$ (c'est-à-dire des structures terminées par une boucle multiple) à un niveau raisonnable, on adopte pour les boucles multiples une forme affine de l'énergie (comme dans la définition 1.23) :*

$$e\left(L_{i,j,i_1,j_1,\ldots,i_k,j_k}\right) = e_{\mathit{fermante}}(i \cdot j) + a \times (\text{taille de la boucle}) + \sum_{l=1}^{k} e_{\mathit{interne}}(i_l \cdot j_l)$$

où $L_{i,j,i_1,j_1,\ldots,i_k,j_k}$ est une extension naturelle de la notation $L_{i,j,i',j'}$ aux boucles multiples. Soit $W'_{i,j}$ l'énergie minimale des structures secondaires sur $a_i \ldots a_j$ dont l'énergie de boucle extérieure est donnée par l'équation affine précédente.

$$V_{i,j}^{\geqslant 3} = e_{\mathit{fermante}}(i \cdot j) + \min_{i+1 \leqslant k < j-1}\left(W'_{i+1,k} + W'_{k+1,j-1}\right)$$

$$W'_{i,j} = \min \begin{cases} V_{i,j} + e_{\mathit{interne}}(i \cdot j) \\ \min_{i \leqslant k < j}\left(W'_{i,k} + (j-k)a, W'_{i,k} + W'_{k+1,j}, (k-i+1)a + W'_{k+1,j}\right) \end{cases}$$

$V_{i,j}^{\geqslant 3}$ est donc calculée en temps $O(j-i)$ pour chaque paire $(i \cdot j)$, ou $O(n^3)$ au total.

Il est donc possible de calculer $V_{1,n}$ et $W_{1,n}$ en temps $O(n^4)$ et en espace $O(n^2)$, tout en gardant une certaine généralité aux énergies des épingles à cheveux, empilements, boucles proéminentes et boucles internes, mais au prix d'une modélisation simpliste de l'énergie des boucles multiples. Pour atteindre une complexité en temps de $O(n^3)$, il suffit de borner la taille des boucles internes. Les variables i' et j' introduites dans le calcul de $V^2_{i,j}$ ne peuvent alors prendre qu'un nombre fini de valeurs, ce qui ramène la complexité du calcul de tous les $V^2_{i,j}$ à $O(n^2)$. Les complexités $O(n^3)$ en temps et $O(n^2)$ en espace (due au stockage des variable $W_{i,j}$, $W'_{i,j}$ et $V_{i,j}$) semblent être les bornes inférieures pour un algorithme raisonnable de calcul d'énergie minimale des repliements d'un ARN.

Remarque 1.16 Jusqu'à présent, nous avons uniquement cherché à calculer l'énergie de la structure secondaire d'un ARN, sans pour autant fournir cette structure secondaire. Pour retrouver la structure secondaire d'un ARN à partir de l'énergie obtenue, il suffit de reprendre les équations de programmation dynamique et de noter récursivement quels termes conduisent au minimum de chaque sous-problème. Cette procédure de construction de la structure secondaire à partir de l'énergie minimale trouvée s'apparente à la procédure donnée p. 46 pour construire un arbre de dérivation optimal à partir de l'attribut optimal d'une chaîne.

1.7.2.3 Grammaires S-attribuées et modèle thermodynamique

Pour obtenir une grammaire S-attribuée du modèle thermodynamique, il faut avant tout construire une grammaire non-contextuelle dont les éléments syntaxiques correspondent aux éléments structuraux du modèle thermodynamique. Le modèle thermodynamique repose sur une décomposition de la structure secondaire en boucles de plusieurs types. Nous utiliserons cette même décomposition en boucles pour créer une première version de grammaire non-contextuelle adaptée au modèle thermodynamique, puis nous optimiserons cette grammaire dans la perspective d'une exécution avec l'algorithme GCP. Il faudra en particulier minimiser le nombre maximum r de non-terminaux apparaissant dans les parties droites des règles.

Tout d'abord, une structure secondaire est composée de deux éléments fondamentaux : les bases non-appariées, que nous notons $\bullet$, et les bases appariées, que nous notons (et). Une portion simple brin, c'est-à-dire contenant uniquement des bases non-appariées, est notée `SingleStrand` et a la syntaxe suivante :

`SingleStrand` $\rightarrow \bullet \mid$ `SingleStrand` $\bullet$

Les bases appariées ferment les boucles (notées `Loop`). Il y a 5 types de boucles :

- les épingles à cheveux, notées `HairpinLoop`,
- les boucles proéminentes, notées `BulgeLoop`,
- les boucles internes, notées `InternalLoop`,
- les boucles multiples, notées `MultipleLoop`,
- les empilements, notés `StackingPair`.

La syntaxe des boucles est donc la suivante :

```
Loop → HairpinLoop | BulgeLoop | InternalLoop | MultipleLoop |
       StackingPair
```

À part pour les boucles multiples, la syntaxe des différentes boucles est simple :

```
HairpinLoop  → ( SingleStrand )
BulgeLoop    → ( SingleStrand Loop ) | ( Loop SingleStrand )
InternalLoop → ( SingleStrand Loop SingleStrand )
StackingPair → ( Loop )
```

Une boucle multiple est théoriquement une suite fermée par une paire de bases d'au moins deux paires intérieures fermant des boucles, ces paires étant éventuellement séparées par des bases. Pour faciliter la description des boucles multiples, nous supposerons que les boucles multiples contiennent un nombre quelconque, éventuellement nul, de paires intérieures[17]. Cette définition est compatible avec la formule affine de calcul d'énergie des boucles multiples, à condition que l'énergie calculée pour une boucle multiple contenant zéro ou une seule paire fermante soit supérieure à celle calculée par la bonne structure (c'est-à-dire `HairpinLoop`, `BulgeLoop`, `InternalLoop` ou `StackingPair`). Les formules de calcul d'énergie données p. 52 vérifient ce critère de compatibilité. Les règles alors obtenues pour les boucles multiples sont les suivantes :

```
LoopSeq      → Loop | • | LoopSeq Loop | LoopSeq •
MultipleLoop → ( LoopSeq )
```

Finalement, une structure secondaire d'ARN, notée `Structure`, est une suite de boucles éventuellement séparées par des bases :

```
Structure → Loop | • | Structure Loop | Structure •
```

Le rassemblement des règles précédentes nous donne la grammaire non-contextuelle suivante.

```
Structure    → Loop | • | Structure Loop | Structure •
Loop         → HairpinLoop | BulgeLoop | InternalLoop |
               MultipleLoop | StackingPair
HairpinLoop  → ( SingleStrand )
BulgeLoop    → ( SingleStrand Loop ) | ( Loop SingleStrand )
InternalLoop → ( SingleStrand Loop SingleStrand )
StackingPair → ( Loop )
LoopSeq      → Loop | • | LoopSeq Loop | LoopSeq •
MultipleLoop → ( LoopSeq )
SingleStrand → • | SingleStrand •
```

[17] Les implémentations des équations des propositions 1.5 et 1.6 (*Vienna package*, cf. section 1.7.2.4) font également cette hypothèse.

Dans l'hypothèse où chaque non-terminal a un attribut de type énergie et une contrainte d'optimisation réalisant en fait une minimisation de l'énergie, la complexité d'analyse avec une grammaire S-attribuée déduite de la grammaire précédente est de $O(n^4)$ en temps et $O(n^3)$ en espace, puisque la règle décrivant la syntaxe des boucles internes contient 3 non-terminaux. Pour obtenir une grammaire contenant au plus $r = 2$ non-terminaux dans la partie droite des règles, nous allons regrouper la définition de la syntaxe de la partie accessible des boucles d'index 2, c'est-à-dire de `BulgeLoop`, `InternalLoop` et `StackingPair`, en un unique non-terminal `Index2`. Ce non-terminal `Index2` doit exprimer le fait que la paire intérieure des boucles d'index 2 est séparée de la paire fermante par zéro ou plusieurs bases non-appariées à gauche et à droite. La grammaire obtenue est alors la suivante :

```
Structure    → Loop | • | Structure Loop | Structure •
Loop         → ( SingleStrand )
Loop         → ( Index2 )
Loop         → ( LoopSeq )
Index2       → Loop | • Index2 | Index2 • | • Index2 •
LoopSeq      → Loop | • | LoopSeq Loop | LoopSeq •
SingleStrand → • | SingleStrand •
```

Dans cette grammaire, nous avons supprimé les définitions de `HairpinLoop` et `MultipleLoop` car ces deux non-terminaux étaient uniquement utilisés pour définir un type particulier de boucle, et il est donc préférable (pour des raisons de performance plutôt que de clarté) de reporter leur syntaxe directement dans le non-terminal `Loop`. La figure 1.11 représente une description de la grammaire S-attribuée déduite de la grammaire non-contextuelle précédente. Comme $r = 2$, les complexités d'analyse pour cette grammaire S-attribuée atteignent l'objectif fixé de $O(n^3)$ en temps et $O(n^2)$ en espace.

La description donnée en figure 1.11 est prévue pour être utilisée par l'outil MT-SAG2C de génération automatique d'analyseurs syntaxiques. La syntaxe particulière de cette description est décrite dans le manuel de référence de MTSAG2C fourni en annexe A. Nous ne détaillerons donc pas cette syntaxe ici. Notons cependant qu'elle est très similaire à celle de YACC, en particulier en ce qui concerne la description des règles et des fonctions d'évaluation d'attributs de la grammaire. Le langage utilisé pour tout ce qui concerne la manipulation des attributs est donc le langage C, ce qui apporte une grande souplesse d'utilisation. Pour le lecteur non-initié, indiquons par exemple que les pseudo-variables \$\$, \$1, \$2, etc. représentent respectivement le non-terminal à gauche d'une règle, le premier symbole de la partie droite d'une règle, le deuxième symbole de la partie droite, etc. La notation \$\$.E représente ainsi la valeur de l'attribut E du non-terminal à gauche.

Remarque 1.17 Fondamentalement, le non-terminal `Index2` est formé d'une boucle qui peut être prolongée à gauche et/ou à droite par des bases non-appariées. Comme les règles de calcul d'énergie sont différentes pour les empilements, les boucles proéminentes et les boucles intérieures, il faut que les fonctions de calcul d'attribut de `Index2`

```
%attribute "int" index
%token A { $$.index = INDEX_A; }
%token C { $$.index = INDEX_C; }
%token G { $$.index = INDEX_G; }
%token U { $$.index = INDEX_U; }
%abbreviation ( A U G C G U ) U A C G U G
%abbreviation • A C G U

%attribute "int" E
%attribute "unsigned short" LS
%attribute "unsigned short" RS
%attribute "int" EDS

%terminal_attribs {}
%default_nonterminal_attribs <E>
%nonterminal_attribs <E LS RS EDS> Index2
%nonterminal_attribs <> SingleStrand

%default_constraint
  {}
  { if ($$.E < $!.E) $!.E = $$.E; }
%constraint Loop
  { if ($$.E > INFINITY / 2) return 0; }
  { if ($$.E < $!.E) $!.E = $$.E; }
%constraint Index2
  {  if ($$.E > INFINITY / 2 ||
        $$.LS + $$.RS > 30)
       return 0;
  }
  {  if ($$.E < $!.E) {
       $!.E = $$.E;    $!.EDS = $$.EDS;
       $!.LS = $$.LS;    $!.RS = $$.RS;
     }
  }
%constraint SingleStrand
  { if ($$.len > 30) return 0; } {}

%%

Structure : Structure Loop
                  { $$.E = $1.E + $2.E; }
          | Loop          { $$.E = $1.E; }
          | Structure • { $$.E = $1.E; }
          | •              { $$.E = 0; };

Loop : ( SingleStrand )
  {  bonus = 0;
     if ($2.len == 4)
       bonus = BonusTetraloop ($2.left);
     $$.E = bonus +
            Hairpin($2.len) +
            Mismatch($1.left,$3.left);
  };
Loop : ( Index2 ) { $$.E = $2.E; }
     | ( LoopSeq )
                  { $$.E = 460 + $2.E; };
```

```
Index2 : Loop
   {  $$.EDS = $1.E;
      $$.LS = $$.RS = 0;
      $$.E = $1.E+DblStrand($1.left-1,$1.right+1);
   };
Index2 : Index2 •
   {  if ($$.right[1] == 0)
        $$.E = INFINITY;
      else {
        $$.EDS = $1.EDS;
        $$.LS = $1.LS;
        $$.RS = $1.RS + 1;
        if ($1.LS == 0)
          $$.E = $$.EDS + Bulge ($$.RS);
        else
          $$.E = $$.EDS +
                 Internal ($$.LS, $$.RS,
                           $$.left, $$.right);
      }
   };
Index2 : • Index2
   {  if ($$.right[1] == 0)
        $$.E = INFINITY;
      else {
        $$.EDS = $2.EDS;
        $$.LS = $2.LS + 1;
        $$.RS = $2.RS;
        if ($2.RS == 0)
          $$.E = $$.EDS + Bulge ($$.LS);
        else
          $$.E = $$.EDS +
                 Internal ($$.LS, $$.RS,
                           $$.left, $$.right);
      }
   };
Index2 : • Index2 •
   {  if ($$.right[1] == 0)
        $$.E = INFINITY;
      else {
        $$.EDS = $2.EDS;
        $$.LS = $2.LS + 1;
        $$.RS = $2.RS + 1;
        $$.E = $$.EDS +
               Internal ($$.LS, $$.RS,
                         $$.left, $$.right);
      }
   };

LoopSeq : LoopSeq Loop
             { $$.E = 10 + $1.E + $2.E; }
        | Loop        { $$.E = 10 + $1.E; }
        | LoopSeq • { $$.E = 40 + $1.E; }
        | •            { $$.E = 40; };

SingleStrand : • {} | SingleStrand • {};
```

Figure 1.11: Grammaire S-attribuée utilisée pour le modèle thermodynamique. Les tables et fonctions C utilisées dans les fonctions d'évaluation d'attributs ne sont pas reportées dans cette figure mais elles reprennent les valeurs données p. 52.

disposent d'un mécanisme leur permettant de choisir quel type de calcul effectuer réellement, en fonction du "passé" du non-terminal `Index2` (i.e. a-t-il déjà été prolongé à gauche et/ou à droite ?). Ce "passé" est donc stocké dans les attributs supplémentaires *EDS*, *LS* et *RS* du non-terminal `Index2`, l'optimisation d'attribut se faisant toujours sur l'énergie E associée à `Index2`.

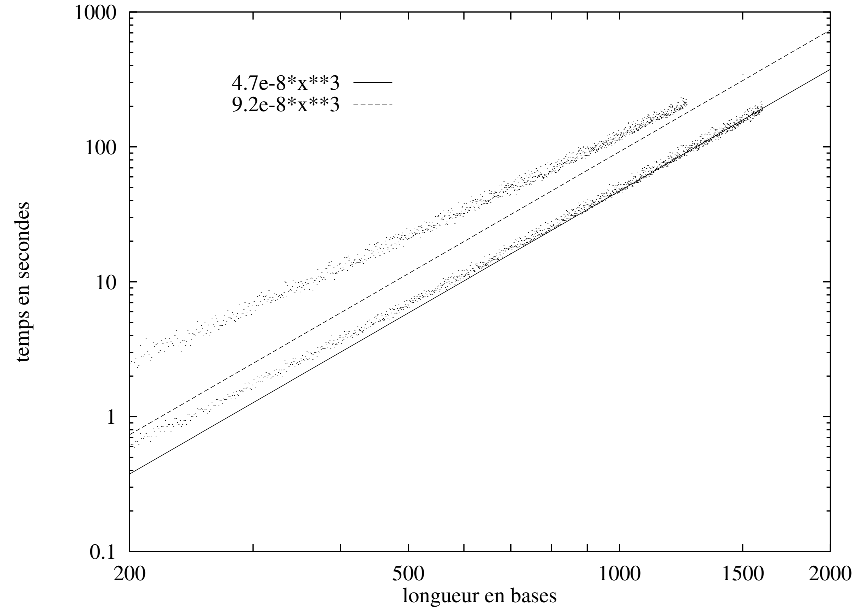
Remarque 1.18 Bien que `Index2` dérive une portion $a_{i+1} \dots a_j$ de la chaîne analysée, il faut tenir compte des terminaux a_i et a_{j+1} pour calculer l'énergie de la boucle d'index 2 définie par `Index2`, car `Index2` n'inclut pas ces terminaux dans ses règles : ils sont utilisés dans la règle `Loop` $\rightarrow$ `( Index2 )`. C'est pour cela que la fonction Internal appelée dans les fonctions d'évaluation d'attributs de `Index2` prend en paramètre la position gauche ($$.left) et la position droite ($$.right) du non-terminal `Index2` qui va être réduit. De plus, l'analyse syntaxique s'effectuant de gauche à droite, il faut toujours vérifier que a_{j+1} est un terminal cohérent, c'est-à-dire que $j+1 \leqslant n$. C'est le rôle du test sur $$.right[1] au début des fonctions d'évaluation d'attributs de `Index2`.

Remarque 1.19 Les syntaxes de `Structure` et `LoopSeq` semblent interchangeables dans la grammaire non-contextuelle, mais elles ne le sont pas dans la grammaire S-attribuée associée. En effet, `Structure` correspond aux boucles extérieures d'énergie nulle, alors que `LoopSeq` utilise la formule affine de calcul d'énergie des boucles multiples.

1.7.2.4 Performances relatives de la programmation dynamique et des grammaires S-attribuées

L'algorithme de programmation dynamique donné en 1.7.2.2, utilisé avec les valeurs d'énergie données p. 52, est au cœur du *Vienna package* disponible sur le serveur `ftp.itc.univie.ac.at` dans `pub/RNA/ViennaRNA-1.03.tar.Z`. Ce logiciel est programmé en C et est donc tout à fait indiqué pour comparer les performances d'une implémentation de l'algorithme de programmation dynamique aux performances d'un analyseur syntaxique utilisant l'algorithme GCP, écrit en C, et généré automatiquement par MTSAG2C à partir de la grammaire donnée figure 1.11. En effet, une analyse rapide de complexité n'est pas suffisante pour départager ces deux méthodes puisqu'elles ont toutes les deux des complexités de $O(n^3)$ en temps et $O(n^2)$ en espace. Tout se joue donc au niveau des constantes multiplicatives.

Comme le programme *Vienna* et l'analyseur syntaxique généré à partir de notre grammaire S-attribuée utilisent le même modèle thermodynamique, ils retournent les mêmes structures secondaires à partir des mêmes ARN. Donc les performances des deux méthodes sont bien comparables. En pratique, les résultats obtenus sur une station Pentium 120 Mhz ont été regroupés en figure 1.12. L'algorithme GCP, tel qu'il est écrit dans l'analyseur syntaxique généré par MTSAG2C, est environ deux fois plus rapide que l'algorithme de programmation dynamique implémenté dans *Vienna* (cf. p. 107 pour une explication de ce phénomène à propos de la grammaire du modèle de Sankoff). Par contre, il utilise près de trois fois plus de mémoire que *Vienna*, 98% de



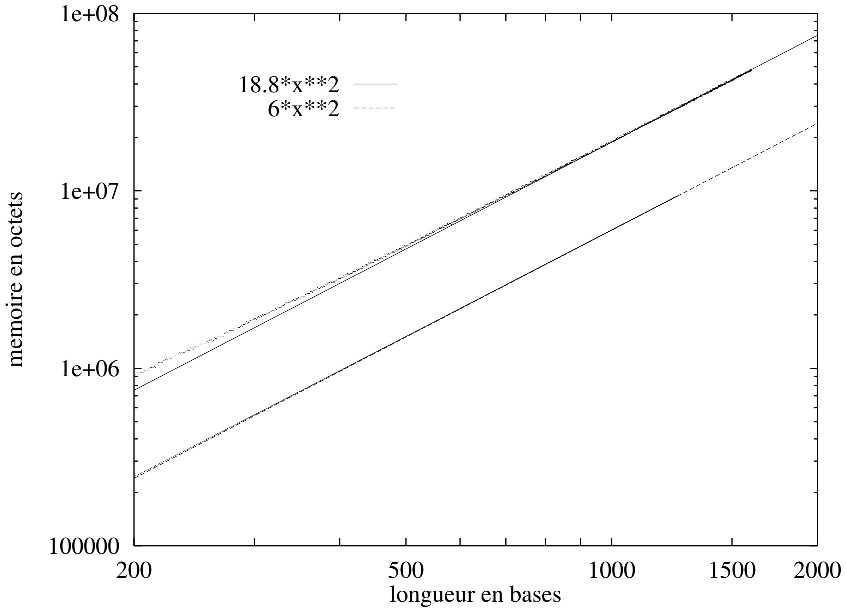


Figure 1.12: Sur le graphe du haut, la série de points au dessus de l'asymptote $9.2 \times 10^{-8}x^3$ correspond au temps de calcul en secondes du programme Vienna, tandis que la série de points au dessus de l'asymptote $4.7 \times 10^{-8}x^3$ correspond au temps de calcul en secondes de l'algorithme GCP. Sur le graphe du bas, la série de points au dessus de l'asymptote $18.8x^2$ correspond à l'espace mémoire utilisé par l'algorithme GCP, tandis que la série de points quasiment confondue avec l'asymptote $6x^2$ correspond à l'espace mémoire utilisé par le programme Vienna.

la mémoire allouée étant réservée par la table $(\mathfrak{T}_{i,j})_{1\leqslant i<j\leqslant n}$ utilisée pour reconstruire les arbres de dérivation. Cette table doit non seulement stocker les attributs définis par l'utilisateur pour tous les non-terminaux (dont EDS, LS et RS pour le non-terminal `Index2`), mais aussi un certain nombre de pointeurs et d'entiers utilisés pour gérer les attributs et la table, alors que les tables $V_{i,j}$, $W_{i,j}$ et $W'_{i,j}$ de l'algorithme de programmation dynamique n'ont à stocker qu'une unique valeur par couple (i, j), soit trois valeurs au total par couple (i, j).

Nous espérons avoir convaincu le lecteur qu'une grammaire S-attribuée est plus facile à écrire que des relations de programmation dynamique et, une fois écrite, beaucoup plus facile à transformer en un programme exploitable, grâce à MTSAG2C. Cette facilité d'écriture n'est pas obtenue au détriment de l'efficacité du code produit, puisque l'analyseur généré par MTSAG2C est plus rapide qu'un programme écrit manuellement, tout en restant raisonnablement gourmand en mémoire.

1.7.3 Application au modèle stochastique

La modélisation stochastique des structures secondaires consiste à attribuer une probabilité à chaque structure secondaire. À la lumière des discussions précédentes sur les grammaires non-contextuelles et leur faculté de codage des structures secondaires par l'intermédiaire d'arbres de dérivation, il est naturel de chercher à obtenir la modélisation stochastique des structures secondaires par une modélisation stochastique des arbres de dérivation d'une grammaire. C'est précisément ce que réalisent les grammaires non-contextuelles stochastiques (ou Stochastic Context-Free Grammars, ou SCFG). Un modèle stochastique n'utilisant pas explicitement les SCFG a été étudié (Eddy & Durbin, 1994). Mais les auteurs de cet autre modèle (appelé Covariance Model, ou CM) reconnaissent qu'il est formellement équivalent aux SCFG, tout en donnant une procédure d'apprentissage de CM qui n'a pas encore été adaptée aux SCFG telles qu'elles sont utilisées dans l'équipe de Haussler (cf. sections 3.9 et 3.10 pour une formulation des SCFG et des CM mettant en évidence les points communs entre les deux modèles). Nous ne détaillerons pas ici les principes et conséquences de l'application des SCFG aux structures secondaires d'ARN (le lecteur intéressé pourra consulter (Sakakibara *et al.*, 1993; Grate *et al.*, 1994; Underwood, 1994; Sakakibara *et al.*, 1994c,a,b; Grate, 1995a; Brown, 1995)), mais nous rappellerons cependant quelques rudiments.

Une SCFG G est une CFG dont chaque règle de dérivation $A \to \alpha$ possède une probabilité $\mathcal{P}(A \to \alpha \mid G)$. La probabilité d'une dérivation $S \to \alpha_1 \to \ldots \to \alpha_k \to u$ pour une SCFG G donnée est donc :

$$\mathcal{P}(S \to \alpha_1 \to \ldots \to \alpha_k \to u \mid G) = \mathcal{P}(S \to \alpha_1 \mid G)\, \mathcal{P}(\alpha_1 \to \alpha_2 \mid G) \ldots \mathcal{P}(\alpha_k \to u \mid G)$$

où la probabilité $\mathcal{P}(\alpha \to \beta \mid G)$ est égale à la probabilité de la règle utilisée dans la dérivation $\alpha \to \beta$. La probabilité d'une séquence u est donc la somme des probabilités précédentes pour toutes les dérivations possibles.

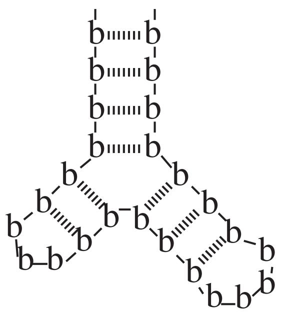
$$\mathcal{P}(u \mid G) = \sum_{S\to\alpha_1\to\ldots\to\alpha_k\to u} \mathcal{P}(S \to \alpha_1 \to \ldots \to \alpha_k \to u \mid G).$$

Il est ensuite fait l'hypothèse que, lorsque la grammaire est bien construite et les probabilités bien choisies, la dérivation de plus forte probabilité représente la structure secondaire cherchée. On cherche donc à trouver $S \to \alpha_1 \to \ldots \to \alpha_k \to u$ telle que

$$\mathcal{P}(S \to \alpha_1 \to \ldots \to \alpha_k \to u \mid G) = \max_{S \to \alpha_1 \to \ldots \to \alpha_k \to u} \mathcal{P}(S \to \alpha_1 \to \ldots \to \alpha_k \to u \mid G).$$

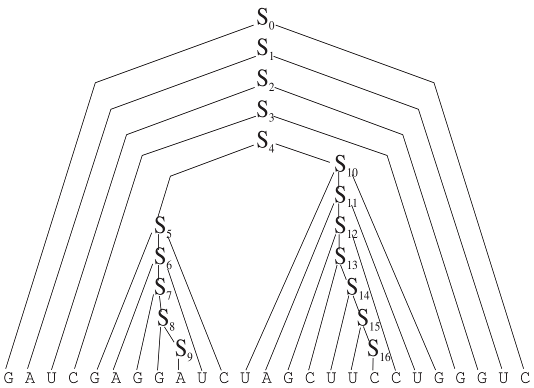
En pratique, étant donnée une famille F_{hom} d'ARN homologues dont la structure consensus est connue, on construit une grammaire G obligeant n'importe quel ARN analysé selon cette grammaire à se replier suivant cette structure consensus. Les probabilités des règles de G sont ensuite ajustées de façon à maximiser la probabilité de dérivation de chaque ARN de F_{hom}. Une fois cette maximisation réalisée, et à condition que G représente bien la structure consensus des ARN de la famille, la probabilité $P(u \mid G)$ d'une séquence u est utilisable pour discriminer les séquences appartenant à F_{hom} des séquences n'y appartenant pas : si la probabilité d'une séquence u est inférieure à un seuil déterminé expérimentalement, il est estimé que u n'appartient pas à la famille, sinon u appartient sans doute à la famille. Ce test de discrimination est potentiellement utilisable pour rechercher dans une base de donnée les séquences appartenant à F_{hom} qui n'auraient pas déjà été annotées comme telles. Les résultats obtenus avec les ARN de transfert ou les petits ARN nucléaires sont très prometteurs (Sakakibara *et al.*, 1994b; Underwood, 1994; Lowe & Eddy, 1996). Leslie Grate a indiqué dans une communication personnelle qu'un analyseur obtenu par MTSAG2C à partir d'une SCFG modifiée par ses soins avait permis de découvrir un ARN de transfert non déjà annoté dans le génome de *Methanococcus jannaschii*.

Les SCFG utilisées par l'équipe de Haussler sont telles que chaque paire de bases ou chaque base isolée d'une structure modélisée est codée par un non-terminal particulier. Des non-terminaux supplémentaires sont introduits pour gérer la variabilité de la structure primaire et les insertions ou les délétions de bases. Par exemple, pour replier les ARN suivant la structure suivante (où les différentes bases ont été remplacées par un b) :



on pourra utiliser la grammaire suivante (aucune règle autorisant l'insertion de bases supplémentaires n'a été ajoutée) :

$$
\begin{array}{ll}
S_0 \to b\ S_1\ b & S_9 \to b \\
S_1 \to b\ S_2\ b & S_{10} \to b\ S_{11}\ b \\
S_2 \to b\ S_3\ b & S_{11} \to b\ S_{12}\ b \\
S_3 \to b\ S_4\ b & S_{12} \to b\ S_{13}\ b \\
S_4 \to S_5\ S_{10} & S_{13} \to b\ S_{14} \\
S_5 \to b\ S_6\ b & S_{14} \to b\ S_{15} \\
S_6 \to b\ S_7\ b & S_{15} \to b\ S_{16} \\
S_7 \to b\ S_8 & S_{16} \to b \\
S_8 \to b\ S_9 &
\end{array}
$$



Cette grammaire possède 17 non-terminaux, chaque non-terminal représentant une paire de bases ou une base particulière dans la structure secondaire.

Ce type de SCFG a pour intérêt supplémentaire de fournir un alignement d'ARN homologues directement à partir des structures produites puisque les bases immédiatement dérivées d'un non-terminal particulier doivent être alignées avec ce non-terminal :

S_0	S_1	S_2	S_3	S_5	S_6	S_7	S_8	S_9	S_6	S_5	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	S_{12}	S_{11}	S_{10}	S_3	S_2	S_1	S_0
G	A	U	C	G	A	G	G	A	U	C	U	A	G	C	U	U	C	C	U	G	G	G	U	C

Les SCFG construites de cette manière ont une taille (nombre de non-terminaux, ou nombre de productions, ou somme des tailles des règles de dérivation) croissant linéairement avec le nombre moyen de bases des ARN de la famille à représenter. Dès que les ARN contiennent plusieurs dizaines de bases, il faut avoir recours à une méta-description de la SCFG à construire, et laisser à un outil spécialement conçu le soin de générer la SCFG à partir de la méta-description. Avec l'outil utilisé dans (Sakakibara *et al.*, 1993), la grammaire précédente possède la méta-description cryptique suivante :

```
S       Helix   4      branch
branch  Branch  rest1  rest2
rest1   Helix   2      loop1
loop1   Loop    3
rest2   Helix   3      loop2
loop2   Loop    4
```

Même si le principe des méta-descriptions facilite l'écriture d'une grammaire représentant une structure consensus d'une famille d'ARN homologues, il ne permet pas à lui seul de trouver cette structure consensus. Les SCFG ne résolvent donc pas le problème de l'alignement et du repliement d'une famille d'ARN homologues dont on ne connaît pas a priori la structure (mais Eddy & Durbin (1994) présentent une méthode adaptée aux CM). Cependant, si une structure consensus d'une famille d'ARN homologues est déjà connue, alors les SCFG permettent de décrire cette structure, et donc d'aligner et de replier les ARN de la famille.

Les SCFG étant toujours des grammaires à ambiguïté exponentielle, l'analyse syntaxique est effectuée en temps $O(|G|n^3)$ et en espace $O(|G|n^2)$. Mais puisque la taille des SCFG augmente linéairement avec la taille moyenne n des ARN à analyser, l'analyse syntaxique suivant ces grammaires est réellement effectuée en temps $O(n^4)$ et en espace $O(n^3)$. Pour pouvoir utiliser les SCFG avec des ARN dépassant la centaine de bases, il faut tenter de décomposer chaque ARN en sous-ARN grâce à l'introduction de points d'ancrage (Grate, 1995a) (cf. section 3.11 p. 164 pour une méthode générique d'introduction des points d'ancrage dans un modèle).

Les probabilités des règles de dérivation des SCFG peuvent être facilement incorporées à des fonctions d'évaluation d'attributs permettant le calcul de la probabilité d'une dérivation. En effet, si $S \to X_1 \dots X_k \to^* u$ est une dérivation telle que $X_i \to^* u_i$ et $u_1 \dots u_k = u$, alors :

$$\mathcal{P}(S \to X_1 \dots X_k \to^* u \mid G) = \mathcal{P}(S \to X_1 \dots X_k \mid G) \prod_{1 \leqslant i \leqslant k} \mathcal{P}(X_1 \to^* u_i \mid G)$$

Donc :

$$f_{S \to X_1 \dots X_k}(x_1 \dots x_k) = \mathcal{P}(S \to X_1 \dots X_k \mid G) \prod_{1 \leqslant i \leqslant k} x_i$$

Si les probabilités de dérivation sont stockées dans un attribut physique `p` des non-terminaux, alors l'ajout d'une contrainte de maximisation de cette probabilité permet de trouver la dérivation de probabilité maximale d'une séquence. En effet, si $S \to X_1 \dots X_k \to^* u$ est une dérivation de probabilité maximale, alors les dérivations $X_i \to^* u_i$ telles que $u_1 \dots u_k = u$ doivent également être des dérivations de probabilité maximale. De plus, l'ajout d'un attribut physique `sum` à chaque non-terminal, dans lequel est stocké la somme des probabilités de dérivation de ce non-terminal en une sous-séquence, permet de calculer simultanément la probabilité de la séquence (cf. figure 1.13).

Nous avons comparé sur deux grammaires l'algorithme GCP à l'analyseur syntaxique de type CYK utilisé par l'équipe de Haussler. La première grammaire utilisée fut celle modélisant les ARN de transfert, décrite dans (Sakakibara *et al.*, 1994b). Cette grammaire possède 75 non-terminaux et 660 productions. La conversion de la grammaire en SAG fut faite en utilisant le principe d'écriture des fonctions d'évaluation d'attributs décrit ci-dessus. L'algorithme de type CYK utilisé par l'équipe de Haussler prend de 2 à 3 secondes pour trouver le meilleur arbre d'analyse sur une DEC AXP 3000/400 (Sakakibara *et al.*, 1994b). L'algorithme GCP prend de 0,5 à 0,6 seconde sur une station équivalente[18]. L'algorithme GCP est donc environ quatre fois plus rapide pour cette SCFG. Mais il faut tempérer ce succès en n'oubliant pas que nos résultats datent de 1996 alors que ceux de Haussler remontent à 1994. L'analyseur syntaxique utilisé par l'équipe de Haussler a sans doute été amélioré depuis[19]. En ce qui concerne la

[18]En fait, les tests sont effectués sur une DEC Alpha 250 4/266, qui est environ trois fois plus rapide qu'une DEC AXP 3000/400. Les temps réellement obtenus vont de 0,18 à 0,2 seconde.

[19]Une communication personnelle de Leslie Grate semble cependant indiquer que les analyseurs générés par MTSAG2C restent les plus performants.

```
%mode "-dynamic"

%token A C G U

%attribute "double" p
%attribute "double" sum

%terminal_attribs {}
%default_nonterminal_attribs < p sum >

%default_constraint
      {}
      { $! .sum += $$.sum;
        if ($$.p > $! .p) $! .p =$$.p;
      }

%rule_model N N
%generic f1 <>
{ $$.sum = $1.sum + $2.sum;
  $$.p   = $1.p   + $2.p;
}

%rule_model T N T
%generic f2 <"double">
{ $$.sum = $2.sum + $#[1];
  $$.p   = $2.p   + $#[1];
}

%rule_model T N
%generic f3 <"double">
{ $$.sum = $2.sum + $#[1];
  $$.p   = $2.p   + $#[1];
}

%rule_model T
%generic f4 <"double">
{ $$.sum = $#[1];
  $$.p   = $#[1];
}

%rule_model N
%generic f5 <"double">
{ $$.sum = $1.sum + $#[1];
  $$.p   = $1.p   + $#[1];
}

%%
START : S_1   S_75   % f1<>;
S_8   : S_9   S_76   % f1<>;
S_76  : S_11  S_77   % f1<>;
S_77  : S_26  S_78   % f1<>;
S_78  : S_27  S_79   % f1<>;
S_79  : S_39  S_63   % f1<>;
S_1 : C  S_2  C   %f2<"-5.47100547437375400">;
S_1 : C  S_2  G   %f2<"-2.71037289647185500">;
S_1 : C  S_2  U   %f2<"-5.45333508881464600">;
S_1 : C  S_2  A   %f2<"-4.13030340107636800">;
S_1 : G  S_2  C   %f2<"-0.4300308912417313">;
S_1 : G  S_2  G   %f2<"-8.08541077499070000">;
S_1 : G  S_2  U   %f2<"-2.81566325181333000">;
S_1 : G  S_2  A   %f2<"-8.17360348702616000">;
S_1 : U  S_2  C   %f2<"-7.92940652651411900">;
S_1 : U  S_2  G   %f2<"-6.96221146477819500">;
S_1 : U  S_2  U   %f2<"-5.39174931134058900">;
S_1 : U  S_2  A   %f2<"-2.58038284337185000">;
S_1 : A  S_2  C   %f2<"-5.41485118280398800">;
S_1 : A  S_2  G   %f2<"-8.15602834220465300">;
S_1 : A  S_2  U   %f2<"-2.19594839200723600">;
S_1 : A  S_2  A   %f2<"-7.87533930524384300">;
:
:
S_96: C  S_96     %f3<"-0.9124555954203570">;
S_96: G  S_96     %f3<"-1.94122514086505400">;
S_96: U  S_96     %f3<"-1.36568421681609400">;
S_96: A  S_96     %f3<"-2.66946911554397500">;
S_96: C  S_46     %f3<"-5.70208391802062000">;
S_96: G  S_46     %f3<"-2.25657020491459800">;
S_96: U  S_46     %f3<"-4.23057674154701400">;
S_96: A  S_46     %f3<"-4.84762398583680100">;
%%
```

Figure 1.13: Extrait de la grammaire utilisée pour les ARN de transfert. Les cinq fonctions génériques f1, f2, f3, f4, f5 représentent les cinq types de règles utilisés dans la grammaire. Les quatre dernières fonctions prennent en paramètre la probabilité de la règle telle que définie dans la SCFG originale. La notation $\$\#[n]$ représente la valeur $n^{\text{ème}}$ paramètre d'une fonction générique.

place prise en mémoire, l'algorithme utilise 1,4 Moctets, hors code. Nous ne disposons pas du résultat pour l'algorithme CYK.

La deuxième grammaire utilisée fut la grammaire U1 pour les petits ARN nucléaires (ou snRNA), construite par Rebecca Underwood (Underwood, 1994; Grate, 1995b). Les snRNA ont une longueur moyenne de 156 bases et cette grammaire compte 462 non-terminaux et 2021 productions. Underwood reporte un temps moyen d'analyse d'environ 30 secondes pour 128 Moctets de mémoire allouée sur une Decstation 5000/240. L'algorithme GCP prend 6,5 secondes et 25 Mo de mémoire sur une DEC Alpha 250 4/266, soit environ 30 secondes et 25 Moctets de mémoire s'il avait été exécuté sur une Decstation 5000/240. Sur cet exemple, notre analyseur est aussi rapide qu'un algorithme de type CYK, mais il occupe 4 fois moins de mémoire. L'examen de la grammaire montre que l'algorithme GCP considère quasiment toutes les dérivations qu'aurait considéré l'algorithme CYK. Nous avons ici un exemple de grammaire ambiguë à un point tel que l'algorithme CYK reste adapté à l'analyse syntaxique pour cette grammaire, au moins en ce qui concerne le temps de calcul. Nous verrons dans la section 3.9.2 p. 150 comment diminuer encore la quantité de mémoire allouée et le temps d'exécution de l'algorithme GCP grâce à une formulation différente des SCFG.

1.8 Conclusion

Nous avons rappelé dans ce chapitre qu'une grammaire non-contextuelle peut représenter efficacement une famille de structures secondaires d'ARN par l'intermédiaire des arbres de dérivation des ARN. Lorsque la famille de structures secondaires possède des éléments structuraux caractéristiques, ces éléments peuvent être codés dans la grammaire sous la forme de non-terminaux et de règles de dérivation spécifiques. Ce type de codage des structures d'ARN a été déjà utilisé avec succès sous la forme de grammaires stochastiques. Son application au modèle thermodynamique est une nouveauté puisque, jusqu'à présent, le modèle thermodynamique était invariablement associé à des équations de programmation dynamique telles que celles données par Zuker (Zuker, 1989b). Alors que les équations de programmation dynamique sont difficilement adaptables à des contraintes structurales particulières[20], les grammaires non-contextuelles se prêtent naturellement à de telles adaptations. De fait, l'usage des grammaires non-contextuelles supprime le reproche habituellement adressé au modèle

[20]Les stratégies destinées à forcer une structure et habituellement utilisées avec les équations de programmation dynamique sont :

- forcer un appariement connu d'un ARN donné en lui attribuant une énergie arbitrairement négative, donc arbitrairement favorable,
- forcer une base connue d'un ARN donné à ne pas s'apparier en attribuant une énergie arbitrairement positive, donc arbitrairement défavorable, à tout appariement impliquant cette base,
- forcer un type de repliement donné (forme de trèfle par exemple), en "bidouillant" les tables donnant les énergies des boucles jusqu'à ce que "ça marche en moyenne".

Ces stratégies sont donc beaucoup moins propres et efficaces que celles rendues possibles par l'introduction des grammaires non-contextuelles (cf. section 1.2.3).

thermodynamique : ne pas garantir des repliements semblables pour des ARN d'une même famille, contrairement aux grammaires stochastiques ou aux méthodes phylogénétiques.

Les grammaires non-contextuelles ont pu être utilisées avec le modèle thermodynamique grâce à leur extension sous forme de grammaires S-attribuées. Schématiquement, les non-terminaux d'une grammaire non-contextuelles représentent les variables des équations de programmation dynamique, tandis que les règles et fonctions d'évaluation d'attribut représentent les relations entre variables. Une conséquence intéressante de la forme générale des grammaires S-attribuées est leur capacité à intégrer le modèle des grammaires stochastiques comme simple cas particulier.

Nous avons également montré dans ce chapitre que les grammaires S-attribuées ne sont pas simplement un moyen d'exprimer élégamment des modèles tels que le modèle thermodynamique ou le modèle stochastique. En effet, l'algorithme GCP introduit en même temps que les grammaires S-attribuées s'est montré plus rapide, tel que nous l'avons implémenté (cf. chapitre 4), qu'une implémentation déjà disponible des équations de programmation dynamique du modèle thermodynamique ou qu'un autre algorithme d'analyse syntaxique utilisé avec les grammaires S-attribuées.

Cependant, les grammaires S-attribuées n'apportent pas, à elles seules, de solution au problème de la détermination des repliements réels des ARN. En effet, si l'introduction de contraintes structurales dans le modèle thermodynamique améliore la qualité des repliements trouvés, elle n'est possible que s'il y a connaissance préalable d'éléments structuraux. De même, une grammaire stochastique représentant une famille de structures ne peut être créée qu'à partir de la donnée de cette famille. Il semble donc que, pour obtenir une structure correcte d'un ARN à l'aide des modèles précédents, il faille déjà connaître cette structure, ou du moins ses éléments principaux ! Nous verrons au chapitre 3 plusieurs techniques permettant de briser ce cercle vicieux et reposant toutes sur l'analyse comparative d'alignements d'ARN.

Chapitre 2

Grammaires multi-bandes et alignements

Le chapitre précédent nous a permis de montrer que les grammaires S-attribuées sont parfaitement adaptées à la description de deux modèles importants de recherche de structures secondaires d'ARN sans pseudo-nœud : le modèle thermodynamique popularisé par Zuker et le modèle stochastique introduit par Haussler *et al.* Nous avons de plus introduit un algorithme d'analyse syntaxique, l'algorithme GCP, et un générateur automatique d'analyseurs syntaxiques utilisant cet algorithme, qui montrent que les grammaires S-attribuées ne sont pas simplement un formalisme élégant mais qu'elles peuvent en outre conduire à des programmes performants.

Cependant, aussi bien le modèle thermodynamique que le modèle stochastique ne sont pas suffisants pour résoudre à eux seuls le problème de la détermination des structures secondaires d'ARN. Nous allons donc généraliser les grammaires S-attribuées en remplaçant les grammaires non-contextuelles sous-jacentes par des grammaires non-contextuelles multi-bandes. Nous verrons plus tard que cette généralisation permet de reformuler à l'aide de grammaires la plupart des principales méthodes utilisées dans le domaine de l'analyse informatique de séquences biologiques. Ce chapitre a pour but de :

- définir les grammaires non-contextuelles multi-bandes,
- donner une première esquisse de leurs applications potentielles,
- généraliser l'algorithme GCP aux grammaires S-attribuées multi-bandes,
- déterminer la complexité de l'algorithme obtenu.

Un détail des applications pratiques, suffisamment nombreuses pour mériter un chapitre à part, puis quelques repères théoriques seront fournis aux chapitres 3 et 5 respectivement.

2.1 Définitions

Le but de cette section est de définir un classe spéciale d'alphabets, dits alphabets m-bandes, ainsi que les outils associés. Ces alphabets m-bandes permettront d'exprimer simplement les problèmes d'alignement de séquences biologiques sous la forme de problèmes de dérivation de mots. La généralisation des CFG, en tant que formalisme adapté à la dérivation de mots, puis la généralisation des SAG, en tant que formalisme adapté à la sélection d'arbres de dérivations, seront alors deux conséquences naturelles de l'introduction des alphabets m-bandes.

Définition 2.1 (Token, Bande)
Soit m un entier naturel strictement positif. Soit $\left(\Sigma_k\right)_{1\leqslant k\leqslant m}$ une collection de m alphabets. Nous appellerons **token** *tout élément de Σ_k. Nous appellerons* **bande** *toute chaîne de tokens appartenant à Σ_k^*. La chaîne vide de Σ_k^* sera notée ϵ. L'entier m sera appelé le* **nombre de bandes***.*

Notation 2.1 (Projection sur une bande) Lorsqu'une variable $\vec{v}$ a une valeur dite m-bandes, c'est-à-dire définie comme vecteur à m dimensions, nous noterons $\vec{v}^{(k)}$ la valeur de la k-ième composante de $\vec{v}$, également appelée valeur de $\vec{v}$ sur la bande k.

Définition 2.2 (index m-bandes)
Un index m-bandes $\vec{\imath}$ est un vecteur de $\mathbb{Z}^m$. Nous dirons qu'un index m-bandes $\vec{\imath}$ est inférieur à un index m-bandes $\vec{\jmath}$ si cet ordre est valable sur chaque bande, et nous noterons cette relation d'ordre partielle par $\vec{\imath} \leqslant \vec{\jmath}$. Nous noterons $\vec{\imath} < \vec{\jmath}$ s'il existe de plus une bande k telle que $\vec{\imath}^{(k)} < \vec{\jmath}^{(k)}$.

Nous utiliserons les notations $\vec{0}$ (resp. $\vec{1}$) pour désigner un index m-bandes valant 0 (resp. 1) sur chaque bande. La somme (resp. la différence) de deux index m-bandes $\vec{\imath}_1$ et $\vec{\imath}_2$ est un index m-bandes $\vec{\jmath}$ défini par $\vec{\jmath}^{(k)} = \vec{\imath}_1{}^{(k)} + \vec{\imath}_2{}^{(k)}$ (resp. $\vec{\jmath}^{(k)} = \vec{\imath}_1{}^{(k)} - \vec{\imath}_2{}^{(k)}$). Donc, $\vec{\imath} \leqslant \vec{\jmath}$ signifie également $\vec{\jmath} - \vec{\imath} \geqslant \vec{0}$.

Définition 2.3 (Mot m-bandes)
Étant donnés m alphabets Σ_k ($1 \leqslant k \leqslant m$), un mot m-bandes est un vecteur de m bandes $(a^{(1)}_{\vec{1}^{(1)}} \dots a^{(1)}_{\vec{n}^{(1)}}, \dots, a^{(m)}_{\vec{1}^{(m)}} \dots a^{(m)}_{\vec{n}^{(m)}})$, où chaque bande $a^{(k)}_{\vec{1}^{(k)}} \dots a^{(k)}_{\vec{n}^{(k)}}$ appartient à Σ_k^*. Dans le cas d'une bande k vide, alors $\vec{n}^{(k)} = 0$ et la notation $a^{(k)}_{\vec{1}^{(k)}} \dots a^{(k)}_{\vec{n}^{(k)}}$ est alors équivalente à la chaîne vide. Nous noterons l'ensemble des mots ainsi définis par $\prod \Sigma_k^* = \bigotimes_{k=1\dots n} \Sigma_k^*$.

Plus simplement, et par homogénéité par rapport au cas mono-bande, les mots m-bandes $(a^{(1)}_{\vec{1}^{(1)}} \dots a^{(1)}_{\vec{n}^{(1)}}, \dots, a^{(m)}_{\vec{1}^{(m)}} \dots a^{(m)}_{\vec{n}^{(m)}})$ seront notés $a_{\vec{1}} \dots a_{\vec{n}}$. Les sous-mots de $a_{\vec{1}} \dots a_{\vec{n}}$, ou facteurs de $a_{\vec{1}} \dots a_{\vec{n}}$, seront notés $a_{\vec{\imath}} \dots a_{\vec{\jmath}} = \left(a^{(1)}_{\vec{\imath}^{(1)}} \dots a^{(1)}_{\vec{\jmath}^{(1)}}, \dots, a^{(m)}_{\vec{\imath}^{(m)}} \dots a^{(m)}_{\vec{\jmath}^{(m)}}\right)$ avec les conventions usuelles $\vec{1} \leqslant \vec{\imath}$, $\vec{\jmath} \leqslant \vec{n}$ et, si $\vec{\imath}^{(k)} > \vec{\jmath}^{(k)}$ alors $a^{(k)}_{\vec{\imath}^{(k)}} \dots a^{(k)}_{\vec{\jmath}^{(k)}} = \epsilon$ (la chaîne vide).

Exemple 2.1 *$(abba, dcd)$ est un mot 2-bandes sur les alphabets $\Sigma_1 = \{a, b\}$ et $\Sigma_2 = \{c, d\}$. Nous écrirons aussi ce mot 2-bandes en notation superposée, c'est-à-dire $\left[\begin{smallmatrix} abba \\ dcd \end{smallmatrix}\right]$, car cette notation est plus naturelle dans le contexte des alignements. Ce mot 2-bandes $a_{\overrightarrow{\binom{1}{1}}} \ldots a_{\overrightarrow{\binom{4}{3}}}$ a un sous-mot 2-bandes $a_{\overrightarrow{\binom{2}{1}}} \ldots a_{\overrightarrow{\binom{3}{2}}}$ représenté par $\left[\begin{smallmatrix} bb \\ dc \end{smallmatrix}\right]$.*

Définition 2.4 (alphabet m-bandes)
Un alphabet m-bandes $\vec{\Sigma}$ est un produit de m alphabets Σ_k auxquels on a ajouté la chaîne vide, privé du produit vide sur toutes les bandes :

$$\vec{\Sigma} = \Big(\bigotimes_{k=1\ldots m} (\Sigma_k \cup \{\epsilon\}) \Big) \setminus \bigotimes_{k=1\ldots m} \{\epsilon\}.$$

Nous pouvons maintenant donner la définition d'un alignement de m bandes. Nous aurons besoin de cette définition pour introduire les grammaires non-contextuelles générant les structures d'alignements.

Définition 2.5 (Alignement m-bandes)
Un élément $a_1 \ldots a_l$ du monoïde libre $\vec{\Sigma}^*$, généré par concaténation formelle d'éléments de $\vec{\Sigma}$, est appelé un alignement m-bandes de longueur l. L'alignement vide de $\vec{\Sigma}^*$ est noté ϵ, au même titre qu'une bande vide de Σ_k^*. Les alignements m-bandes sont donc les chaînes de $\vec{\Sigma}^*$.

Lorsque la concaténation d'éléments de l'alphabet m-bandes $\vec{\Sigma}$ n'est plus formelle, c'est-à-dire lorsqu'il est tenu compte de la présence éventuelle de ϵ sur une ou plusieurs bandes, nous obtenons un mot m-bandes. Les alignements m-bandes et les mots m-bandes sont donc liés par ce que nous appellerons une ϵ-suppression.

Définition 2.6 (ϵ-suppression)
Étant donné un alignement m-bandes $a_1 \ldots a_l$, un mot m-bandes est obtenu par concaténation des symboles de la projection de $a_1 \ldots a_l$ sur chaque bande. Cette concaténation définit un homomorphisme τ_m de $\vec{\Sigma}^*$ dans $\prod \Sigma_k^*$, homomorphisme que nous noterons également $\langle\rangle$.

$$\begin{aligned} \tau_m : \vec{\Sigma}^* &\longrightarrow \prod \Sigma_k^* \\ a_1 \ldots a_l &\longmapsto a_{\vec{1}} \ldots a_{\vec{n}} = \tau_m(a_1 \ldots a_l) = \langle a_1 \ldots a_l \rangle \end{aligned}$$

À la lumière de la notation $\langle\rangle$ désignant l'ϵ-suppression, le monoïde $\prod \Sigma_k^*$ sera aussi noté $\langle \vec{\Sigma}^* \rangle$.

Exemple 2.2 *Le mot 2-bandes $\left[\begin{smallmatrix} abba \\ dcd \end{smallmatrix}\right]$ peut être défini comme une ϵ-suppression de l'alignement $\left\langle \left[\begin{smallmatrix} \epsilon \\ d \end{smallmatrix}\right] \left[\begin{smallmatrix} a \\ \epsilon \end{smallmatrix}\right] \left[\begin{smallmatrix} b \\ \epsilon \end{smallmatrix}\right] \left[\begin{smallmatrix} b \\ c \end{smallmatrix}\right] \left[\begin{smallmatrix} a \\ d \end{smallmatrix}\right] \right\rangle$ ou de l'alignement $\left\langle \left[\begin{smallmatrix} a \\ \epsilon \end{smallmatrix}\right] \left[\begin{smallmatrix} b \\ d \end{smallmatrix}\right] \left[\begin{smallmatrix} \epsilon \\ c \end{smallmatrix}\right] \left[\begin{smallmatrix} b \\ \epsilon \end{smallmatrix}\right] \left[\begin{smallmatrix} a \\ d \end{smallmatrix}\right] \right\rangle$, pour ne citer que deux alignements 2-bandes parmi les 50 alignements 2-bandes de longueur 5 de ce mot $\left[\begin{smallmatrix} abba \\ dcd \end{smallmatrix}\right]$.*

Remarque 2.1 Tout alignement m-bandes provient d'un unique mot m-bandes, mais la réciproque n'est pas vraie comme le montre l'exemple précédent. Le monoïde $\langle\vec{\Sigma}^*\rangle$ n'est donc pas un monoïde libre dès que $m > 1$, contrairement à $\vec{\Sigma}^*$. En fait, le nombre d'alignements m-bandes générant un même mot m-bandes par ϵ-suppression croît exponentiellement avec la taille de ce mot m-bandes. Dans ce contexte, le pointillé dans la notation $a_{\vec{\imath}} \ldots a_{\vec{\jmath}}$ d'un mot m-bandes ne doit pas être imaginé comme étant une ellipse d'une concaténation formelle multi-bandes possédant une expansion unique, même s'il est possible de noter, pour $\vec{k}$ tel que $\vec{\imath} \leqslant \vec{k} \leqslant \vec{\jmath}$, $a_{\vec{\imath}} \ldots a_{\vec{\jmath}} = a_{\vec{\imath}} \ldots a_{\vec{k}}\, a_{\vec{k}+\vec{1}} \ldots a_{\vec{\jmath}}$.

La définition des alignements que nous venons de donner est une généralisation de concepts qui peuvent être trouvés dans (Myers, 1991; Searls & Murphy, 1995). La différence principale entre cette définition et les définitions des auteurs des travaux précédents réside dans l'identification explicite faite du rapport existant entre les alignements et les chaînes d'un alphabet m-bandes. Cette identification est fondamentale puisqu'elle permet d'introduire les grammaires non-contextuelles m-bandes (ou MTCFG, pour m-tape context-free grammars, ou multi-tape context-free grammars) comme simples extensions des grammaires non-contextuelles habituelles.

Définition 2.7 (Grammaire non-contextuelle m-bandes)
Une grammaire non-contextuelle m-bandes $\vec{G} = (\vec{V}_T, V_N, P, S)$ est composée d'un ensemble fini de terminaux $\vec{V}_T$ tel que $\vec{V}_T$ **est un alphabet m-bandes**, *d'un ensemble fini de non-terminaux V_N tel que $V_N \cap \vec{V}_T = \emptyset$, d'un ensemble fini P de productions (dites aussi règles de réécriture ou règles de dérivation), et d'un symbole de départ, ou axiome, $S \in V_N$. On note $V = V_N \cup \vec{V}_T$ le vocabulaire de la grammaire.*

Chaque production de P est de la forme $A \to \alpha$, où $A \in V_N$ et $\alpha \in V^*$. Le non-terminal A est la partie gauche de la production et α est sa partie droite. Les productions sont donc aussi des relations sur $V_N \times V^*$.

Cette définition ne se distingue de la définition des grammaires non-contextuelles donnée p. 17 que par le remplacement de l'alphabet V_T par un alphabet m-bandes $\vec{V}_T$. Mais ce changement de type d'alphabet a une conséquence importante sur le langage généré par une MTCFG $\vec{G}$. Ce langage est l'ensemble des mots m-bandes obtenus par ϵ-suppression d'un alignement m-bandes dérivable de S :

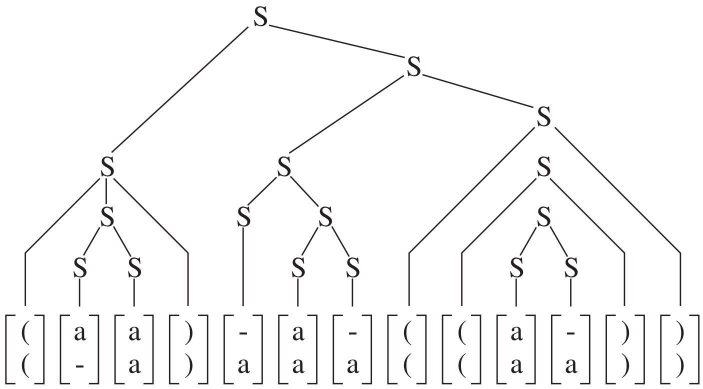
$$L(\vec{G}) = \left\{ \langle u \rangle \in \langle \vec{V}_T^* \rangle \;\middle|\; S \to^* u \right\}$$

Notons que cette définition du langage généré par $\vec{G}$ est la même que la définition usuelle dans le cas où $m = 1$. L'analyse d'un mot m-bandes revient donc à trouver les dérivations conduisant aux alignements m-bandes u tels que $\langle u \rangle$ est ce mot m-bandes.

Exemple 2.3 *La MTCFG suivante aligne deux expressions correctement parenthésées contenant uniquement des parenthèses et des a. Nous avons remplacé le symbole ϵ par un tiret '$-$' parce que cette notation est plus appropriée, et même attendue, dans le contexte des alignements.*

$$S \to \left[\begin{smallmatrix} (\\ (\end{smallmatrix}\right] S \left[\begin{smallmatrix}) \\ ) \end{smallmatrix}\right] \;\Big|\; \left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right] \;\Big|\; \left[\begin{smallmatrix} a \\ - \end{smallmatrix}\right] \;\Big|\; \left[\begin{smallmatrix} - \\ a \end{smallmatrix}\right] \;\Big|\; S\,S$$

Deux bandes peuvent être alignées par cette MTCFG si la structure définie par les parenthèses est équivalente sur les deux bandes. Par exemple, $(aa)a((a))$ et $(a)aaa((aa))$ sont équivalentes, et le mot 2-bandes $\genfrac{[}{]}{0pt}{}{(aa)a((a))}{(a)aaa((aa))}$ appartient au langage généré par la grammaire car c'est une ϵ-suppression d'un alignement généré par la grammaire, tel que l'alignement obtenu par la dérivation suivante :



Par contre, $(aa)a((a))$ et $(aa)a(a)((a))$ ou $((aa))a((a))$ ne sont pas équivalentes et il n'existe aucun alignement dont une ϵ-suppression pourrait donner $\genfrac{[}{]}{0pt}{}{(aa)a((a))}{(aa)a(a)((a)}$ ou $\genfrac{[}{]}{0pt}{}{(aa)a((a))}{((aa))a((a))}$.

Exemple 2.4 *La figure 2.1 est un exemple de structure d'alignement obtenue pour deux ARN avec une grammaire dont la version abrégée est la suivante :*

$$
\begin{aligned}
DS &\to \genfrac{[}{]}{0pt}{}{(}{(}DS\genfrac{[}{]}{0pt}{}{)}{)} \mid \genfrac{[}{]}{0pt}{}{(}{(}SS\genfrac{[}{]}{0pt}{}{)}{)} \mid \genfrac{[}{]}{0pt}{}{(}{(}SS\ DS\ DS\genfrac{[}{]}{0pt}{}{)}{)}\\
DS &\to \genfrac{[}{]}{0pt}{}{-}{(}DS\genfrac{[}{]}{0pt}{}{-}{)} \mid \genfrac{[}{]}{0pt}{}{-}{(}SS\genfrac{[}{]}{0pt}{}{-}{)} \mid \genfrac{[}{]}{0pt}{}{-}{(}SS\ DS\ DS\genfrac{[}{]}{0pt}{}{-}{)}\\
DS &\to \genfrac{[}{]}{0pt}{}{(}{-}DS\genfrac{[}{]}{0pt}{}{)}{-} \mid \genfrac{[}{]}{0pt}{}{(}{-}SS\genfrac{[}{]}{0pt}{}{)}{-} \mid \genfrac{[}{]}{0pt}{}{(}{-}SS\ DS\ DS\genfrac{[}{]}{0pt}{}{)}{-}\\
SS &\to \genfrac{[}{]}{0pt}{}{.}{.} \mid \genfrac{[}{]}{0pt}{}{-}{.} \mid \genfrac{[}{]}{0pt}{}{.}{-}\\
SS &\to \genfrac{[}{]}{0pt}{}{.}{.}SS \mid \genfrac{[}{]}{0pt}{}{-}{.}SS \mid \genfrac{[}{]}{0pt}{}{.}{-}SS
\end{aligned}
$$

En réalité, les couples de tokens (parenthèse ouvrante, parenthèse fermante) situés sur une même bande le long d'une règle de grammaire doivent être remplacés par les différentes paires de bases possibles dans les ARN et le token point par les différentes bases possibles (cf. p. 242 pour la définition de la syntaxe des tokens abrégés). La grammaire alors obtenue compte 192 règles, dont quelques unes sont reproduites ci-dessous.

$$
\begin{aligned}
DS \to\ & \genfrac{[}{]}{0pt}{}{A}{A}DS\genfrac{[}{]}{0pt}{}{U}{U} \mid \genfrac{[}{]}{0pt}{}{A}{U}DS\genfrac{[}{]}{0pt}{}{U}{A} \mid \genfrac{[}{]}{0pt}{}{A}{G}DS\genfrac{[}{]}{0pt}{}{U}{C} \mid \genfrac{[}{]}{0pt}{}{A}{C}DS\genfrac{[}{]}{0pt}{}{U}{G} \mid \genfrac{[}{]}{0pt}{}{A}{G}DS\genfrac{[}{]}{0pt}{}{U}{U} \mid \genfrac{[}{]}{0pt}{}{A}{U}DS\genfrac{[}{]}{0pt}{}{U}{G} \mid\\
& \genfrac{[}{]}{0pt}{}{U}{A}DS\genfrac{[}{]}{0pt}{}{A}{U} \mid \genfrac{[}{]}{0pt}{}{U}{U}DS\genfrac{[}{]}{0pt}{}{A}{A} \mid \genfrac{[}{]}{0pt}{}{U}{G}DS\genfrac{[}{]}{0pt}{}{A}{C} \mid \genfrac{[}{]}{0pt}{}{U}{C}DS\genfrac{[}{]}{0pt}{}{A}{G} \mid \genfrac{[}{]}{0pt}{}{U}{G}DS\genfrac{[}{]}{0pt}{}{A}{U} \mid \genfrac{[}{]}{0pt}{}{U}{U}DS\genfrac{[}{]}{0pt}{}{A}{G} \mid\\
& \genfrac{[}{]}{0pt}{}{G}{A}DS\genfrac{[}{]}{0pt}{}{C}{U} \mid \genfrac{[}{]}{0pt}{}{G}{U}DS\genfrac{[}{]}{0pt}{}{C}{A} \mid \genfrac{[}{]}{0pt}{}{G}{G}DS\genfrac{[}{]}{0pt}{}{C}{C} \mid \genfrac{[}{]}{0pt}{}{G}{C}DS\genfrac{[}{]}{0pt}{}{C}{G} \mid \genfrac{[}{]}{0pt}{}{G}{G}DS\genfrac{[}{]}{0pt}{}{C}{U} \mid \genfrac{[}{]}{0pt}{}{G}{U}DS\genfrac{[}{]}{0pt}{}{C}{G} \mid\\
& \genfrac{[}{]}{0pt}{}{C}{A}DS\genfrac{[}{]}{0pt}{}{G}{U} \mid \genfrac{[}{]}{0pt}{}{C}{U}DS\genfrac{[}{]}{0pt}{}{G}{A} \mid \genfrac{[}{]}{0pt}{}{C}{G}DS\genfrac{[}{]}{0pt}{}{G}{C} \mid \genfrac{[}{]}{0pt}{}{C}{C}DS\genfrac{[}{]}{0pt}{}{G}{G} \mid \genfrac{[}{]}{0pt}{}{C}{G}DS\genfrac{[}{]}{0pt}{}{G}{U} \mid \genfrac{[}{]}{0pt}{}{C}{U}DS\genfrac{[}{]}{0pt}{}{G}{G} \mid\\
& \genfrac{[}{]}{0pt}{}{G}{A}DS\genfrac{[}{]}{0pt}{}{U}{U} \mid \genfrac{[}{]}{0pt}{}{G}{U}DS\genfrac{[}{]}{0pt}{}{U}{A} \mid \genfrac{[}{]}{0pt}{}{G}{G}DS\genfrac{[}{]}{0pt}{}{U}{C} \mid \genfrac{[}{]}{0pt}{}{G}{C}DS\genfrac{[}{]}{0pt}{}{U}{G} \mid \genfrac{[}{]}{0pt}{}{G}{G}DS\genfrac{[}{]}{0pt}{}{U}{U} \mid \genfrac{[}{]}{0pt}{}{G}{U}DS\genfrac{[}{]}{0pt}{}{U}{G} \mid\\
& \genfrac{[}{]}{0pt}{}{U}{A}DS\genfrac{[}{]}{0pt}{}{G}{U} \mid \genfrac{[}{]}{0pt}{}{U}{U}DS\genfrac{[}{]}{0pt}{}{G}{A} \mid \genfrac{[}{]}{0pt}{}{U}{G}DS\genfrac{[}{]}{0pt}{}{G}{C} \mid \genfrac{[}{]}{0pt}{}{U}{C}DS\genfrac{[}{]}{0pt}{}{G}{G} \mid \genfrac{[}{]}{0pt}{}{U}{G}DS\genfrac{[}{]}{0pt}{}{G}{U} \mid \genfrac{[}{]}{0pt}{}{U}{U}DS\genfrac{[}{]}{0pt}{}{G}{G}
\end{aligned}
$$

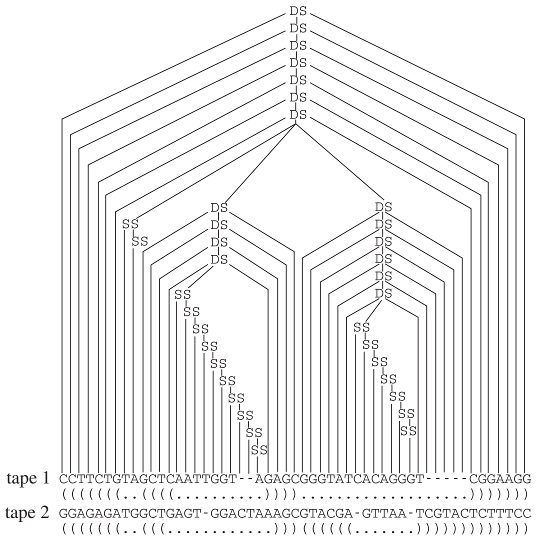


Figure 2.1: Arbre de dérivation d'un alignement de deux ARN. Les appariements sont déduits des dérivations de DS (Double-Strand) et ils sont donnés en dessous de chaque bande. Notons qu'un Double-Strand a été défini comme une structure dont les extrémités doivent être appariées sur au moins une bande, alors qu'un Single-Strand (SS) ne peut avoir sur chaque bande que des bases non-appariées.

La notion de CFG propre (p. 17) s'étend naturellement aux MTCFG. Lorsque nous généraliserons l'algorithme GCP (p. 32) aux MTCFG, nous supposerons une fois encore que la grammaire à laquelle est appliqué l'algorithme généralisé est une grammaire propre.

Pour étudier la complexité d'analyse des MTCFG par l'algorithme GCP m-bandes (nous appellerons ainsi la généralisation de l'algorithme GCP), nous aurons besoin de la notion de grammaire projetée. Une grammaire projetée est simplement la projection des règles d'une MTCFG sur une bande.

Définition 2.8 (Grammaire projetée)
Soit $\vec{G} = (\vec{V}_T, V_N, P, S)$ une grammaire non-contextuelle m-bandes. Pour chaque bande k ($1 \leqslant k \leqslant m$), la grammaire projetée $\vec{G}^{(k)} = (\vec{V}_T^{(k)}, V_N, P^{(k)}, S)$ est définie par :

$$\vec{V}_T^{(k)} = \{a^{(k)} \mid a \in \vec{V}_T\}$$
$$P^{(k)} = \{A \to u_0^{(k)} B_1 u_1^{(k)} \dots B_j u_j^{(k)} \mid j \geqslant 0, u_j \in \vec{V}_T^*, A \to u_0 B_1 u_1 \dots B_j u_j \in P\}$$

Par convention, la projection de tout non-terminal A sur la bande k est égale à ce non-terminal : $A^{(k)} = A$.

Exemple 2.5 *La MTCFG de l'exemple 2.3* a la même grammaire projetée sur chaque bande :

$$S \to (\ S\ ) \quad | \quad \epsilon \quad | \quad a \quad | \quad S\ S$$

Remarque 2.2 Une MTCFG peut être propre sans qu'aucune de ses grammaires projetées ne soit propre. La MTCFG de l'exemple 2.3 et ses grammaires projetées données dans l'exemple précédent en sont une illustration. Rappelons cependant qu'il est toujours possible de transformer une grammaire quelconque en une grammaire propre (cf. remarque 1.1 p. 18).

Enfin, les grammaires S-attribuées m-bandes (MTSAG pour Multi-Tape, ou m-tape, S-Attribute Grammars) sont obtenues comme extensions des grammaires non-contextuelles m-bandes, au même titre que les grammaires S-attribuées sont des extensions de grammaires non-contextuelles. La définition suivante est donc une quasi-copie de la définition donnée p. 39.

Définition 2.9 (Grammaire S-attribuée m-bandes)
Une grammaire S-attribuée m-bandes notée $\vec{G} = (\vec{V_T}, V_N, P, S, \mathcal{A}, S_\mathcal{A}, F_P)$ est une extension de la grammaire non-contextuelle m-bandes $\vec{G} = (\vec{V_T}, V_N, P, S)$, où l'on adjoint à chaque $X \in V$ un attribut x appartenant à un ensemble d'attributs $\mathcal{A}$, et à chaque élément $\alpha \in V^*$ une chaîne d'attributs λ de $\mathcal{A}^*$. $S_\mathcal{A}$ est une fonction de $\vec{V_T}$ dans $\mathcal{A}$ affectant un attribut à chaque terminal. F_P est un ensemble de fonctions de $\mathcal{A}^*$ dans $\mathcal{A}$, notées $f_{A\to\alpha}$ dont le rôle est de calculer l'attribut de chaque non-terminal présent dans une dérivation. Une fonction $f_{A\to\alpha}$ appartient à F_P si et seulement si $A \to \alpha$ appartient à P.

2.2 Analyse syntaxique des MTSAG

De nombreux aspects techniques liés à l'analyse syntaxique des MTSAG ont été reportés dans les section 5.1 et 5.2 afin de ne pas diluer les éléments fondamentaux de l'analyse syntaxique dans un monceau de détails non strictement nécessaires à la compréhension.

Les principes fondamentaux ayant permis la création de l'algorithme CYK ou de l'algorithme de Earley n'ont pas changé avec l'apparition des bandes multiples. Il est en fait relativement facile d'imaginer une généralisation de ces algorithmes visant à leur faire traiter des MTCFG plutôt que des CFG. Mais puisque nous avons choisi au chapitre 1 d'utiliser l'algorithme GCP pour analyser une unique bande, il est logique de généraliser l'algorithme GCP plutôt qu'un autre au cas m-bandes.

Ce choix n'est pas simplement dicté par une volonté de réutilisation d'un code déjà écrit. Il repose également sur le fait que l'algorithme GCP est plus efficace que l'algorithme de Earley, grâce à la non génération des états n'appartenant pas au noyau et à la factorisation des états de mêmes préfixes gauches. De plus, nous verrons que la diminution des complexités d'analyse de l'algorithme GCP lorsque la SAG est non-ambiguë, ou même LR(k), est beaucoup plus utile aux MTSAG qu'elle ne pouvait l'être aux SAG[1].

[1] Nous avons utilisé les SAG dans le contexte d'une recherche de structures secondaires d'ARN. Ce contexte étant synonyme de grammaires à ambiguïté exponentielle, la diminution de complexité pour les grammaires non-ambiguës semblait ne présenter au mieux qu'un intérêt théorique. L'étude

Étant données une MTSAG et ses contraintes de sélection et d'optimisation, l'algorithme GCP décrit au chapitre 1 construit essentiellement n listes d'analyse $(T_j)_{(1 \leqslant j \leqslant n)}$ contenant chacune une liste d'états d'analyse $[\Delta \rightarrowtail \alpha, i, \lambda]$ tels que :

$$\forall A \in \Delta, \exists(\beta, \gamma), S \to^* a_1 \ldots a_i A\gamma \to a_1 \ldots a_i \alpha\beta\gamma \to^* a_1 \ldots a_j \beta\gamma. \tag{2.1}$$

La chaîne d'attributs λ est associée à α, et chaque attribut de λ est optimal au sens des contraintes de sélection (définition 2.14) et d'optimisation (définition 2.13). En l'absence de contrainte de sélection, tous les états d'analyse vérifiant la condition (2.1) apparaissent effectivement dans les listes d'analyse.

Pour l'essentiel, la généralisation de l'algorithme GCP aux MTSAG consiste à remplacer les différents index apparaissant dans l'algorithme par des index m-bandes. Ainsi, les états attribués s'écriront $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ au lieu de $[\Delta \rightarrowtail \alpha, i, \lambda]$. Cependant, comme de petites adaptations non réductibles à un simple changement de notation seront nécessaires, nous prendrons le temps de détailler la présentation de l'algorithme GCP m-bandes et de souligner les différences entre les deux versions. Le plan de la présentation sera calqué sur celui utilisé au chapitre 1.

Définition 2.10 (État attribué m-bandes)
Soit $\vec{G} = (\vec{V}_T, V_N, P, S, \mathcal{A}, S_{\mathcal{A}}, F_P)$ une MTSAG propre. Nous appellerons état attribué m-bandes tout objet de la forme $\left[\{A_1, \ldots, A_k\} \rightarrowtail \alpha, \vec{\imath}, \lambda\right]$ tel que $k > 0$ et tel que, pour tout A_j $(1 \leqslant j \leqslant k)$, il existe β tel que $A_j \to \alpha\beta$. La variable λ est une chaîne des attributs des symboles de α.

Comme dans le cas mono-bande, les états attribués m-bandes seront appelés états lorsqu'aucune confusion ne sera possible.

Définition 2.11 (Liste d'analyse m-bandes)
Soit $a_{\vec{1}} \ldots a_{\vec{n}} \in \langle \vec{V}_T^* \rangle$ un mot m-bandes à analyser. Une liste d'analyse m-bandes $T_{\vec{\jmath}}$ est une liste d'états attribués m-bandes $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ construits de manière à vérifier la condition minimum suivante d'utilité a priori : pour tout état attribué m-bandes $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda] \in T_{\vec{\jmath}}$ et pour tout $A \in \Delta$, il existe β, γ, u et v tels que :

$$S \to^* uA\gamma \to u\alpha\beta\gamma \to^* uv\beta\gamma, \quad \langle u \rangle = a_{\vec{1}} \ldots a_{\vec{\imath}} \quad \text{et} \quad \langle v \rangle = a_{\vec{\imath}+\vec{1}} \ldots a_{\vec{\jmath}} \tag{2.2}$$

Nous verrons que la réciproque est vraie et que, en l'absence de contrainte de sélection, l'existence de A, β et γ tels que la condition (2.2) est vérifiée signifie qu'il existe $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda] \in T_{\vec{\jmath}}$ tel que $A \in \Delta$ (cf. théorème 2.1).

La condition minimum d'utilité a été légèrement modifiée pour tenir compte de la notion d'ϵ-suppression et de la nouvelle définition du langage généré par $\vec{G}$. Le mot $a_{\vec{1}} \ldots a_{\vec{n}}$ appartient à $L(\vec{G})$ si et seulement si il existe $[\Delta \rightarrowtail \alpha, \vec{0}, \lambda] \in T_{\vec{n}}$ tel que $S \in \Delta$ et $S \to \alpha$.

---

de l'algorithme GCP généralisé aux MTSAG montrera quel usage peut être fait de cette diminution de la complexité d'analyse.

Il n'y a pas de liste d'analyse $T_{\vec{0}}$, et toutes les listes d'analyse $T_{\vec{\jmath}}$ telles que $\vec{0} < \vec{\jmath} \leqslant \vec{n}$ sont à calculer. Le nombre de listes d'analyse est donc égal à $\left(\prod_{1\leqslant k\leqslant m}(\vec{n}^{(k)}+1)\right)-1$. Ce nombre est bien égal à n lorsque $m=1$.

Les fonctions Reduce, Expect, $\mathsf{Initial}$ et Goto dépendant seulement des règles de la grammaire et de la notion de dérivation, et non du nombre de bandes et de la notation utilisée pour ces bandes, leur définition reste inchangée. Ce n'est pas le cas des ensembles de non-terminaux attendus par une liste d'analyse m-bandes.

Définition 2.12 (Ensemble $\Delta_{\vec{\jmath}}$ des non-terminaux attendus par $T_{\vec{\jmath}}$)
Une fois $T_{\vec{\jmath}}$ construit, l'ensemble des non-terminaux attendus pour prolonger au moins un état de $T_{\vec{\jmath}}$ est noté $\Delta_{\vec{\jmath}}$. Cet ensemble est défini pour $\vec{\jmath} > \vec{0}$ par :

$$\Delta_{\vec{\jmath}} = \bigcup_{[\Delta\rightarrowtail\alpha,\vec{\imath},\lambda]\in T_{\vec{\jmath}}} \mathsf{Expect}([\Delta \rightarrowtail \alpha])$$

L'ensemble $\Delta_{\vec{0}}$ est obtenu de la même manière que Δ_0:

$$\Delta_{\vec{0}} = \{A \in V_N \mid \exists\beta, S \rightarrow^* A\beta\}$$

Les contraintes d'optimisation et de sélection utilisées dans l'algorithme GCP sont aussi à modifier.

Définition 2.13 (Contrainte d'optimisation m-bandes)
Soit $\vec{G}$ une MTSAG et $a_{\vec{1}} \dots a_{\vec{n}}$ un mot m-bandes analysé. Soit x l'attribut d'une dérivation $A \rightarrow \alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow u$, et soit x' l'attribut d'une autre dérivation $A \rightarrow \alpha'_1 \rightarrow \cdots \rightarrow \alpha'_k \rightarrow u$, avec $\langle u\rangle = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$.

La contrainte d'optimisation m-bandes $\mathcal{C}_A$ associée au non-terminal A prend en paramètre $\vec{\imath}$, $\vec{\jmath}$, x, x' et retourne un nouvel attribut x'' déduit de ces quatre paramètres. On écrira $x'' = \mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x, x')$.

Définition 2.14 (Contrainte de sélection m-bandes)
Soit $(A, \vec{\imath}, x) := (A, \vec{\imath}, f_{A\rightarrow\alpha(\lambda)})$ un triplet obtenu pendant le calcul de $T_{\vec{\jmath}}$ par réduction d'une règle $A \rightarrow \alpha$ dans l'étape 5 de l'algorithme GCP m-bandes.

La contrainte de sélection $\mathcal{C}_A^{\mathcal{S}}$ associée au non-terminal A est un prédicat prenant en paramètre $\vec{\imath}$, $\vec{\jmath}$ et x et retournant vrai *si le triplet $(A, \vec{\imath}, x)$ doit être sélectionné, et* faux *sinon.*

Le dernier élément à modifier avant de présenter l'algorithme GCP m-bandes est la file de priorité des triplets (X, i, x) de l'algorithme GCP. Cette file de priorité a été introduite afin d'éviter d'utiliser l'attribut x d'un symbole X alors toutes que les dérivations possibles de X en $a_{i+1} \dots a_j$ n'ont pas encore été considérées, et donc que le calcul de x est incomplet. Rappelons qu'un triplet (X', i', x') est ajouté à la file de priorité pendant le calcul de T_j lorsque X' est dérivable en $a_{i'+1} \dots a_j$ avec un attribut x'. Dans le cas mono-bande, un symbole X d'attribut x et dérivable en $a_{i+1} \dots a_j$ a la possibilité d'influencer X' uniquement si $i' < i$ ou si $i' = i$ et $X' \rightarrow^+ X$. Pour laisser à X une chance d'influencer X', (X, i, x) doit être extrait de la file de priorité avant (X', i', x'). Ce principe est généralisable à chaque bande d'une dérivation m-bandes. On en déduit la définition suivante des files de priorité m-bandes.

Définition 2.15 (File de priorité m-bandes)
Soit Q un ensemble de triplets $(X, \vec{\imath}, x) \in V \times \{\vec{\imath} \mid \vec{0} \leqslant \vec{\imath} < \vec{n}\} \times \mathcal{A}$. Soient $(X, \vec{\imath}, x)$ et $(X', \vec{\imath}', x')$ deux éléments de Q. Nous dirons que $(X', \vec{\imath}', x') < (X, \vec{\imath}, x)$, ou que l'extraction de $(X, \vec{\imath}, x)$ est prioritaire sur celle de $(X', \vec{\imath}', x')$ si $\vec{\imath}' < \vec{\imath}$ ou si $\vec{\imath}' = \vec{\imath}$ et $X' \to^+ X$. Si $\vec{\imath}$ et $\vec{\imath}'$ ne sont pas comparables, alors les deux triplets peuvent être extraits indépendamment l'un de l'autre. La grammaire $\vec{G}$ étant propre la relation $<$ définit un ordre partiel sur les triplets. Nous noterons $\mathsf{Extract}$ la fonction retournant et supprimant un triplet maximum $(X, \vec{\imath}, x)$ de Q. Nous noterons cette extraction $(X, \vec{\imath}, x) := \mathsf{Extract}(Q)$.

Munis des définitions précédentes, nous pouvons maintenant présenter l'algorithme GCP m-bandes.

Algorithme 2.1 (Algorithme GCP m-bandes)
Soit Q la file de triplets $(X, \vec{\imath}, x)$ qui vient d'être définie avec sa fonction $\mathsf{Extract}$. Au départ, les listes d'analyse $T_{\vec{\jmath}}$ sont vides. Soit $\Delta_{\vec{0}} = \{A \in V_N \mid \exists \beta, S \to^* A\beta\}$. Pour $\vec{\jmath}$ allant de $\vec{0}$ (exclus) à $\vec{n}$ (inclus), en ordre croissant d'index m-bandes[2], exécuter les étapes suivantes :

1. *Initialiser la file Q:*
   $$Q := \left\{ \big(a, \vec{\imath}, S_{\mathcal{A}}(a)\big) \;\middle|\; \vec{\jmath} - \vec{1} \leqslant \vec{\imath} < \vec{\jmath} \;\wedge\; a \in \vec{V}_T \;\wedge\; \tau_m(a) = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}} \right\}$$
2. *Extraction du triplet à traiter en priorité :*
   $(X, \vec{\imath}, x) := \mathsf{Extract}(Q)$
3. *Création des états attribués commençant par X :*
   $$T_{\vec{\jmath}} := T_{\vec{\jmath}} \cup \left\{ [\Delta \rightarrowtail X, \vec{\imath}, x] \;\middle|\; [\Delta \rightarrowtail X] = \mathsf{Initial}(\Delta_{\vec{\imath}}, X) \right\}$$
4. *Prolongement d'états attribués par X :*
   $$T_{\vec{\jmath}} := T_{\vec{\jmath}} \cup \left\{ [\Delta \rightarrowtail \alpha X, \vec{h}, \lambda x] \;\middle|\; \begin{array}{l} \exists [\Delta' \rightarrowtail \alpha, \vec{h}, \lambda] \in T_{\vec{\imath}}, \\ [\Delta \rightarrowtail \alpha X] = \mathsf{Goto}([\Delta' \rightarrowtail \alpha], X) \end{array} \right\}$$
5. *Pour chaque état attribué $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ ajouté à $T_{\vec{\jmath}}$ par une des étapes 3 et 4, et pour chaque $A \in \mathsf{Reduce}([\Delta \rightarrowtail \alpha])$, calculer le triplet $(A, \vec{\imath}, x) := \big(A, \vec{\imath}, f_{A \to \alpha}(\lambda)\big)$.*

   Si $\mathcal{C}^{\mathcal{S}}_A(\vec{\imath}, \vec{\jmath}, x)$ est faux alors passer directement à l'étape 6.

   S'il existe déjà un triplet $(A, \vec{\imath}, x')$ appartenant à Q, remplacer l'attribut x' de ce triplet par $\mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x, x')$. Si aucun triplet de ce type n'existe dans Q, ajouter $(A, \vec{\imath}, x)$ à Q.
6. *Répéter les étapes 2 à 6 tant que Q n'est pas vide;*
7. *Calculer $\Delta_{\vec{\jmath}} := \mathsf{Expect}(T_{\vec{\jmath}})$.*

Le mot m-bandes $a_{\vec{1}} \dots a_{\vec{n}}$ appartient à $L(\vec{G})$ si et seulement si le triplet $(S, \vec{0}, x)$ a été extrait de Q pendant le calcul de $T_{\vec{n}}$. On dit alors que le mot a été reconnu par l'algorithme et qu'il a un attribut optimal x.

[2]Autrement dit, si $\vec{\jmath}_1 < \vec{\jmath}_2$ alors $\vec{\jmath}_1$ doit être considéré avant $\vec{\jmath}_2$, ce qui fournit facilement un ordre total à partir de l'ordre partiel. Voir à ce sujet la section 4.1.2 p. 183.

Grâce aux notations introduites au début de ce chapitre, et particulièrement celle d'index m-bandes, l'algorithme GCP m-bandes ne diffère de l'algorithme GCP mono-bande qu'en un petit nombre d'endroits.

- L'ordre de calcul des listes d'analyse $T_{\vec{\jmath}}$ est moins explicite que celui des listes d'analyse T_j. Nous introduirons en section 4.1.3 p. 184 un code sur les index m-bandes éliminant cet inconvénient.
- Plusieurs triplets $\big(a, \vec{\imath}, S_{\mathcal{A}}(a)\big)$ sont ajoutés à Q lors de la première étape, au lieu d'un seul triplet. C'est à ce moment que l'ambiguïté de lecture sur les bandes est prise en compte.

Les autres étapes restent identiques à celles de l'algorithme GCP mono-bande, modulo le remplacement des index par des index m-bandes. Lorsque $m = 1$, l'algorithme GCP m-bandes est évidemment rigoureusement équivalent à l'algorithme GCP mono-bande.

Théorème 2.1 (Utilité a priori des états de l'algorithme GCP m-bandes) *Lorsque les listes d'analyse $T_{\vec{\jmath}}$ sont construites comme décrit dans l'algorithme 2.1, et lorsque les contraintes de sélection sont inutilisées, les deux propositions suivantes sont équivalentes :*

1. $[\Delta \rightarrowtail \alpha, \vec{\imath}] \in T_{\vec{\jmath}}$
2. *Pour tout $A \in \Delta$, il existe β, γ, u et v tels que*

$$S \to^* uA\gamma \to u\alpha\beta\gamma \to^* uv\beta\gamma, \quad \langle u\rangle = a_{\vec{1}} \dots a_{\vec{\imath}} \quad et \quad \langle v\rangle = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$$

Démonstration : elle est en tous points équivalente à celle du théorème 1.1, les index étant remplacés par des index m-bandes et les terminaux par des terminaux m-bandes. Il suffit juste de prendre garde à la nouvelle forme d'initialisation de Q. L'introduction de contraintes de sélection vient évidemment modifier ce résultat puisque seul $1 \Rightarrow 2$ reste valide (cf. remarque 1.13 p. 45). □

La suite de la démonstration de la correction de l'algorithme GCP m-bandes reprend les arguments utilisés p. 30, p. 33 et p. 44 pour démontrer la correction des différentes évolutions de l'algorithme GCP mono-bande.

2.2.1 Reconstruction de l'arbre m-bandes optimal

La méthode utilisée n'a pas changé depuis la section 1.6.3 p. 46. Cependant, l'apparition des index m-bandes modifie les complexités obtenues.

Lorsqu'un triplet $(X, \vec{\imath}, x)$ est extrait de Q pendant le calcul de $T_{\vec{\jmath}}$, son apparition est mémorisée dans une table particulière $(\mathfrak{T}_{\vec{\imath},\vec{\jmath}})_{1 \leqslant \vec{\imath} < \vec{\jmath} \leqslant n}$ initialement vide :

$$\mathfrak{T}_{\vec{\imath},\vec{\jmath}} := \mathfrak{T}_{\vec{\imath},\vec{\jmath}} \cup \big\{(X, x)\big\}$$

Le coût de stockage de cette table et de ses éléments est $O\Big(\prod_{1 \leqslant k \leqslant m} \big(\vec{n}^{(k)}\big)^2\Big)$ dans le cas général, mais il est facile d'implémenter cette table de façon à tenir compte des diminutions de complexité de calcul en espace sur chaque bande (cf. section 2.3).

Notons $\mathfrak{R}$ la fonction qui à $(A, \vec{\imath}, \vec{\jmath}, x)$ associe un arbre de dérivation de $A \to^* u$ possédant un attribut x et tel que $\langle u \rangle = a_{\vec{\imath}+\vec{1}} \ldots a_{\vec{\jmath}}$.

Proposition 2.1 (Reconstruction de l'arbre m-bandes optimal)
Soit r le nombre maximum de non-terminaux apparaissant dans les parties droites α des règles $A \to \alpha$ de $\vec{G}$. Soit x l'attribut optimal d'une dérivation $S \to^* u$ telle que $\langle u \rangle = a_{\vec{1}} \ldots a_{\vec{n}}$. Alors $\mathfrak{R}(S, \vec{0}, \vec{n}, x)$ retourne l'arbre de dérivation associé à l'attribut x en temps
$O\Big(\prod\limits_{1 \leqslant k \leqslant m} \big(\vec{n}^{(k)}\big)^r \Big)$.

Démonstration : Le calcul de $\mathfrak{R}(A, \vec{\imath}, \vec{\jmath}, x)$ s'effectue toujours en deux temps:

1. Recherche d'une dérivation permettant de calculer l'attribut optimal x de A parmi l'ensemble des dérivations valides de A, c'est-à-dire telles que :
$$\begin{gathered} A \to u_{-1} B_0 u_0 \ldots B_p u_p \to^* u \\ \langle u \rangle = a_{\vec{\imath}+\vec{1}} \ldots a_{\vec{\jmath}} \\ B_l \to^* v_l \quad (0 \leqslant l \leqslant p) \\ B_l \text{ a un attribut optimal } x_l \\ \langle v_l \rangle = a_{\vec{s}_l+1} \ldots a_{\vec{e}_l} \end{gathered}$$
2. Si $m \neq -1$ (i.e. si $\langle u_{-1} \rangle \neq a_{\vec{\imath}+\vec{1}} \ldots a_{\vec{\jmath}}$), appeler récursivement $\mathfrak{R}(B_l, \vec{s}_l, \vec{e}_l, x_l)$ $(0 \leqslant l \leqslant p)$ pour connaître le reste de l'arbre.

Nous allons montrer que
$$\mathrm{Temps}\big(\mathfrak{R}(A, \vec{\imath}, \vec{\jmath}, x)\big) = O\Big(\prod_{1 \leqslant k \leqslant m} \big(\vec{\jmath}^{(k)} - \vec{\imath}^{(k)}\big)^{r-1} \Big) + \sum_{l=0}^{p} \mathrm{Temps}\big(\mathfrak{R}(B_l, \vec{s}_l, \vec{e}_l)\big),$$
ce qui est suffisant pour assurer $\mathrm{Temps}\big(\mathfrak{R}(A, \vec{\imath}, \vec{\jmath}, x)\big) = O\Big(\prod\limits_{1 \leqslant k \leqslant m} \big(\vec{\jmath}^{(k)} - \vec{\imath}^{(k)}\big)^{r} \Big)$.

Soit $A \to u_{-1} B_0 u_0 \ldots B_p u_p$ une règle de la grammaire ayant A pour non-terminal à gauche. Remarquons que :

- il y a un nombre fini de telles règles,
- $p \leqslant r - 1$;
- il y a $O\Big(\prod_{1 \leqslant k \leqslant m} \big(\vec{\jmath}^{(k)} - \vec{\imath}^{(k)} - (|\langle u_{-1} \rangle^{(k)}| + \cdots + |\langle u_p \rangle^{(k)}|)\big)^p \Big)$ manières de décomposer cette règle entre $a_{\vec{\imath}+\vec{1}}$ et $a_{\vec{\jmath}}$.
- une décomposition est valide lorsque :
$$\forall l, 1 \leqslant l \leqslant p, (B_l, s_l) \in \mathfrak{T}_{\vec{s}_k, \vec{e}_k}.$$
Cette vérification s'effectue en p étapes.

La recherche d'une décomposition valide du non-terminal A peut donc se faire en temps $O\Big(\prod_{1\leqslant k\leqslant m}\big(\vec{\jmath}^{(k)}-\vec{\imath}^{(k)}\big)^{r-1}\Big)$. Une fois la décomposition fixée, il suffit d'appeler $\mathfrak{R}$ pour obtenir les sous-arbres des non-terminaux B_k de la décomposition. $\square$

2.3 Complexités de l'algorithme GCP m-bandes

Théorème 2.2
Soient $\vec{G}$ une MTSAG m-bandes propre et $a_{\vec{1}}\dots a_{\vec{n}}$ une chaîne analysée. Soient $\vec{G}^{(k)}$ ($1\leqslant k\leqslant m$) ses projections sur les m bandes. Chaque grammaire $\vec{G}^{(k)}$ possède une grammaire propre équivalente notée $\vec{G}_P^{(k)}$ à laquelle des fonctions d'évaluation d'attributs peuvent être adjointes[3].

La complexité en temps (resp. en espace) de l'algorithme GCP m-bandes pour $\vec{G}$ est égale au produit des complexités en temps (resp. en espace) de l'algorithme GCP mono-bande pour les grammaires $\vec{G}_P^{(k)}$ et les mots $a^{(k)}_{\vec{1}^{(k)}}\dots a^{(k)}_{\vec{n}^{(k)}}$.

Démonstration : Soit $[\Delta\rightarrowtail X_1\dots X_p,\vec{\imath}_1,x_1\dots x_p]$ un état attribué de $T_{\vec{\jmath}}$. Bien que cela ne soit pas explicitement apparent dans cette notation, les attributs $x_1,\dots,x_p$ sont des attributs optimaux de sous-dérivations de limites connues. Ainsi, x_1 est l'attribut optimal de X_1 pour des dérivations $X_1\rightarrow^* u_1$ tel que $\langle u_1\rangle = a_{\vec{\imath}_1+\vec{1}}\dots a_{\vec{\imath}_2}$ avec un index $\vec{\imath}_2$ connu, mais ayant disparu de l'état attribué tel que noté ci-dessus. Nous noterons donc maintenant cet état $\big[\Delta\rightarrowtail X_1\dots X_p,\vec{\imath}_1,(\vec{\imath}_1,x_1)\dots(\vec{\imath}_p,x_p)\big]$ pour marquer le fait que :

$$\begin{array}{llll} x_1 & \text{est l'attribut optimal des dérivations} & X_1\rightarrow^* u_1 & \langle u_1\rangle = a_{\vec{\imath}_1+\vec{1}}\dots a_{\vec{\imath}_2}\\ \vdots & & \vdots & \vdots\\ x_p & \text{est l'attribut optimal des dérivations} & X_p\rightarrow^* u_p & \langle u_p\rangle = a_{\vec{\imath}_p+\vec{1}}\dots a_{\vec{\jmath}} \end{array}$$

Nous noterons également $\vec{\imath}_{p+1}=\vec{\jmath}$ (ne pas confondre $\vec{\imath}_{p+1}$ et $\vec{\imath}_p+\vec{1}$).

Les opérations effectuées par l'algorithme GCP m-bandes pour créer cet état sont reproductibles sur l'algorithme GCP m-bandes projeté sur la bande k. Ces opérations projetées conduisent à l'ajout d'un état

$$\big[\Delta\rightarrowtail X_1^{(k)}\dots X_p^{(k)},\vec{\imath}_1^{(k)},(\vec{\imath}_1^{(k)},x_1)\dots(\vec{\imath}_p^{(k)},x_p)\big]$$

à $T_{\vec{\jmath}^{(k)}}$. Si l'on montre que l'ajout de cet état lorsque $\vec{\imath}_1^{(k)}\neq\vec{\jmath}^{(k)}$ implique l'ajout d'un état semblable par l'algorithme GCP mono-bande appliqué à la grammaire projetée $\vec{G}^{(k)}$, alors le théorème est une conséquence immédiate car il suffit de faire l'hypothèse la plus défavorable d'indépendance des bandes les unes par rapport aux autres et de faire les produits des complexités d'analyse sur chaque bande.

---

[3]Ces fonctions ne sont pas nécessairement sémantiquement équivalentes aux fonctions de la grammaires $\vec{G}$. Elles doivent simplement être calculables en $O(1)$, comme les fonctions de $\vec{G}$.

Soit $A \in \Delta$. Nous allons montrer que l'algorithme GCP mono-bande appliqué à $\vec{G}_P^{(k)}$ ajoute en fait à $T_{\vec{\jmath}^{(k)}}$ un état

$$[\Delta' \rightarrowtail X_{q_1}^{(k)} \dots X_{q_l}^{(k)}, \vec{\imath}_{q_1}^{(k)}, (\vec{\imath}_{q_1}^{(k)}, x'_{q_1}) \dots (\vec{\imath}_{q_l}^{(k)}, x_{q'_l})] \tag{2.3}$$

tel que $A \in \Delta'$, où les $X_{q_i}^{(k)}$ $(1 \leqslant i \leqslant l)$ forment le sous-ensemble des $X_1 \dots X_p$ initiaux tel que $\vec{\imath}_{q_i}^{(k)} < \vec{\imath}_{q_i+1}^{(k)}$ (i.e. les symboles ayant une dérivation de projection vide sur la bande k ont été supprimés).

La démonstration est triviale si pour tout q $(1 \leqslant q \leqslant p)$, $\vec{\imath}_q^{(k)} < \vec{\imath}_{q+1}^{(k)}$ puisqu'aucune projection de symbole ne correspond à une dérivation vide.

Si tous les symboles supprimés sont des terminaux, alors la démonstration est tout aussi triviale puisque les opérations d'extension d'états par des terminaux sont appliquées aux bandes sur lesquelles ces terminaux n'ont pas de projection vide.

Si certains symboles supprimés sont des non-terminaux, alors cela signifie que ces non-terminaux sont dérivables en ϵ pour la grammaire projetée $\vec{G}^{(k)}$. Le processus de transformation de la grammaire $\vec{G}^{(k)}$ en grammaire propre $\vec{G}_P^{(k)}$ a pour effet d'introduire des règles où ces non-terminaux dérivables en ϵ n'apparaissent pas et l'état (2.3) est donc un état valide de $\vec{G}_P^{(k)}$.

Notons que tous les symboles ne peuvent être supprimés puisque $\vec{\imath}_1{}^{(k)} \neq \vec{\jmath}^{(k)}$. $\square$

Remarque 2.3 L'existence de ce théorème, dont nous verrons de nombreuses applications dans le chapitre suivant, montre qu'il était utile de choisir un algorithme d'analyse syntaxique mono-bande plus performant que l'algorithme de Cocke-Younger-Kasami qui aurait conduit en général à des complexités d'analyse de $O\left(\prod_{1 \leqslant k \leqslant m} \left(n^{(k)}\right)^3\right)$ en temps et $O\left(\prod_{1 \leqslant k \leqslant m} \left(n^{(k)}\right)^2\right)$ en espace.

Remarque 2.4 La table 1.2 de la page 48 récapitule quelques complexités connues dans le cas mono-bande.

Remarque 2.5 Les complexités évaluées grâce au théorème 2.2 sont des bornes supérieures obtenues en supposant l'indépendance des bandes les unes par rapport aux autres. C'est loin d'être systématiquement le cas comme le montre la grammaire 2-bandes suivante :

$$S \rightarrow S\left[\begin{smallmatrix}(\\(\end{smallmatrix}\right]S\left[\begin{smallmatrix})\\)\end{smallmatrix}\right] \;\mid\; \left[\begin{smallmatrix}a\\a\end{smallmatrix}\right]$$

Les deux projections identiques de cette grammaires sont $LR(0)$ et non-récursives à droite donc l'analyse syntaxique est théoriquement au plus quadratique en temps et espace. Mais comme aucun terminal ne contient de symbole ϵ, les alignements générés ne peuvent avoir de gap et la grammaire est en fait équivalente à ses grammaires projetées. Donc l'algorithme GCP m-bandes effectue l'analyse syntaxique en temps et espace linéaires pour cette grammaire 2-bandes.

Chapitre 3

Applications

Dans ce chapitre, nous allons présenter plusieurs applications des MTSAG. Sept d'entre elles seront des reformulations parfois importantes de modèles déjà décrits dans la littérature. Nous verrons en particulier une réécriture 2-bandes intéressante des SCFG déjà décrites en 1.7.3. Les autres exemples illustreront la puissance (recherche des pseudo-nœuds) ou la souplesse (points d'ancrage et contraintes d'alignement) du formalisme des MTSAG. Chaque application de la liste suivante sera décrite dans une section séparée :

- alignements de séquences par distance d'édition,
- introduction des contraintes d'alignements dans les MTSAG,
- alignement et repliement simultanés des ARN à l'aide du modèle de Sankoff,
- introduction d'une généralisation du modèle de Sankoff,
- alignement des ARN à l'aide du modèle de Corpet et Michot,
- alignement des ARN à l'aide des modèles de Bafna *et al.*,
- alignement d'arbres ordonnés,
- formulation 2-bandes des modèles de Markov cachés,
- formulation 2-bandes des SCFG,
- formulation 2-bandes des modèles de covariance de Eddy et Durbin,
- introduction de points d'ancrage dans les MTSAG,
- proposition de méthode fiable d'alignement et de repliement des ARN,
- recherche de pseudo-nœuds dans les structures secondaires d'ARN.

Toutes les MTSAG décrites dans ce chapitre sont des MTSAG 2-bandes, bien que certains résultats se généralisent aisément à 3 bandes, compte non tenu des problèmes d'augmentation de complexité d'analyse. Les MTSAG sont ensuite transformées automatiquement en analyseur syntaxique utilisant l'algorithme GCP m-bandes présenté dans le chapitre précédent. L'outil de transformation automatique employé est MT-SAG2C. Cet outil utilise une syntaxe d'écriture des MTSAG proche de celle de YACC, et il est nécessaire de lire l'annexe A avant ce chapitre, ou en parallèle avec celui-ci, pour comprendre parfaitement les divers exemples de MTSAG. Un lecteur déjà familiarisé avec la syntaxe de YACC sera avantagé. Cependant, une maîtrise superficielle de la syntaxe de MTSAG2C est suffisante pour comprendre les idées sous-tendant chaque application.

3.1 Alignements par distance d'édition

Nous allons voir dans cette section que les équations de programmation dynamique, qui sont habituellement associées aux différents modèles d'alignement de séquences par distance d'édition, peuvent très bien être clarifiées à l'aide des MTSAG.

L'alignement de séquences nucléiques ou de séquences d'acides aminés est un problème qui, du fait de son importance en biologie moléculaire, a été souvent étudié et pour lequel de nombreuses solutions ont été proposées (Myers, 1991; Chan *et al.*, 1992; Abdeddaim, 1996; Sagot, 1996). Nous allons illustrer ce problème par un exemple. Les cinq lignes suivantes forment un alignement de cinq fragments d'ARN :

```
CAACCCUUAUCC-UUUGUAACUCAAAGGAGGAAUGUUGGGA
CAACCCUCGCCU-CUAGUCACUCUAGAGGGGAAGGUGGGGA
AGACCCGCACUU-CUAAUUACAUUAGAAGGGAAGGAACGGG
AGACCUCGACCUGCUAACCUUCUUAGAGGGGAAGUCCGAGG
AGACCUUAACCUACUAAACUUCUUAGAGGGGAAGUUUGAGG
```

Les alignements de séquences, tel que l'alignement précédent, doivent être lus colonne après colonne. Un alignement se doit de maximiser la "ressemblance" des lettres dans chaque colonne. Ici, nous avons encadré les colonnes contenant au moins quatre lettres identiques. La "ressemblance" des lettres est un concept variable en fonction du type de séquence considéré et du résultat souhaité. Ainsi, si l'on souhaite aligner des séquences d'acides aminés, la ressemblance est habituellement formellement définie à l'aide d'une matrice de score (matrices PAM (Dayhoff *et al.*, 1978)). Si, comme dans l'exemple précédent, on souhaite aligner des séquences d'acides nucléiques, la ressemblance est beaucoup plus souvent assimilée à une identité formelle des bases.

Afin de maximiser la ressemblance des lettres dans chaque colonne d'un alignement de séquences, les séquences sont parfois interrompues par un ou plusieurs "gaps", habituellement représentés par une séquence ininterrompue de caractères spéciaux '-'. Ainsi, on pourra comparer l'alignement précédent, qui contient trois gaps (un pour chacun des 3 premiers ARN), avec l'alignement suivant, obtenu en superposant simplement les cinq séquences :

```
CAACCCUUAUCCUUUGUAACUCAAAGGAGGAAUGUUGGGA
CAACCCUCGCCUCUAGUCACUCUAGAGGGGAAGGUGGGGA
AGACCCGCACUUCUAAUUACAUUAGAAGGGAAGGAACGGG
AGACCUCGACCUGCUAACCUUCUUAGAGGGGAAGUCCGAGG
AGACCUUAACCUACUAAACUUCUUAGAGGGGAAGUUUGAGG
```

Ce dernier alignement contient 17 colonnes bien conservées, ces 17 colonnes étant séparées en 7 groupes, au lieu de 23 colonnes en 6 groupes pour l'alignement précédent. Intuitivement, l'introduction de gaps a permis d'améliorer le "score de l'alignement" et c'est justement l'une des tâches des algorithmes d'alignement que de trouver un ensemble de gaps maximisant un "certain score d'alignement". Le score d'alignement est une version quantitative des intuitions biologiques censées expliquer les mutations d'une séquence à l'autre.

Un des buts essentiels de l'alignement de séquences est de mettre en évidence les points communs entre séquences différentes, provenant éventuellement d'organismes différents. Les zones bien conservées d'une séquence à l'autre sont alors des indices d'importance fonctionnelle (Sankoff & Kruskal, 1983). En effet, la pression évolutive favorise la conservation de sous-séquences essentielles, alors qu'elle autorise une plus grande liberté dans les "zones de remplissage" entre sites fonctionnels.

Nous nous limiterons ici à la présentation des concepts associés à l'alignement de deux séquences $a_1 \ldots a_m$ et $b_1 \ldots b_n$ car la généralisation directe des méthodes d'alignement de deux séquences à trois séquences ou plus (i.e. alignement multiple) entraîne une croissance exponentielle des complexités avec le nombre de séquences à aligner, même en utilisant des méthodes sophistiquées d'élagage de l'espace de recherche (Carillo & Lipman, 1988; Lipman *et al.*, 1989; Altschul & Lipman, 1989; Pevzner, 1992; Gupta *et al.*, 1995). Un exposé didactique de l'alignement multiple et de quelques méthodes est proposé dans (Fuellen, 1996). En pratique, l'alignement multiple de séquences est souvent réalisé soit à l'aide d'heuristiques utilisant les alignements de paires de séquences (Taylor, 1987; Subbiah & Harrison, 1989; Vingron & Pevzner, 1993; Huang, 1994b; Thompson *et al.*, 1994; Higgins *et al.*, 1992; Wang & Gusfield, 1996), soit après décomposition des séquences en blocs faciles à aligner (Vingron & Argos, 1989; Sobel & Martinez, 1986; Sagot, 1996), soit grâce à des modèles de Markov cachés (cf. section 3.8).

Pour obtenir de bons alignements, c'est-à-dire des alignements biologiquement significatifs, il faut exprimer quantitativement un certain nombre de critères qualitatifs :

1. Un alignement de deux lettres identiques (ou "match", tel $\left[\frac{A}{A}\right]$) est préférable à un alignement de deux lettres différentes (ou "mismatch", tel $\left[\frac{A}{G}\right]$). Le degré de différence autorisé entre lettres peut être pondéré à l'aide d'une matrice de similarité adaptée ;

2. Un alignement de deux lettres est préférable à un alignement d'une lettre avec un gap (délétion $\left[\frac{A}{-}\right]$ ou insertion $\left[\frac{-}{A}\right]$) ;

3. À nombre total de gaps identique dans une ligne d'un alignement, des gaps groupés sont préférables à des gaps distribués ;

4. Le coût marginal d'extension d'une suite ininterrompue de gaps est plus faible que le coût initial de création d'un gap (la fonction de coût des gaps est concave). L'optimisation d'un alignement pour ce quatrième critère réalise automatiquement une optimisation du troisième.

```
Align : [• •]          { $$.c = s($1[1].left[0].b, $1[2].left[0].b); }
      | [• _]          { $$.c = -délétion; }
      | [_ •]          { $$.c = -insertion; }
      | Align [• •]    { $$.c = $1.c + s($2[1].left[0].b, $2[2].left[0].b); }
      | Align [• _]    { $$.c = $1.c - délétion; }
      | Align [_ •]    { $$.c = $1.c - insertion; }
      ;
%constraint Align {} { if ($$.c > $!.c) $!.c = $$.c }
```

Figure 3.1: Le non-terminal Align possède un attribut c que la contrainte cherche à maximiser. Le score d'alignement de deux bases a =$1[1].left[0].b et b =$1[2].left[0].b est donné par la fonction $s(a, b)$, tandis que les coûts de délétion et d'insertion sont donnés par les constantes $\textit{délétion} > 0$ et $\textit{insertion} > 0$. Le token abrégé • représente l'ensemble des nucléotides s'il s'agit d'aligner deux ARN, ou l'ensemble des acides aminés s'il s'agit d'aligner deux protéines.

La MTSAG de la figure 3.1 est un premier exemple de grammaire utilisable pour optimiser l'alignement de deux séquences. Cette grammaire déclare en fait qu'un alignement est une suite de matches, mismatches, insertions ou délétions et que l'alignement optimal est celui dont l'arbre de dérivation possède un score maximal. La fonction s de similarité entre bases et les coûts d'insertion et délétion sont déterminés empiriquement en comparant les alignements obtenus avec les alignements attendus et en modifiant les valeurs jusqu'à ce que le résultat soit satisfaisant. En pratique, on prendra $s(a, a) > s(a, b \neq a)$ et $\textit{insertion} = \textit{délétion}$ (i.e. le score d'un match est supérieur au score d'un mismatch et les coûts d'insertion et de délétion sont égaux).

Ce modèle d'alignement, également appelé *linear gap penalty model*, ne considère pas qu'un gap est plus difficile à créer qu'à étendre alors que, biologiquement parlant, c'est précisément cette dernière hypothèse qui est la plus pertinente. Il faut donc transformer la fonction linéaire de coût des gaps de la grammaire précédente en une fonction plus raisonnable, qu'on choisit en pratique affine pour faciliter les calculs. Ainsi, si $k > 0$ est la taille d'un gap, on pourra utiliser la fonction $g(k) = -\alpha - \beta(k-1)$ avec $\alpha > \beta > 0$. La grammaire de la figure 3.2 permet d'optimiser l'alignement de deux séquences suivant ce modèle affine, également appelé *global alignment with affine gap penalty model*. Les paramètres s, α et β sont optimisés empiriquement, comme pour la grammaire de la figure 3.1.

Il existe une variante du modèle affine dont le but n'est plus d'optimiser l'alignement global de deux séquences $a_1 \dots a_m$ et $b_1 \dots b_n$, mais de trouver deux sous-séquences $a_i \dots a_j$ et $b_{i'} \dots b_{j'}$ telles que leur alignement optimal ait le plus grand score possible. Ce dernier modèle est appelé *local alignment with affine gap penalty model*, ou modèle de Smith-Waterman (Smith & Waterman, 1981). La mise en pratique du modèle d'alignement local est assez simple : il suffit d'attribuer un score nul à n'importe

```
Align    : Delete         { $$.c = $1; }
         | Insert         { $$.c = $1; }
         | [• •]          { $$.c = s($1[1].left[0].b, $1[2].left[0].b); } ;
         | Align [• •]    { $$.c = $1 + s($2[1].left[0].b, $2[2].left[0].b); } ;
Insert   : Align [_ •]    { $$.c = $1 - α; }
         | Insert [_ •]   { $$.c = $1 - β; }
         | [_ •]          { $$.c = -α; } ;
Delete   : Align [• _]    { $$.c = $1 - α; }
         | Delete [• _]   { $$.c = $1 - β; }
         | [• _]          { $$.c = -α; } ;
%default_constraint {} { if ($$.c > $!.c) $!.c = $$.c }
```

Figure 3.2: Le non-terminal Align représente n'importe quel alignement, alors que les non-terminaux Insert et Delete gèrent la création et l'extension des gaps pour la fonction de coût $g(k) = -\alpha - \beta(k-1)$. La contrainte de tous les non-terminaux est une contrainte de maximisation.

quel alignement de $a_1 \ldots a_{i-1}$ avec $b_1 \ldots b_{i'-1}$ et de $a_{j+1} \ldots a_m$ avec $b_{j'+1} \ldots b_n$:

$$\underbrace{\begin{matrix} a_1 \ldots a_{i-1} \\ b_1 \ldots b_{i'-1} \end{matrix}}_{\text{score nul}} \underbrace{\begin{matrix} a_i \ldots a_j \\ b_{i'} \ldots b_{j'} \end{matrix}}_{\text{score affine}} \underbrace{\begin{matrix} a_{j+1} \ldots a_m \\ b_{j'+1} \ldots b_n \end{matrix}}_{\text{score nul}}$$

Autrement dit, il existe une zone au début et à la fin d'un alignement où les scores d'alignement sont nuls. La grammaire de la figure 3.3 illustre une telle modification.

La formulation de tous les modèles présentés dans cette section repose habituellement sur des équations de programmation dynamique (les équations correspondant au modèle de Smith-Waterman sont données en figure 3.4). Pour ces modèles élémentaires, les équations de programmation dynamique sont, avec l'habitude, faciles à déduire de la grammaire équivalente. Précisons également qu'il existe une autre formulation équivalente à l'aide d'automates à deux bandes (Searls & Murphy, 1995). L'automate du modèle de Smith-Waterman est par exemple représenté en figure 3.5. À condition de disposer d'un éditeur graphique d'automates, la représentation sous forme d'automate à deux bandes est sans doute le moyen le plus clair de décrire les modèles d'alignement basés sur un schéma régulier d'alignement. Si l'automate doit être décrit à l'aide d'un fichier texte, alors la formulation grammaticale est sans doute tout aussi pratique.

Même si l'automate de la figure 3.5 est une représentation agréable de la grammaire linéaire à gauche de la figure 3.3, il est irréaliste d'espérer trouver une présentation graphique agréable pour toutes les MTSAG. En effet, l'automate équivalent à une MTSAG quelconque est un automate à pile non-déterministe m-bandes (cf. section 5.4 p. 218), et ce type d'automate ne possède pas de représentation graphique ou textuelle simple et directement compréhensible.

Toutes les grammaires précédentes (figures 3.1, 3.2 et 3.3) sont des grammaires linéaires à gauche. Cela signifie que les analyseurs syntaxiques générés par MTSAG2C à partir de ces grammaires ont tous des complexités en temps et en espace égales au

```
End    : Align            { $$.c = $1; }
       | End [• •]        { $$.c = $1; }
       | End [• _]        { $$.c = $1; }
       | End [_ •]        { $$.c = $1; } ;
Align  : Delete           { $$.c = $1; }
       | Insert           { $$.c = $1; }
       | Begin            { $$.c = 0; }
       | [• •]            { $$.c = s($1[1].left[0].b,$1[2].left[0].b); } ;
       | Align [• •]      { $$.c = $1 + s($2[1].left[0].b,$2[2].left[0].b); } ;
Insert : Align [_ •]      { $$.c = $1 - α; }
       | Insert [_ •]     { $$.c = $1 - β; }
       | [_ •]            { $$.c = -α; } ;
Delete: Align [• _]       { $$.c = $1 - α; }
       | Delete [• _]     { $$.c = $1 - β; }
       | [• _]            { $$.c = -α; } ;
Begin  : [• •]            { $$.c = 0; }
       | [• _]            { $$.c = 0; }
       | [_ •]            { $$.c = 0; }
       | Begin [• •]      { $$.c = 0; }
       | Begin [• _]      { $$.c = 0; }
       | Begin [_ •]      { $$.c = 0; } ;
%default_constraint {} { if ($$.c > $!.c) $!.c = $$.c }
```

Figure 3.3: Le non-terminal de départ de cette grammaire est `End`. La contrainte de tous les non-terminaux est une fois encore une contrainte de maximisation. Cela revient à dire que l'alignement cherché est celui dont l'arbre de dérivation a un score maximal.

$$e[k,l] = \max_{\substack{0 \leqslant i \leqslant k \\ 0 \leqslant j \leqslant k}} m[i,j]$$

$$m[k,l] = \max \begin{cases} m[k-1,l-1] + s(a_k, b_l) \\ i[k,l] \\ d[k,l] \end{cases}$$

$$d[k,l] = \max \begin{cases} m[k-1,l] - \alpha \\ d[k-1,l] - \beta \end{cases}$$

$$i[k,l] = \max \begin{cases} m[k,l-1] - \alpha \\ i[k,l-1] - \beta \end{cases}$$

Figure 3.4: Équations de programmation dynamique du modèle d'alignement local de Smith-Waterman. La valeur à calculer est $e[m,n]$. Au départ, $m[0,0] = 0$.

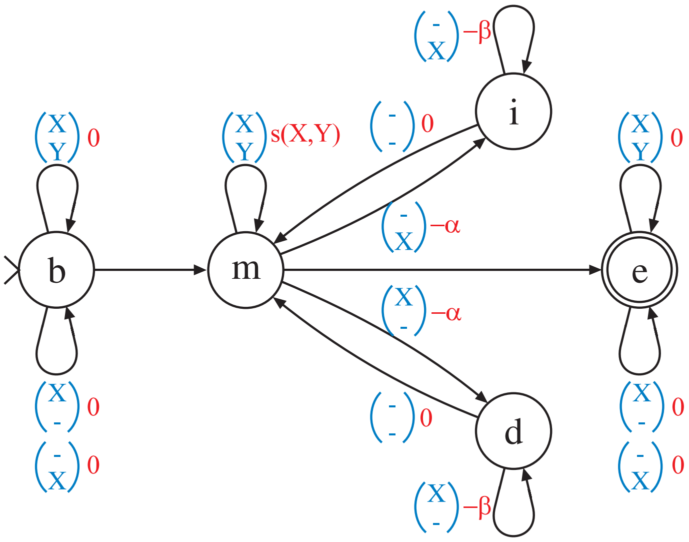


Figure 3.5: Automate à deux bandes équivalent à la grammaire de la figure 3.3 et aux équations de programmation dynamique de la figure 3.4. Cet automate possède cinq états. Les états b et e attribuent des scores nuls aux alignements de début et de fin de séquence respectivement. L'état m gère les scores des matches et des mismatches. Les états i et d gèrent les scores d'insertion et de délétion respectivement. Un alignement optimal de deux séquences est un chemin acceptant les deux séquences et ayant un score maximal.

produit des longueurs des bandes, c'est-à-dire $O(mn)$. L'implémentation directe des équations de programmation dynamique associées à une grammaire linéaire à gauche conduit aussi à des programmes dont la complexité en temps est $O(mn)$. En revanche, la complexité en espace peut passer à $O(m+n)$ au lieu de $O(mn)$, par exemple en adoptant un schéma de calcul du type *divide and conquer* (Hirschberg, 1975; Myers & Miller, 1988; Huang & Miller, 1991; Chao *et al.*, 1993; Chao, 1994; Huang, 1994a; Chao *et al.*, 1995; Thompson, 1995).

Nous avons comparé les performances d'un analyseur syntaxique automatiquement généré par MTSAG2C à partir d'une grammaire linéaire à gauche, et les performances d'une implémentation manuelle des équations de programmation dynamique associées à cette grammaire linéaire à gauche. La grammaire utilisée pour les tests est donné en figure 3.6. Les résultats obtenus sont regroupés dans la table 3.1 (calculs effectués sur une station DEC Alpha 250 4/266). Cette table montre que l'analyseur syntaxique et l'implémentation manuelle des équations ont tous deux des complexités quadratiques en temps et en espace. Par contre, l'analyseur est approximativement 35 fois plus lent que l'implémentation manuelle des équations et occupe environ 5 fois plus de mémoire. En fait, il est exagéré d'utiliser un analyseur syntaxique général avec une grammaire aussi simple que la grammaire de la figure 3.6, où toutes les projections sont linéaires à gauche et non-ambiguës. En effet, l'analyseur généré par MTSAG2C à partir de la grammaire prend en charge un certain nombre de structures de données prévues pour

```
%nb_tapes 2                %empty _    %token C G T A
%attribute "int" v         %parentheses . C G T A
%terminal_attribs{}        %default_nonterminal_attribs <v>
%default_constraint        {}{ if ($$.v < $!.v) $!.v = $$.v; }
%%
frame0 : frame0 [• •=1]  { $$.v = $1.v; }
       | frame0 [• •!1]  { $$.v = $1.v + 2; }
       | frame1 [_ •]    { $$.v = $1.v + 1; }
       | frame2 [• _]    { $$.v = $1.v + 1; }
       | [_ _]           { $$.v = 0; };
frame1 : frame0 [• _]    { $$.v = $1.v + 3; }
       | frame1 [• •=1]  { $$.v = $1.v + 1; }
       | frame1 [• •!1]  { $$.v = $1.v + 3; }
       | frame2 [_ •]    { $$.v = $1.v + 3; };
frame2 : frame0 [_ •]    { $$.v = $1.v + 3; }
       | frame1 [• _]    { $$.v = $1.v + 3; }
       | frame2 [• •=1]  { $$.v = $1.v + 1; }
       | frame2 [• •!1]  { $$.v = $1.v + 3; };
```

Figure 3.6: Cette MTSAG à deux bandes définit un schéma d'alignement où les gaps modifiant la phase de lecture recoivent une pénalité plus grande que les gaps conservant la phase de lecture. Cette grammaire est une réécriture d'un automate à deux bandes donné dans (Searls & Murphy, 1995).

	MTSAG	équations prog. dyn.
2 bandes de 125 bases		
temps en secondes :	0,36	0,01
espace en Moctets :	0,9	0,2
2 bandes de 250 bases		
temps en secondes :	1,5	0,04
espace en Moctets :	3,4	0,7
2 bandes de 500 bases		
temps en secondes :	5,9	0,17
espace en Moctets :	13,5	2,9

Table 3.1: Comparaison entre un analyseur syntaxique généré automatiquement par MTSAG2C à partir de la MTSAG de la figure 3.6 et un programme obtenu manuellement à partir des équations de programmation dynamique déduites de cette même grammaire.

gérer les ambiguïtés des MTSAG quelconques, alors que l'analyse syntaxique n'est ambiguë sur aucune bande. De plus, l'analyseur syntaxique n'évite aucun des calculs normalement effectués par l'implémentation manuelle des équations de programmation dynamique (la situation devient plus favorable lorsque le modèle se complique, comme pour le modèle de Sankoff p. 95).

Cependant, il faut reconnaître qu'il est beaucoup plus simple d'écrire une grammaire rendant compte d'une méthode d'alignement que les équations de programmation dynamique associées, surtout lorsque le modèle d'alignement devient plus complexe (Peltola *et al.*, 1986). Les MTSAG restent donc utilisables pour prototyper de nouvelles méthodes d'alignement, mais pas pour produire un code performant. Précisons que cette remarque n'est valable que pour les MTSAG dont les projections sont linéaires, comme les MTSAG linéaires à gauche données dans cette section. Nous verrons plus loin que des modèles plus complexes sont mieux gérées par des MTSAG et par MTSAG2C que par un programmeur se contentant de transcrire les équations de programmation dynamique associées au modèle. Dans une certaine mesure, il existe un seuil de complexité du modèle au delà duquel les équations de programmation dynamique deviennent trop nombreuses et compliquées pour être programmées, ou même écrites. Ce seuil de complexité est comparable au seuil existant dans le cas des grammaires LR(1) : il ne viendrait à l'esprit de personne de construire manuellement un analyseur syntaxique pour une grammaire LR(1) telle que celle du langage C (Lee, 1985), ou même de construire manuellement les tables d'analyse d'un analyseur LR. Ce type de tâche automatisable est bien plus efficacement réalisé par un programme spécialement conçu comme YACC, mais il va de soi qu'un programmeur compétent, et disposant de beaucoup de temps devant lui, sera à même de produire un analyseur plus performant que celui généré par YACC.

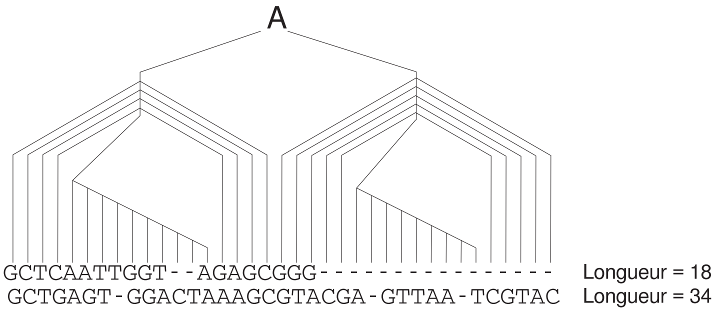
3.2 Contraintes d'alignement

Les contraintes d'alignement sont des exemples de contraintes de sélection (cf. définition 2.14 p. 79 et section A.4.2.10 p. 245) sans doute utilisables avec toutes les MTSAG traitant des problèmes d'alignement. En quelques mots, les contraintes d'alignement, en tant que contraintes de sélection, permettent d'élaguer la forêt d'arbres de dérivation parcourue par l'analyseur syntaxique afin de l'accélérer et d'économiser la mémoire. Comme pour toute contrainte de sélection, il faut se garder de trop élaguer la forêt, sous peine de supprimer des sous-arbres valides.

Les contraintes d'alignement sont des conséquences de l'observation suivante : l'alignement optimal de plusieurs séquences semblables ne possède pas de grand gap dans les régions “bien conservées” (i.e. dans les régions relativement similaires d'une séquence à l'autre, la similitude étant jugée à l'aide de critère variables en fonction du problème) car la pénalité associée à un tel gap est suffisante pour rejeter loin de l'alignement optimal tout alignement le contenant. Il est donc inutile de chercher à aligner plusieurs séquences semblables suivant des dérivations contenant des gaps longs. Il va de soi qu'il est bien difficile d'exprimer quantitativement une telle remarque qualitative. Seule l'expérience permet de dégager des règles empiriques. En tout état de cause, les contraintes d'alignement n'étant pas ici introduites dynamiquement, l'œil de l'expert reste nécessaire pour estimer au préalable le degré de similitude des séquences et donc le seuil de longueur maximum de gap.

Pour simplifier, nous considérons dans la suite de la discussion que les MTSAG ont deux bandes. Les résultats obtenus sont facilement généralisables à un nombre supérieur de bandes.

Une dérivation d'un alignement contenant un grand gap sur une des bandes se traduit par l'existence d'un non-terminal A de la dérivation tel que la différence de longueur entre la première bande et la deuxième bande est importante (nous retrouvons derrière ce dernier mot la notion "d'œil de l'expert").



Si l'on souhaite limiter cette différence de longueur, il suffit d'introduire une contrainte de sélection rejetant toute dérivation dont la différence de longueur entre les deux bandes dépasse une certaine valeur :

```
%constraint A { if (labs ($$[1].len - $$[2].len) > valeur) return 0;  }
              { ...contrainte d'optimisation usuelle... }
```

Par exemple, nous pouvons remplacer la contrainte par défaut de la figure 3.6 par la contrainte suivante, pour limiter la taille totale des gaps à 20 :

```
%default_constraint { if (labs ($$[1].len - $$[2].len) > 20) return 0;  }
                    { if ($$.v < $!.v) $!.v = $$.v;  }
```

L'utilisation de la contrainte précédente avec les mêmes exemples que dans la table 3.1 améliore considérablement les performances d'analyse, comme le montre la table 3.2. La lecture de cette table semble suggérer une complexité sub-quadratique en temps dans cet exemple particulier lorsque la contrainte est utilisée. En fait, l'algorithme explore encore un nombre quadratique de possibilités, comme l'atteste l'évolution quadratique de la complexité en espace, mais certaines de ces possibilités sont immédiatement éliminées par la contrainte alors que d'autres nécessitent un nombre plus élevé d'opérations. La constante multiplicative du terme quadratique de la complexité en temps est donc beaucoup trop petite pour que le terme quadratique soit prédominant pour d'aussi petites longueurs de séquences. L'utilisation de séquences plus longues permettrait de mettre de nouveau en évidence la complexité quadratique. Notons cependant que l'introduction de contraintes d'alignements (ou d'autres

	MTSAG sans contrainte	MTSAG avec contrainte	programmation directe des équations
2 bandes de 125 bases			
temps en secondes :	0,36	0,15	0,01
espace en Moctets :	0,9	0,7	0,2
2 bandes de 250 bases			
temps en secondes :	1,5	0,4	0,04
espace en Moctets :	3,4	2,5	0,7
2 bandes de 500 bases			
temps en secondes :	5,9	1,2	0,17
espace en Moctets :	13,5	9,2	2,9

Table 3.2: Les colonnes de gauche et de droite de la table sont des copies des colonnes de la table 3.1. La colonne centrale correspond aux performances de l'analyseur syntaxique utilisant une contrainte d'alignement. Il y a peu d'espace mémoire gagné grâce à l'introduction d'une contrainte. En fait, la majeure partie des allocations mémoire effectuées par l'analyseur sont destinées à des structures dont la taille dépend du produit des longueurs des chaînes d'entrée.

contraintes de sélection) peut contribuer à diminuer les complexités en temps et en espace dans d'autres cas (cf. section 3.11).

Dans l'hypothèse où l'on souhaite réellement autoriser quelques grands gaps dans un alignement, il suffit d'introduire dans le modèle d'alignement des non-terminaux spécialement destinés à gérer ces grands gaps, et de ne pas associer de contrainte d'alignement à ces non-terminaux. De cette manière, on autorise effectivement la présence de grands gaps, mais ces gaps sont explicitement référencés comme tels dans la grammaire et ils ne sont plus des conséquences accidentelles du modèle d'alignement et des séquences à aligner.

Il est également possible de faire varier la taille des gaps autorisés en fonction d'attributs spéciaux stockés avec les tokens des bandes. Dans l'exemple ci-dessous, chaque token de la première bande possède un attribut physique spécial appelé `gap`. L'attribut physique `gap` d'un token représente la taille maximale d'un gap dans un alignement dérivable de A commençant par ce token sur la première bande :

```
%constraint A
     { if (labs ($$[1].len - $$[2].len) > $$[1].left[0].gap) return 0;  }
     { ...contrainte d'optimisation usuelle... }
```

3.3 Alignement des ARN – Modèle de Sankoff

Dans cette section, nous allons montrer que le modèle de Sankoff, utilisé pour aligner et replier simultanément les ARN, est bien mieux traité par les MTSAG que par les équations de programmation dynamique originales proposées par Sankoff.

3.3.1 Nécessité d'un alignement plus sophistiqué pour les ARN

Les différents modèles d'alignement de séquences biologiques traités dans la section 3.1 font l'hypothèse que le score d'alignement dans une colonne est indépendant des autres colonnes (linear gap penalty), ou dépend de façon markovienne de la colonne précédente (affine gap penalty, frame gap penalty). Ces modèles ont été appliqués avec succès à l'alignement de séquences de protéines ou à l'alignement de séquences d'ADN codant pour des protéines, à condition que les séquences ne soient pas trop différentes.

L'hypothèse de dépendance markovienne des colonnes d'un alignement est malheureusement inapplicable lorsqu'il s'agit d'aligner des ARN catalytiques ou des séquences d'ADN codant pour des ARN catalytiques. En effet, deux remarques fondamentales réduisent à néant tout espoir de pouvoir appliquer un modèle markovien d'alignement à ces ARN :

- La structure tertiaire des ARN catalytiques est une composante fondamentale de leur fonction enzymatique. La pression de sélection tend donc à conserver les éléments structuraux importants, mais pas nécessairement la composition en bases de ces éléments structuraux. La maximisation d'identité dans les colonnes n'est donc pas un critère pertinent[1].

- Deux bases appariées dans un élément structural important d'une structure secondaire d'ARN tendent à muter corrélativement. Ainsi, si les deux bases appariées sont `A` et `U` et si la première base mute en `C`, alors la deuxième base mutera préférentiellement en `G` pour maintenir l'appariement (**mutation compensatoire**). Ces dépendances à "longue-distance" dans les alignements d'ARN ne sont pas modélisables par un modèle markovien.

Pour illustrer les deux remarques précédentes, considérons l'alignement correct de deux séquences d'ARN de transfert ayant le même anticodon `TGC` (Sprinzi *et al.*, 1996). La première séquence provient du génome de la drosophile et la deuxième du génome du phage T5. La structure secondaire associée à chaque séquence est représentée immédiatement en dessous, sous la forme d'une suite de parenthèses correctement imbriquées.

```
AGGGUUGUAGUUAAUU----AUAACAUUUGAUUUGCAUUCAAAAAGUAUUG-AAU-AUU-CAAUCUACCUUA
((((( (  ((((        )))) (((((       )))))    ((((         ))))) )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        ))))((((((       ))))))   (((((       ))))))))))))
```

[1] Les protéines forment elles aussi des structures tertiaires et cette remarque est donc, a priori, applicable aux protéines. En pratique, la structure a effectivement une influence sur les corrélations entre colonnes d'un alignement, mais cette influence est moins marquée que celle de la fonction de chaque acide aminé. Par exemple, si le caractère acide d'un acide aminé est indispensable à la fonction, alors les mutations conserveront ce caractère acide, etc. Dans certains cas, l'influence de la structure a pu être modélisée avec succès à l'aide d'adaptations de procédures d'apprentissage des HMM (Eddy, 1995) (voir aussi la section 3.8). La modélisation efficace des dépendances structurales pour les protéines "problématiques", bien qu'étant encore un problème sans solution entièrement satisfaisante, est moins fondamentale que pour les ARN.

Comme nous pouvons le constater, ces deux ARN de transfert, aux fonctions pourtant identiques, ont des séquences très différentes. Un algorithme utilisant une technique d'alignement local ne tenant pas compte de la structure secondaire sous-jacente (Pearson & Lipman, 1988) fournit l'alignement suivant des deux mêmes séquences :

```
  AGGGUUGUAGUUAAUUAUAACAUU--UGAUUUGCAUUCAAAAAGUAUUGAAU--AUUCAAUCUA---CCUUA
  ((((( (  ((((    )))) ((  (((       )))))    ((((        ))))) )   ))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        ))))((((((       ))))))   (((((       ))))))))))))
```

Dans ce second alignement, 29 colonnes ont des acides nucléiques identiques, contre 27 colonnes pour le premier alignement, et il y a seulement 3 gaps, contre 4 pour le premier. Ce deuxième alignement est donc préférable au premier si l'on se réfère uniquement aux critères d'alignement donnés en section 3.1. Par contre, l'examen de l'alignement des structures secondaires sous-jacentes montre que le deuxième alignement comporte en fait de nombreuses erreurs. En effet, il est essentiel que les bases appariées appartenant à des éléments structuraux identiques soient alignées, ce qui n'est visiblement pas le cas pour la majorité des paires du deuxième alignement.

En pratique, pour aligner correctement une famille d'ARN, il faut tenir compte de la structure consensus de ces ARN. La prise en compte de la structure peut se faire à plusieurs niveaux :

- Il est possible de chercher dans un alignement d'une famille d'ARN les paires de colonnes dont les bases sous-jacentes sont le plus souvent appariées. Les suites imbriquées ininterrompues de colonnes appariées les plus significatives (i.e. les plus longues) signalent alors des hélices communes dans les structures des ARN alignés. En fixant l'alignement des paires de colonnes trouvées et en affinant l'alignement des parties restantes, on peut espérer faire converger le processus vers un alignement correct.

  L'exemple précédent semble indiquer une certaine inefficacité de cette méthode. C'est pourtant celle qui a été utilisée pour trouver la structure consensus des ARN de transfert, puis celle des ARN ribosomaux (Fox & Woese, 1973; Noller *et al.*, 1981) (voir également (Woese & Pace, 1993)). En réalité, pour que la méthode fonctionne, il faut disposer de nombreux ARN de la même famille. De plus, ces ARN ne doivent pas tous être aussi différents deux à deux que les ARN de transfert de la drosophile et du phage T5 ayant servi à illustrer notre propos. Les paires d'ARN suffisamment semblables peuvent alors être alignées par un algorithme ne tenant pas compte de leur structure, sans risque d'obtenir de trop nombreuses erreurs d'alignements. L'examen des alignements obtenus permet de mettre en évidence des mutations compensatoires, qui sont un signe d'existence d'appariement. En procédant ainsi de proche en proche, la structure consensus finit par être révélée.

- À l'inverse, il est possible de commencer par replier les ARN. Les structures obtenues sont ensuite alignées (Shapiro & Zhang, 1990; Chevalet & Michot, 1992). Cette procédure a été imaginée à une époque où l'algorithme de repliement de

Zuker (cf. section 1.7.2 p. 49) commençait à fournir des structures fiables pour les ARN de transfert et où l'on pouvait espérer démontrer sa fiabilité dans de nombreux autres cas. Il était donc naturel d'envisager un alignement des structures plutôt qu'un alignement direct des séquences (cf. section 3.7 p. 127 pour des exemples de modèles utilisés).

La structure secondaire d'un ARN étant facilement représentable par un arbre ordonné[2], la recherche d'un alignement optimal de deux structures devient alors un problème de recherche d'une relation entre deux arbres (Shapiro & Zhang, 1990) ou d'alignement de deux arbres (Jiang *et al.*, 1994). La difficulté fondamentale soulevée par cette stratégie vient de l'inexactitude des structures initiales. Il est maintenant admis que le repliement a priori des ARN à l'aide d'un algorithme tel que celui de Zuker peut générer sans préavis des structures globalement et localement fausses (Han & Kim, 1993). Toute tentative d'optimisation des procédures d'alignement de structures est sans intérêt si la qualité des structures de départ, et donc la qualité des résultats obtenus, est compromise. Les modifications de l'algorithme de Zuker visant à lui faire générer des structures sous-optimales (Williams & Tinoco, 1986; Zuker, 1989b) dont l'analyse doit permettre de trouver les éléments structuraux les plus significatifs (Le *et al.*, 1989a,b) améliorent à peine la situation. L'utilisation directe des ensembles de structures sous-optimales est encore plus aléatoire car elle engendre une explosion combinatoire difficile à maîtriser, surtout lorsqu'il s'agit d'aligner les ensembles de structures secondaires de deux ARN, voire trois ou plus. De plus, aucune garantie formelle quant à la qualité des structures fournies n'est donnée.

- Une dernière approche, que nous allons développer plus avant dans cette section et dans les deux sections suivantes consiste à aligner et replier simultanément les ARN. Cette approche ne présente pas les inconvénients des approches précédentes mais nous verrons qu'elle entraîne un accroissement de complexité important en temps et en espace.

3.3.2 Modèle de Sankoff

Historiquement, le premier auteur ayant exposé une méthode de repliement et d'alignement simultanés pour les ARN fut Sankoff (1985). La remarque fondamentale faite par Sankoff est la suivante : si la programmation dynamique est à la base de l'algorithme d'alignement global avec pénalité linéaire de gap et de l'algorithme standard de repliement des ARN selon le modèle thermodynamique, alors pourquoi ne pas envisager un algorithme de programmation dynamique modifié pour réaliser les deux opérations simultanément ?

Nous noterons ici $a_1 \ldots a_m$ la séquence primaire d'un premier ARN A, et $b_1 \ldots b_n$ la séquence primaire d'un second ARN B. Afin de faciliter[3] la fusion des deux algorithmes

[2] Un arbre est ordonné lorsque l'ordre des fils de chaque sommet est déterminé.

[3] Nous reviendrons sur cette contrainte de facilité plus loin.

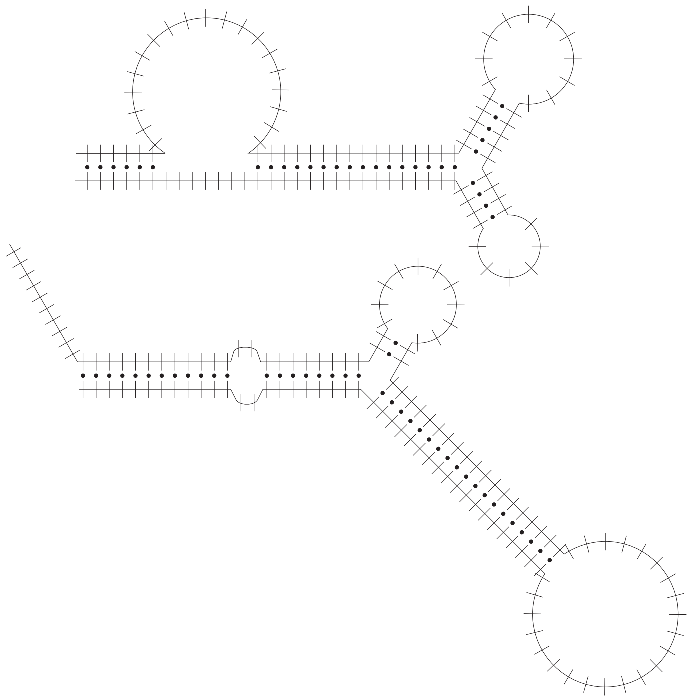


Figure 3.7: Les deux structures de cette figure sont équivalentes. Les bases individuelles sont représentées par des traits sur les lignes et les appariements par des points entre les traits. En pratique, les contraintes d'alignement attribueront un score très défavorable à l'alignement de ces deux structures ce qui conduira l'algorithme à chercher d'autres structures équivalentes.

de programmation dynamique, Sankoff impose un certain nombre de contraintes à l'alignement et aux deux structures S_A et S_B produits par son algorithme.

1. Les deux structures ont la même configuration de branchement, déterminée par les paires externes et les boucles multiples (cf. section 1.7.2.1 p. 51 pour une définition de ces termes). Plus précisément, soient $i_1 < \cdots < i_v$ les positions des bases de A impliquées dans une paire accessible d'une boucle extérieure ou d'une boucle multiple de A. Soient $j_1 < \cdots < j_w$ les positions équivalentes dans B. Alors l'algorithme de Sankoff produit des séquences telles que $v = w$ et $(i_f \cdot i_g) \in S_A \Leftrightarrow (j_f \cdot j_g) \in S_B$. La figure 3.7 est une illustration de cette contrainte.

2. Les bases des paires accessibles de la boucle externe et des boucles multiples sont alignées. Avec les définitions précédentes, cela signifie que les bases en positions i_p et j_p $(1 \leqslant p \leqslant v)$ sont alignées.

3. Les bases des paires fermantes des épingles à cheveux sont alignées.

4. Les boucles d'index 2 (empilements, boucles proéminentes et boucles multiples) sont soit alignées, soit insérées en totalité, soit effacées en totalité d'une structure à l'autre.

3.3.3 Équations de programmation dynamique proposées par Sankoff

La seule flexibilité autorisée par l'algorithme de Sankoff d'une structure à l'autre se situe au niveau des boucles d'index 2 et des bases non-appariées. Les coûts d'alignements sont pris en compte au niveau de ces zones flexibles pour les empêcher d'être de tailles trop différentes (cas de la figure 3.7). Le coût d'alignement avec pénalité linéaire de gap entre une portion $a_{i_1} \ldots a_{j_1}$ de A et une portion $b_{i_2} \ldots b_{j_2}$ de B est donné par $D(i_1, j_1; i_2, j_2)$.

Proposition 3.1 (Sankoff (1985))
Soit $x > 0$ le coût d'alignement de deux bases différentes et $y > 0$ le coût d'alignement d'une base avec un gap. Le coût d'alignement entre $a_{i_1} \ldots a_{j_1}$ et $b_{i_2} \ldots b_{j_2}$ est donné par :

$$D(i_1, j_1; i_2, j_2) = \begin{cases} D(i_1, j_1; i_2, j_2 - 1) + y & \\ D(i_1, j_1 - 1; i_2, j_2) + y & \\ D(i_1, j_1 - 1; i_2, j_2 - 1) + x & si\ a_{j_1} \neq b_{j_2} \\ D(i_1, j_1 - 1; i_2, j_2 - 1) & si\ a_{j_1} = b_{j_2} \end{cases}$$

Le coût d'alignement est donc calculé par récurrence avec pour conditions initiales :

$$D(i_1, i_1; i_2, i_2) = \begin{cases} x & si\ a_{i_1} \neq b_{i_2} \\ 0 & si\ a_{i_1} = b_{i_2} \end{cases}$$

Si $i_1 > j_1$ et $i_2 < j_2$, alors $D(i_1, j_1; i_2, j_2) = (j_2 - i_2 + 1)y$ et ce coût correspond à l'insertion complète de $b_{i_2} \ldots b_{j_2}$. Symétriquement, si $i_1 < j_1$ et $i_2 > j_2$, alors $D(i_1, j_1; i_2, j_2) = (j_1 - i_1 + 1)y$ et ce coût correspond à la délétion complète de $a_{i_1} \ldots a_{j_1}$.

Étant données cette fonction D et les contraintes d'alignement et d'équivalence des structures, les propositions 1.4, 1.5, 1.6 p. 56 données pour le modèle thermodynamique se généralisent aisément à deux ARN. Il suffit de doubler les indices et de tenir compte du coût d'alignement aux endroits où une flexibilité d'alignement est autorisée.

Définition 3.1 (Sankoff (1985))
Soit $W(i_1, j_1; i_2, j_2)$ le coût minimum possible d'alignement et de repliement simultanés pour deux structures secondaires équivalentes sur $a_{i_1} \ldots a_{j_1}$ et $b_{i_2} \ldots b_{j_2}$, contenant **éventuellement** *les paires $(i_1 \cdot j_1)$ et $(i_2 \cdot j_2)$, et dont l'énergie de boucle extérieure est nulle.*

Soit $V(i_1, j_1; i_2, j_2)$ le coût minimum possible d'alignement et de repliement simultanés pour deux structures secondaires équivalentes sur $a_{i_1} \ldots a_{j_1}$ et $b_{i_2} \ldots b_{j_2}$ contenant **obligatoirement** *les paires $(i_1 \cdot j_1)$ et $(i_2 \cdot j_2)$.*

Soit $W'(i_1, j_1; i_2, j_2)$ le coût minimum possible d'alignement et de repliement simultanés pour deux structures secondaires équivalentes sur $a_{i_1} \ldots a_{j_1}$ et $b_{i_2} \ldots b_{j_2}$, contenant **éventuellement** *les paires $(i_1 \cdot j_1)$ et $(i_2 \cdot j_2)$, et dont l'énergie de boucle extérieure est calculée à l'aide d'une relation affine :*

$$a \times (\textit{taille de la boucle}) + \sum_{\textit{paire} \in \textit{boucle}} e_{interne}(\textit{paire})$$

Proposition 3.2 (Sankoff (1985))
La fonction W se calcule grâce à la récurrence suivante :

$$W(i_1, j_1; i_2, j_2) = \min \begin{cases} V(i_1, j_1; i_2, j_2) \\ \min\limits_{\substack{i_1 \leqslant k_1 < j_1 \\ i_2 \leqslant k_2 < j_2}} \big(W(i_1, k_1; i_2, k_2) + W(k_1+1, j_1; k_2+1, j_2)\big) \\ D(i_1, j_1; i_2, j_2) \end{cases}$$

Proposition 3.3 (Sankoff (1985))
La fonction V se calcule grâce à la récurrence suivante :

$$V(i_1, j_1; i_2, j_2) = \min \begin{cases} e(L_{i_1,j_1}) + e(L_{i_2,j_2}) + D(i_1, j_1; i_2, j_2) \\ \qquad \text{si } (i_1 \cdot j_1) \text{ et } (i_2 \cdot j_2) \text{ sont les paires fermantes d'épingles à cheveux.} \\ e(L_{i_1,j_1,i'_1,j'_1}) + e(L_{i_2,j_2,i'_2,j'_2}) + V(i'_1, j'_1; i'_2, j'_2) \\ \qquad + D(i_1, i'_1 - 1; i_2, i'_2 - 1) + D(j'_1 + 1, j_1; j'_2 + 1, j_2) \\ \qquad \text{où, soit } L_{i_1,j_1,i'_1,j'_1} \text{ et } L_{i_2,j_2,i'_2,j'_2} \text{ sont des boucles d'index 2 (cas alignement), soit } L_{i_1,j_1,i'_1,j'_1} \text{ est une boucle d'index 2 et } (i'_2 \cdot j'_2) = (i_2 \cdot j_2) \text{ (cas délétion), soit } L_{i_2,j_2,i'_2,j'_2} \\ \qquad \text{est une boucle d'index 2 et } (i'_1 \cdot j'_1) = (i_1 \cdot j_1) \text{ (cas insertion). On supposera de plus que la taille des boucles est bornée.} \\ e_{fermante}(i_1 \cdot j_1) + e_{fermante}(i_2 \cdot j_2) \\ \quad + \min\limits_{\substack{i_1+1 \leqslant k_1 < j_1-1 \\ i_2+1 \leqslant k_2 < j_2-1}} \big(W'(i_1+1, k_1; i_2+1, k_2) \\ \qquad\qquad + W'(k_1+1, j_1-1; k_2+1, j_2-1)\big) \end{cases}$$

avec $V(i_1, i_1; i_2, i_2) = \infty$.

```
%token A C G U { }

%abbreviation ( A U G C G U ) U A C G U G
%abbreviation • A C G U

%default_constraint
{}
{ if ($$.E < $!.E) $!.E = $$.E; }

%%

Structure     : Structure Loop
              | Loop
              | Structure •
              | •

Loop          : ( SingleStrand )
              | ( Index2 )
              | ( LoopSeq )

Index2        : Loop
              | Index2 •
              | • Index2
              | • Index2 •

LoopSeq       : LoopSeq Loop
              | Loop
              | LoopSeq •
              | •

SingleStrand  : •
              | SingleStrand •
```

Figure 3.8: Squelette de la grammaire S-attribuée décrite dans la figure 1.11 p. 61. Les attributs et fonctions d'évaluation d'attributs ont été supprimés de cette grammaire.

Proposition 3.4 (Sankoff (1985))
La fonction W' se calcule grâce à la récurrence suivante :

$$W'(i_1,j_1;i_2,j_2) = \min \begin{cases} V(i_1,j_1;i_2,j_2) + e_{interne}(i_1\cdot j_1) + e_{interne}(i_2\cdot j_2) \\ \min\limits_{\substack{i_1\leqslant k_1<j_1\\ i_2\leqslant k_2<j_2}} \begin{cases} W'(i_1,k_1;i_2,k_2) + \\ \qquad (j_1-k_1+j_2-k_2)a + D(k_1+1,j_1;k_2+1,j_2) \\ W'(i_1,k_1;i_2,k_2) + W'(k_1+1,j_1;k_2+1,j_2) \\ W'(k_1+1,j_1;k_2+1,j_2) + \\ \qquad (k_1-i_1+1+k_2-i_2+1)a + D(i_1,k_1;i_2,k_2) \end{cases} \end{cases}$$

avec $W'(i_1,j_1;i_2,i_2) = W'(i_1,i_1;i_2,j_2) = \infty$.

La valeur finalement cherchée est $W(1,m;1,n)$, c'est-à-dire le coût minimal d'alignement de deux structures équivalentes pour les ARN A et B.

Une analyse élémentaire des équations précédentes montre que, à condition de transformer les fonctions V, W et W' en tables indexées par i_1, j_1, i_2, j_2, le temps de calcul de $W(1,m;1,n)$ est $O(m^3n^3)$ et l'espace nécessaire est $O(m^2n^2)$.

3.3.4 MTSAG du modèle de Sankoff

Après avoir exposé le modèle de Sankoff et ses contraintes, puis les équations de programmation dynamique proposées par Sankoff, nous allons montrer comment formuler ce modèle à l'aide d'une MTSAG. La figure 3.8 est un rappel des éléments essentiels de la grammaire utilisée pour replier les ARN selon le modèle thermodynamique.

Nous allons adopter pour cette grammaire la même démarche que Sankoff : nous allons la généraliser en tenant compte du coût d'alignement et des contraintes d'équivalence.

Une des premières conséquences de la généralisation de la grammaire à l'alignement de deux ARN est la déclaration d'un nombre de bandes égal à 2:

```
%nb_tapes 2
```

Ensuite, nous allons généraliser le token abrégé • en autorisant l'apparition de gaps. Il faut pour cela déclarer préalablement la syntaxe des gaps :

```
%empty _
%abbreviation • _ A C G U
```

La grammaire ayant deux bandes, ses terminaux seront écrits sous la forme `[• •]`, ou sous une forme équivalente. Le coût d'alignement étant linéaire, il est possible de calculer dès la construction des terminaux le coût associé à l'alignement, en le stockant dans un attribut physique `cost` de chaque terminal.

```
%terminal_attribs
     {
         if ($$[1].len == 0)
            $$.cost = DELETE;
         else if ($$[2].len == 0)
            $$.cost = INSERT;
         else if ($$[1].left[0].index != $$[2].left[0].index)
            $$.cost = MISMATCH;
         else
            $$.cost = 0;
     }
```

Les attributs physiques LS, RS de `Index2` doivent être transformés en LS1, RS1 pour la première bande et LS2, RS2 pour la deuxième bande :

```
%attribute "unsigned short" LS1 LS2 RS1 RS2
```

Une fois ces déclarations préliminaires effectuées, nous pouvons généraliser les règles de grammaire en doublant chaque occurrence d'un terminal abrégé '•', '(', ou ')'. Le doublement des parenthèses ouvrantes et fermantes correspond à la contrainte d'alignement des paires accessibles des boucles extérieures et multiples. Les fonctions d'évaluation d'attributs sont généralisées en tenant compte de l'énergie de chaque bande et du coût d'alignement stocké dans l'attribut `cost` des terminaux. Les premières règles de cette généralisation ont la forme suivante :

```
Structure : Structure Loop    { $$.E = $1.E + $2.E;  }
          | Structure [• •]   { $$.E = $1.E + $2.cost;  }
          | Loop              { $$.E = $1.E;  }
          | [• •]             { $$.E = $1.cost;  };

Loop      : [( (] SingleStrand [) )]
             {
                int bonus = 0;
                if ($2[1].len == 4) bonus += BonusTetraloop (1, $2[1].left);
```

```
%nb_tapes 2
%empty _
%token A C G U { }
%abbreviation ( A U G C G U ) U A C G U G
%abbreviation • _ A C G U

%terminal_attribs
  {
    if ($$[1].len == 0)
      $$.cost = DELETE;
    else if ($$[2].len == 0)
      $$.cost = INSERT;
    else if ($$[1].left[0].index != $$[2].left[0].index)
      $$.cost = MISMATCH;
    else
      $$.cost = 0;
  }

%default_constraint
{}
{ if ($$.E < $!.E) $!.E = $$.E; }

%%
```

```
Structure    : Structure Loop
             | Loop
             | Structure [• •]
             | [• •]
Loop         : [( (] SingleStrand [) )]
             | [( (] Index2 [) )]
             | [( (] LoopSeq [) )]

Index2       : Loop
             : [_ (] Index2 [_ )]
             | [( _] Index2 [) _]
             | [• •] Index2
             | Index2 [• •]
             | [• •] Index2 [• •]

LoopSeq      : LoopSeq Loop
             | Loop
             | LoopSeq [• •]
             | [• •]

SingleStrand : [• •]
             | SingleStrand [• •]
```

Figure 3.9: Squelette de la MTSAG adaptée au modèle de Sankoff.

```
          if ($2[2].len == 4) bonus += BonusTetraloop (2, $2[2].left);
          $$.E = $1.cost + $2.E + $3.cost + bonus +
            Hairpin($2[1].len) + Mismatch(1,$$[1].left,$$[1].right) +
            Hairpin($2[2].len) + Mismatch(2,$$[2].left,$$[2].right);
        }
  | [( (] Index2 [) )]  { $$.E = $1.cost + $2.E + $3.cost;  }
  | [( (] LoopSeq [) )] { $$.E = 2 * 460 + $1.cost + $2.E + $3.cost;  };
```

La grammaire obtenue n'autorise pas encore les insertions ou les délétions de boucles d'index 2. Il faut donc ajouter les règles nécessaires.

```
Index2 : [_ (] Index2 [_ )]
       | [( _] Index2 [) _]
```

Nous n'avons pas inclus les fonctions d'évaluation d'attributs dans les règles précédentes car elles ne sont que des modifications triviales des fonctions d'évaluation d'attributs données p. 61. L'essentiel des modifications réalisées vient du changement de la syntaxe d'accès aux attributs qui doit tenir compte des changements de bande.

Le squelette de la grammaire obtenue après les modifications que nous venons d'énoncer est donné figure 3.9. La projection de cette grammaire sur chaque bande étant ambiguë (chaque projection est en fait une version modifiée de la grammaire de la figure 3.8), les complexités de l'analyseur généré à partir de cette grammaire sont $O(m^3n^3)$ en temps et $O(m^2n^2)$ en espace, comme pour les équations de programmation dynamique données par Sankoff.

3.3.5 Discussion

Comme nous pouvons le constater, l'extension de la grammaire 1-bande dédiée au modèle thermodynamique en grammaire 2-bandes dédiée au modèle de Sankoff a été relativement aisée. En particulier, les facilités syntaxiques offertes par le générateur d'analyseurs syntaxiques MTSAG2C pour le calcul des attributs physiques des terminaux et la définition de tokens abrégés permettent une description concise de la grammaire et des fonctions d'évaluation d'attributs. Ainsi, la grammaire de la figure 3.9 ne comporte que 19 règles visibles, alors que l'expansion des tokens abrégés (cf. p. 242 et p. 250) entraîne la création de 605 règles réelles, qui auraient été extrêmement pénibles à écrire manuellement. Un autre avantage de la MTSAG 2-bandes par rapport aux équations de programmation dynamique données en 3.3.3 vient du fait que les contraintes d'alignement et d'équivalence de structure imposées par le modèle de Sankoff restent apparentes dans la grammaire, alors qu'elles le sont beaucoup moins dans les équations.

D'une certaine manière, la transformation d'un modèle en équations, et la disparition de l'esprit du modèle de départ dans les équations obtenues, introduisent un seuil de complexité du modèle. Au delà de ce seuil, les équations deviennent difficilement gérables car leur caractère local bloquent la perception de la structure et des contraintes du modèle de départ. Il est donc loisible de se demander si les contraintes d'équivalence structurale du modèle présenté dans cette section n'ont pas été introduites par Sankoff simplement pour obtenir des relations de programmation dynamique. En effet, l'équivalence structurale est une contrainte bien trop forte puisque la quasi totalité des familles d'ARN (y compris les ARN de transfert) possèdent des ARN ayant des composantes structuralement dissemblables. Compte non tenu des complexités de calcul, il aurait donc été souhaitable de supprimer cette contrainte dès le départ pour améliorer la pertinence du modèle. Nous verrons en section 3.4 une MTSAG autorisant le repliement et l'alignement simultanés d'ARN sans utiliser la contrainte d'équivalence structurale de Sankoff.

Cependant, le principe d'alignement et de repliement simultanés introduit par Sankoff reste valable et nous allons voir sur un exemple particulier qu'il permet de trouver la structure correcte de deux ARN de transfert et un alignement raisonnable là où les modèles classiques échouent.

Considérons donc un ARN de transfert du phage T5 pour l'anticodon TGC et un ARN de transfert d'une halobactérie pour le même anticodon (ARN DA0260 et DA0420 de (Sprinzi *et al.*, 1996)). L'alignement correct ainsi que les structures de ces deux ARN sont regroupés ci-dessous :

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAG-GGAGGGUUCGAGUCCCUCUUUGUCCACCA (T5)
(((((((  ((((        ))))((((((       ))))))    (((((       ))))))))))))
GGGCCCAUAGCUCAGU-GGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA--- (halo)
(((((((  ((((        )))) (((((       )))))     (((((       ))))))))))))
```

L'alignement prévu par un programme d'alignement local ne tenant pas compte de la structure (Pearson & Lipman, 1988) fournit un alignement incorrect, mais beaucoup plus cohérent que dans l'exemple donné p. 97 grâce à la plus grande ressemblance des séquences. L'alignement obtenu est reproduit ci-dessous, en même temps qu'un rappel des structures sous-jacentes, qui restent, rappelons le, inconnues du programme d'alignement local.

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGG-AGGGUUCGAGUCCCUCUUUGUCCACCA (T5)
(((((((  ((((        ))))((((((       ))))))   ( ((((       ))))))))))))
GGGCCCAUAGC-UCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA--- (halo)
(((((((  (( ((       )))) (((((       )))))     (((((       ))))))))))))
```

L'application du modèle thermodynamique de Zuker avec les énergies données dans la section 1.7.2.1 p. 51, qui sont les énergies couramment adoptées en pratique, fournit la structure suivante pour l'ARN de transfert du phage T5 :

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA (T5)
((((((   ((((        ))))((((((       ))))))  ((((((       ))))))))))))
```

Cette structure est correcte à deux appariements près, et elle possède la forme en trèfle caractéristique des ARN de transfert. Par contre, le même programme appliqué à l'ARN de l'halobactérie fournit la structure incorrecte suivante :

```
GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA (halo)
(((((((( (((((( (((      (((((((   )))))))  )))))))))(((    )))))))))))
```

La forme caractéristique en trèfle a entièrement disparu. Bien sûr, il aurait été possible de "bidouiller" les énergies du modèle thermodynamique pour trouver une structure correcte dans les deux cas. Cette stratégie "d'amélioration" des énergies, bien qu'ayant longtemps été en vogue à une époque où l'on pensait que le modèle thermodynamique pouvait capturer l'essentiel d'une structure secondaire, est fondamentalement inefficace : quelles que soient les énergies utilisées par le modèle thermodynamique, il sera toujours possible de trouver un ARN très mal replié par le modèle. Nous avons donc testé le modèle de Sankoff en lui soumettant ces deux ARN, tout en gardant les mêmes énergies thermodynamiques que dans les tests précédents et en introduisant des pénalités de gap standard. Le résultat obtenu fut le suivant :

```
GGGCGAAUAGUGUCAGCGGGAGCAC-ACCAGACUUGCAAUCUGGUAGG--GAGGGUUCGAGUCCCUCUUUGUCCACCA (T5)
(((((((  ((((        )))) ((((((       ))))))     (((((       ))))))))))))
GGGCCCAUAGCUC-AGUGGUAGAGUGCCUCCU-UUGCA-AGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA--- (halo)
(((((((  ((((        )))) ((((((       ))))))     (((((       ))))))))))))
```

Cette fois-ci, les deux structures sont correctes et l'alignement est, en l'absence d'information complémentaire, satisfaisant. Le repliement et l'alignement simultanés ont donc eu un effet de rétroaction positif sur la détermination de structure de l'ARN de l'halobactérie. Bien que prometteur, ce résultat se généralise mal à toutes les paires d'ARN de transfert du fait des contraintes d'alignement et d'équivalence de structure du modèle de Sankoff. En particulier, si un des deux ARN possède une boucle supplémentaire et/ou un bras manquant par rapport à l'autre ARN (cas de certains ARN mitochondriaux), le modèle de Sankoff risque de mal replier et aligner les deux ARN (cf. section suivante). Même dans des cas moins extrêmes, le modèle de Sankoff semble avoir du mal à trouver les alignements et les structures corrects sans "bidouillage" des énergies et des coûts d'alignements. La généralisation du modèle de Sankoff introduite dans la section suivante tentera de résoudre ces problèmes.

Notons que l'alignement précédent a été trouvé en un peu plus de 83 secondes et 56 Moctets sur une station DEC Alpha 250 4/266 par l'analyseur syntaxique automatiquement généré à partir de la MTSAG décrite figure 3.9, et après introduction de contraintes d'alignement dans cette MTSAG (cf. section 3.2).

Nous avons également comparé les performances relatives de l'analyseur syntaxique généré à partir de la la MTSAG (analyseur désigné ci-après A pour la MTSAG sans contrainte d'alignement et $A+C$ pour la MTSAG avec contraintes d'alignement) et d'un programme écrit séparément à partir des équations de programmation dynamique données par Sankoff (programme désigné ci-après M). Afin de conserver des temps de calcul raisonnables pour M, nous avons utilisé les deux séquences suivantes, déduites des deux ARN de transfert précédents par troncature d'une portion alignée.

```
GGGCGAAUAGUGUCAGCGGGAGCACGAGGGUUCGAGUCCCUCUUUGUCCACCA
GGGCCCAUAGCUCAGUGGUAGAGUGCUGGGUUGGAAUCCCAGUGGGUCCA
```

Ces ARN ont pour longueurs respectives 53 et 50 bases. Les calculs ont été effectués sur une station DEC Alpha 250 4/266. L'analyseur A a eu besoin de 118 secondes et 72 Moctets pour aligner les deux ARN. Grâce aux contraintes d'alignement, $A+C$ n'a mis que 23 secondes et 20 Moctets pour effectuer le même travail. Finalement, M n'a abouti qu'après plus de 33 heures de calcul et 32 Moctets.

L'analyseur syntaxique A est donc près de mille fois plus rapide et trois fois plus gourmand en mémoire[4] qu'une implémentation directe des équations de programmation dynamique dans M. La comparaison avec $A+C$ est encore plus défavorable pour M. Le rapport des performances est du même ordre de grandeur pour des exemples plus petits. En apportant autant de soin à la programmation de M qu'à celle du noyau présent dans tous les analyseurs syntaxiques générés par MTSAG2C, il aurait peut-être été possible de diviser par dix le temps de calcul de M. Malgré cela, il subsiste une grande différence de vitesse qui mérite une explication.

Nous avions déjà remarqué p. 62 que la version MTSAG 1-bande du modèle thermodynamique était plus rapide qu'une implémentation manuelle des équations de programmation dynamique habituellement associées à ce modèle. La version 2-bandes du modèle de Sankoff accentue nettement ce phénomène pour une raison simple : les équations de programmation dynamique suggèrent naturellement une implémentation bottom-up des calculs, alors que nos analyseurs syntaxiques effectuent ces calculs de façon bottom-up, mais avec un contrôle top-down supplémentaire. Plus précisément,

[4]On retrouve ici, et pour les mêmes raisons, le rapport 3 déjà recontré avec le modèle thermodynamique mono-bande (cf. figure 1.12 p. 63).

les équations de programmation dynamique sont communément évaluées en tabulant les valeurs optimales de toutes les sous-structures possibles. En partant des conditions initiales des équations, les valeurs de sous-structures de plus en plus grandes sont calculées. Aucune vérification n'est faite pour s'assurer que la valeur associée à une sous-structure donnée sera effectivement utilisée dans une structure plus grande incluant cette sous-structure. En fait, la procédure d'évaluation utilisée s'apparente à celle décrite dans l'algorithme d'analyse syntaxique de Cocke-Younger-Kasami, avec tous les inconvénients habituellement associés à cet algorithme. En revanche, l'algorithme GCP m-bandes ne considère jamais une sous-structure qui n'a aucune chance d'être employée dans un arbre de dérivation (c'est la condition minimum d'utilité a priori de la définition 2.11 p. 78). Grâce à ce filtrage top-down, de nombreux calculs inutiles sont évités. Ainsi, le programme A a effectué 98.829.133 évaluations d'attributs pour aligner les deux séquences test précédentes, alors que M a eu besoin de 113.635.145.970 évaluations, soit environ 1150 fois plus.

Dans son article original, Sankoff ne donne aucune indication des performances pratiques de son modèle, même s'il suggère brièvement une forme d'optimisation semblable aux contraintes d'alignement aisément introduites dans une MTSAG. En revanche, il suggère de lever un certain nombre de contraintes d'équivalence structurale pour améliorer la qualité des alignements, sans fonder cette suggestion sur une quelconque base expérimentale et sans donner d'équation correspondante. Tout porte à croire que les équations de programmation dynamique de Sankoff approchent là une limite de complexité d'écriture. Les MTSAG permettent de repousser cette limite en supprimant toute nécessité de construction et de programmation des équations et en permettant au créateur d'un modèle de se concentrer sur les caractéristiques de ce modèle plutôt que sur des détails d'implémentation. Cet avantage va de pair avec la rapidité des analyseurs générés par MTSAG2C. Nous verrons dans la section suivante un exemple de grammaire mettant en œuvre des améliorations des idées originales de Sankoff.

3.4 Généralisation du modèle de Sankoff

L'application des MTSAG au modèle de Sankoff dans la section précédente nous a permis de constater que le principe d'alignement et de repliement simultanés de deux ARN permettait d'obtenir de meilleurs résultats que leur alignement ou leur repliement séparé. Cependant, les contraintes structurales introduites par Sankoff dans son modèle et l'utilisation d'une énergie linéaire de gap dans les alignements ne permettent pas d'obtenir des alignements corrects dans tous les cas.

Dans cette section, nous allons décrire et critiquer une première généralisation du principe d'alignement et de repliement simultanés tel qu'exposé par Sankoff. Puis nous introduirons une deuxième généralisation qui ne présentera pas les inconvénients de la première.

Revenons tout d'abord sur l'exemple, donné en début de section 3.3, d'un alignement de deux ARN de transfert ayant des séquences très différentes, et en particulier des proportions GC/AT inversées. L'alignement correct (tel que référencé dans (Sprinzi *et al.*, 1996)) de ces deux ARN de transfert est le suivant :

Exemple 3.1

```
AGGGUUGUAGUUAAUU----AUAACAUUUGAUUUGCAUUCAAAAAGUAUUG-AAU-AUU-CAAUCUACCUUA---
((((( (  ((((        )))) (((((       )))))    ((((         ))))) )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
((((((   ((((        ))))((((((       ))))))   (((((       )))))))))))
```

L'utilisation de la grammaire relative au modèle de Sankoff décrite dans la section précédente avec ces deux ARN fournit l'alignement suivant :

```
AGGGUUGUAGUUA-AUU---AUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAU--A-UUCAAUCUACCUUA---
         ((((        )))) (((((       )))))    (((((       )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
         ((((        )))) (((((       )))))    (((((       )))))
```

Les trois bras internes de la structure secondaire de chaque ARN sont, dans l'ensemble, correctement identifiés et alignés. L'ajout d'un appariement supplémentaire dans le bras $T\Psi C$ (cf. figure 1.3 p. 10 pour une définition des bras d'un ARN de transfert) est parfaitement normal, en l'absence d'information complémentaire sur la structure tertiaire de l'ARN. Toutes les paires du bras accepteur sont cependant manquées. Ce phénomène s'explique par la présence d'une boucle interne au niveau du bras accepteur du premier ARN. Le modèle de Sankoff n'autorisant pas un alignement des paires d'une boucle d'index 2 avec les bases non appariées d'une autre boucle d'index 2, il lui faudrait introduire un double décalage dans les structures tel que représenté ci-dessous.

```
AGGGUU-GUAGUUA-AUU---AUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAU--A-UUCAAUC-UACCUUA---
(((((  (  ((((        )))) (((((       )))))    (((((       ))))))  )))))
GGGCG-AAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUU-UGUCCACCA
((((( ((  ((((        )))) (((((       )))))    (((((       ))))))) )))))
```

Ce décalage a un coût, mesuré par la pénalité linéaire des gaps introduits et les pénalités des mauvaises correspondances (mismatches). Ce coût, combiné à celui de la boucle interne, rend préférable la conversion en bases extérieures (dont le coût de structure est nul) des paires impliquées. Pour supprimer ces problèmes, il est possible d'une part d'attribuer un coût de structure aux boucles extérieures calqué sur celui des boucles multiples, et d'autre part d'autoriser l'alignement entre une paire d'une boucle d'index 2 et les bases non appariées d'une autres boucle d'index 2. Deux règles de grammaire sont donc ajoutées à la grammaire de la figure 3.9 p. 104 :

Index2 : [• (] Index2 [•)]
  | [(•] Index2 [) •]

Rappelons que le token abrégé • représente n'importe quelle base ou le token vide, tandis que les parenthèses représentent n'importe quel appariement valide. Grâce à ces pénalités et règles supplémentaires, l'alignement obtenu est :

```
AGGGUUGUAGUUA-AUU---AUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAU--A-UUCAAUCUACCUUA---
((((( (  ((((        )))) (((((       )))))    (((((       )))))) )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        )))) (((((       )))))    (((((       ))))))))))))
```

L'alignement et les structures sont bien meilleurs. Par contre, la paire AU terminale du bras de l'anticodon du deuxième ARN n'est pas trouvée car le modèle de Sankoff nécessite un alignement des paires accessibles des boucles multiples.

Notons que les structures trouvées pour les deux ARN de transfert précédents à l'aide du modèle thermodynamique mono-bande de Zuker sont, pour le premier ARN :

```
AGGGUUGUAGUUAAUUAUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAUAUUCAAUCUACCUUA
...((((((......)))))).(((((.......)))))....(((((....)))))........
```

et pour le deuxième :

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
((((((...((((........))))((((((.......))))))..((((((.......))))))))))))....
```

Alors que quasiment toute la structure a été trouvée pour le deuxième ARN (seul un appariement est mal situé), la structure du premier ARN n'est correcte que pour les bras de l'anticodon et $T\Psi C$. Alignement et repliement sont donc complémentaires et permettent d'obtenir, lorsqu'ils sont utilisés simultanément, une structure consensus correcte de deux ARN.

L'ajout de deux règles dans la grammaire du modèle de Sankoff n'a cependant pas résolu le problème de l'alignement de deux séquences possédant des structures résolument différentes et pour lesquelles la notion d'équivalence structurale définie par Sankoff est inapplicable.

Exemple 3.2 *L'alignement des deux ARN suivants montre qu'ils ne sont pas structuralement équivalents :*

```
GGGCGAAU----------------AGAGGGUUCGAGUCCCUCUUUGUCCACCA
((((((                  ((((((       ))))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCUGGGUUGGAAUCCCAGUGGGUCCA---
((((((   ((((       ))))((((((       ))))))))))))
```

L'empilement inséré dans le deuxième ARN n'est pas retrouvé par le modèle de Sankoff (que ce soit avec les deux règles supplémentaires ou non) :

```
GGGCGAA-----------U--A---GAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((                  (((((       ))))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCUGGGUUGGAAUCCCAGUGGGUCCA---
(((((((                  (((((       ))))))))))))
```

Le remplacement des pénalités linéaires de gap par des pénalité affines de gap, suivant une technique qui sera détaillée p. 120, permet d'obtenir un deuxième alignement un peu meilleur, mais où un des bras du deuxième ARN reste absent :

```
GGGCGAAUA----------------GAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((                  (((((       ))))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCUGGGUUGGAAUCCCAGUGGGUCCA---
(((((((                  (((((       ))))))))))))
```

Pour permettre une plus grande souplesse dans l'alignement de deux structures, et donc relaxer la condition d'équivalence structurale du modèle de Sankoff, il faut continuer à généraliser la grammaire 2-bandes obtenue à partir de la grammaire 1-bande du modèle thermodynamique. À cet effet, nous introduisons la notion de 2-grammaire.

Définition 3.2 (2-grammaire)
Soit G une grammaire non-contextuelle 1-bande. La 2-grammaire $\vec{G}^2$ de G est une grammaire 2-bandes définie de la manière suivante:

- *Si A et A' sont deux non-terminaux de G, alors (A, A') est un non-terminal de $\vec{G}^2$;*
- *Le non-terminal de départ de $\vec{G}^2$ est (S, S) ;*
- *Si $A \to u_{-1}B_0u_0 \dots B_pu_p$ et $A' \to u'_{-1}B'_0u'_0 \dots B'_pu'_p$ sont deux règles de grammaire de G contenant le même nombre $p+1$ de non-terminaux, alors les règles de la forme suivante appartiennent à $\vec{G}^2$:*

$$(A, A') \to \left[\begin{smallmatrix} u_{-1} \\ u'_{-1} \end{smallmatrix}\right](B_0, B'_0)\left[\begin{smallmatrix} u_0 \\ u'_0 \end{smallmatrix}\right] \dots (B_p, B'_p)\left[\begin{smallmatrix} u_p \\ u'_p \end{smallmatrix}\right],$$

  où $\left[\begin{smallmatrix} u \\ v \end{smallmatrix}\right]$ représente un alignement 2-bandes des chaînes u et v de longueur égale à $\max(|u|, |v|)$. Il est possible d'avoir $p = -1$ (pas de non-terminaux à droite) ou $u_i = \epsilon$ (pas d'alignement avec u_i).

- *Si $A \to u_{-1}B_0u_0$ est une règle de G et si A' est un non-terminal de G, alors les deux règles suivantes appartiennent à $\vec{G}^2$:*

$$\begin{aligned} (A, A') &\to \left[\begin{smallmatrix} u_{-1} \\ \epsilon \end{smallmatrix}\right](B_0, A')\left[\begin{smallmatrix} u_0 \\ \epsilon \end{smallmatrix}\right] \\ (A', A) &\to \left[\begin{smallmatrix} \epsilon \\ u_{-1} \end{smallmatrix}\right](A', B_0)\left[\begin{smallmatrix} \epsilon \\ u_0 \end{smallmatrix}\right] \end{aligned}$$

Intuitivement, une 2-grammaire décrit l'ensemble des couples de dérivations compatibles, dans le sens où un non-terminal d'une règle utilisée sur la première bande doit toujours être "aligné" avec un non-terminal d'une règle utilisée sur la deuxième bande. Les paires (A, A') représentent cet alignement des non-terminaux.

Exemple 3.3 *La grammaire suivante contient 7 règles :*

$$\begin{aligned} &A \to B \mid C \mid A\,B \mid A\,C \\ &B \to (\,A\,) \\ &C \to \bullet \mid C\,\bullet \end{aligned}$$

Sa 2-grammaire contient 47 règles :

$$
\begin{aligned}
(A,A) \rightarrow & (B,A) \mid (B,B) \mid (B,C) \mid (C,A) \mid (C,B) \mid (C,C) \mid (A,B) \mid (A,C) \\
& \mid (A,A)(B,B) \mid (A,A)(B,C) \mid (A,A)(C,C) \mid (A,A)(C,B) \\
(A,B) \rightarrow & \left[\begin{smallmatrix}\epsilon\\(\end{smallmatrix}\right](B,A)\left[\begin{smallmatrix}\epsilon\\)\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\epsilon\\(\end{smallmatrix}\right](C,A)\left[\begin{smallmatrix}\epsilon\\)\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\epsilon\\(\end{smallmatrix}\right](A,A)\left[\begin{smallmatrix}\epsilon\\)\end{smallmatrix}\right] \mid (B,B) \mid (C,B) \\
(B,A) \rightarrow & \left[\begin{smallmatrix}(\\\epsilon\end{smallmatrix}\right](A,B)\left[\begin{smallmatrix})\\\epsilon\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}(\\\epsilon\end{smallmatrix}\right](A,C)\left[\begin{smallmatrix})\\\epsilon\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}(\\\epsilon\end{smallmatrix}\right](A,A)\left[\begin{smallmatrix})\\\epsilon\end{smallmatrix}\right] \mid (B,B) \mid (B,C) \\
(A,C) \rightarrow & (B,C)\left[\begin{smallmatrix}\epsilon\\\bullet\end{smallmatrix}\right] \mid (C,C)\left[\begin{smallmatrix}\epsilon\\\bullet\end{smallmatrix}\right] \mid (A,C)\left[\begin{smallmatrix}\epsilon\\\bullet\end{smallmatrix}\right] \mid (B,C) \mid (C,C) \\
(C,A) \rightarrow & (C,B)\left[\begin{smallmatrix}\bullet\\\epsilon\end{smallmatrix}\right] \mid (C,C)\left[\begin{smallmatrix}\bullet\\\epsilon\end{smallmatrix}\right] \mid (C,A)\left[\begin{smallmatrix}\bullet\\\epsilon\end{smallmatrix}\right] \mid (C,B) \mid (C,C) \\
(B,B) \rightarrow & \left[\begin{smallmatrix}(\\(\end{smallmatrix}\right](A,A)\left[\begin{smallmatrix})\\)\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}(\\\epsilon\end{smallmatrix}\right](A,B)\left[\begin{smallmatrix})\\\epsilon\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\epsilon\\(\end{smallmatrix}\right](B,A)\left[\begin{smallmatrix}\epsilon\\)\end{smallmatrix}\right] \\
(B,C) \rightarrow & \left[\begin{smallmatrix}(\\\epsilon\end{smallmatrix}\right](A,C)\left[\begin{smallmatrix})\\\bullet\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}(\\\epsilon\end{smallmatrix}\right](A,C)\left[\begin{smallmatrix})\\\epsilon\end{smallmatrix}\right] \mid (B,C)\left[\begin{smallmatrix}\epsilon\\\bullet\end{smallmatrix}\right] \\
(C,B) \rightarrow & \left[\begin{smallmatrix}\epsilon\\(\end{smallmatrix}\right](C,A)\left[\begin{smallmatrix}\bullet\\)\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\epsilon\\(\end{smallmatrix}\right](C,A)\left[\begin{smallmatrix}\epsilon\\)\end{smallmatrix}\right] \mid (C,B)\left[\begin{smallmatrix}\bullet\\\epsilon\end{smallmatrix}\right] \\
(C,C) \rightarrow & (C,C)\left[\begin{smallmatrix}\bullet\\\bullet\end{smallmatrix}\right] \mid (C,C)\left[\begin{smallmatrix}\bullet\\\epsilon\end{smallmatrix}\right] \mid (C,C)\left[\begin{smallmatrix}\epsilon\\\bullet\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\bullet\\\bullet\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\bullet\\\epsilon\end{smallmatrix}\right] \mid \left[\begin{smallmatrix}\epsilon\\\bullet\end{smallmatrix}\right]
\end{aligned}
$$

Après avoir défini la notion de 2-grammaire, nous pouvons l'étendre en notion de 2-grammaire S-attribuée obtenue à partir d'une grammaire S-attribuée. Les attributs et fonctions d'évaluation d'attributs de la 2-grammaire sont naturellement obtenus à partir des fonctions correspondantes de la grammaire 1-bande de départ[5].

Les contraintes d'optimisation et de sélection d'une 2-grammaire $\vec{G}^2$ ne sont pas directement déductibles des contraintes correspondantes de G. Il faut donc adapter les contraintes au problème. Dans le cas d'une minimisation d'énergie, la contrainte d'optimisation sera une minimisation de la somme des énergies calculées sur chaque bande. Dans le cas d'une maximisation de probabilité, une maximisation du produit des probabilités calculées sur chaque bande sera utilisée.

Nous avons appliqué le principe de contruction des 2-grammaires S-attribuées à une grammaire générique 1-bande du modèle thermodynamique de repliement des ARN. Pour construire la grammaire générique 1-bande, nous avons considéré les cinq non-terminaux utilisés de la grammaire de la figure 1.11 p. 61. Ces cinq non-terminaux sont :

Structure pour les structures secondaires dans leur ensemble,

Loop pour les boucles,

Index2 pour les boucles d'index 2,

LoopSeq pour les boucles multiples,

SingleStrand pour les suite de bases non appariées.

Toutes les combinaisons valides de ces cinq non-terminaux ont été représentées dans une grammaire S-attribuée dont les règles sont données en figure 3.10.

[5]Après extension éventuelle pour tenir compte de l'information apportée par l'alignement sous la forme d'un coût linéaire ou affine de gap par exemple.

$$
\begin{aligned}
&A \rightarrow B \mid E \mid A\,B \mid B\,A \mid A\,E \mid E\,A \mid A\bullet \mid \bullet A \mid \bullet A\,\bullet \\
&B \rightarrow (\,C\,) \mid (\,D\,) \mid (\,E\,) \\
&C \rightarrow B \mid C \mid C\,E \mid E\,C \mid C\bullet \mid \bullet C \mid \bullet C\,\bullet \\
&D \rightarrow B \mid E \mid D\,B \mid B\,D \mid D\,E \mid E\,D \mid D\bullet \mid \bullet D \mid \bullet D\,\bullet \\
&E \rightarrow E\,E \mid E\bullet \mid \bullet E \mid \bullet E\,\bullet
\end{aligned}
$$

Figure 3.10: Grammaire 1-bande utilisée pour généraliser le modèle de Sankoff. Pour conserver une figure de taille raisonnable, les non-terminaux ont été abrégés : A pour Structure, B pour Loop, C pour Index2, D pour LoopSeq, E pour SingleStrand. Les mêmes notations que dans la figure 1.11 ont été utilisées pour représenter les bases et les appariements.

La 2-grammaire construite à partir de cette grammaire comporte 828 règles[6]. Après simplification de la grammaire obtenue (essentiellement en regroupant les productions concernant les terminaux), 244 règles comportant de nombreux tokens abrégés (cf. section A.4.2.5 p. 242) ont été énumérées. Ces règles ont été développées par MTSAG2C en une grammaire comportant 4356 règles et, bien entendu, $5 \times 5 = 25$ non-terminaux.

Notons que l'écriture, selon la méthode classique, des équations de programmation dynamique correspondant à ces 244 règles aurait représenté un travail considérable.

Exemple 3.4 *L'application de l'analyseur syntaxique généré à partir de cette grammaire 2-bandes aux ARN de l'exemple 3.2 a permis de retrouver l'empilement manqué par le modèle de Sankoff :*

```
GGGCGAAU----------------AGAGGGUUCGAGUCCCUCUUUGUCCACCA
((((((                  ((((((       ))))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCUGGGUUGGAAUCCCAGUGGGUCCA---
((((((   ((((       ))))((((((       ))))))))))))
```

Exemple 3.5 *Le même programme appliqué aux deux ARNt de l'exemple 3.1 permet de trouver la paire manquante au niveau du bras de l'anticodon :*

```
AGGGUUGUAGUUAAUU----AUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAU---AUUCAAUCUACCUUA---
((((( (  ((((        )))) (((((       )))))    (((((       )))))) )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        ))))((((((       ))))))   (((((       ))))))))))))
```

Les bons résultats précédents ont un prix. Même en utilisant des contraintes d'alignement, il faut plus de 200 Moctets de mémoire et plus de 6 heures de temps CPU (sur un Pentium 120 Mhz) pour obtenir l'alignement des deux ARN de transfert de l'exemple précédent, ou 60 Moctets et 1 heure pour l'alignement des deux ARN de

[6] Bien entendu, nous n'avons pas écrit manuellement cette 2-grammaire et ses fonctions d'évaluation d'attributs. Nous avons utilisé un programme de génération automatique d'une 2-grammaire S-attribuée à partir d'une grammaire S-attribuée.

```
%nb_tapes 1

%attribute "int" idx
%token A { $$.idx = IDX_A; }
%token C { $$.idx = IDX_C; }
%token G { $$.idx = IDX_G; }
%token U { $$.idx = IDX_U; }

%abbreviation ( A U G C G U ) U A C G U G
%abbreviation • A C G U

%attribute "short" E
%terminal_attribs {}
%default_nonterminal_attribs <E>

%default_constraint
        { if ($$.E > INFINITY / 2) return 0; }
        { if ($$.E < $!.E) $!.E = $$.E; }

%%

RNA : RNA RNA { $$.E = $1.E + $2.E; }
    | DS          { $$.E = $1.E + 10; }
    | • RNA       { $$.E = $2.E + 20; }
    | RNA •       { $$.E = $1.E + 20; }
    | • RNA •     { $$.E = $2.E + 40; }
    | •           { $$.E = 20; };

DS : ( RNA ) { if ($2[2].len < 3) $$.E = INFINITY;
               else $$.E = $2.E + 360; }
   | ( DS )   { $$.E = $2.E - 100; };

%%
```

Figure 3.11: Cette grammaire exprime seulement le fait qu'une structure secondaire est une suite de bases et d'empilements, tandis qu'un empilement est formé en fermant une structure ou un empilement à l'aide d'une paire de bases. Les énergies des appariements sont fixées et égale à -100, tandis que les énergies des boucles sont égales à 460 plus 20 par base isolée et 10 par paire intérieure. Les deux non-terminaux ont un unique attribut E (pour l'énergie) et une minimisation de l'énergie est réalisée. La contrainte de sélection empêche les boucles d'avoir une taille inférieure à 3 bases.

l'exemple 3.2. Les 25 non-terminaux et 4356 règles de la grammaire utilisée sont clairement excessifs.

La grammaire 1-bande de la figure 3.10, à l'origine de l'énorme 2-grammaire précédente, est en fait une généralisation de celle du modèle de Zuker. Nous allons simplifier cette grammaire afin d'obtenir une 2-grammaire plus petite, mais suffisante pour aligner et replier correctement les mêmes exemples que précedemment. La grammaire 1-bande obtenue est donnée figure 3.11. Cette grammaire simpliste est bien sûr incapable de trouver à elle seule la structure correcte d'un ARN de transfert, comme le montrent les exemples suivants (la structure incorrecte est celle retournée par l'analyseur syntaxique tandis que la structure correcte est celle documentée dans (Sprinzi *et al.*, 1996)).

Exemple 3.6 *ARN de transfert de la drosophile (da4880 dans (Sprinzi* et al., *1996))*

```
AGGGUUGUAGUUAAUUAUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAUAUUCAAUCUACCUUA
   ((((((((  )))))))) (((((         )))))    ((((((  ))))))          incorrect
((((( (  ((((    )))) (((((         )))))    ((((        ))))) )))))  correct
```

Exemple 3.7 *ARN de transfert du phage T5 (da0260)*

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
 ((((      )))) (((((             )))))      ((((((((((        ))))))))))       incorrect
(((((((  ((((          ))))((((((        ))))))   (((((        ))))))))))))     correct
```

Exemple 3.8 *ARN de transfert de l'halobactérie (da0420)*

```
GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA
(((((((((((((( ((((     (((((((  ))))))) )))))))))))         )))))))    incorrect
(((((((  ((((      )))) (((((      )))))     (((((      ))))))))))))    correct
```

L'incapacité d'une grammaire simpliste à trouver une structure secondaire correcte pour un ARN donné est précisément ce qui a conduit à l'introduction de modèles thermodynamiques plus sophistiqués, dont celui de Zuker. Nous allons maintenant vérifier que l'introduction d'alignements via une 2-grammaire permet, grâce au supplément d'information ainsi disponible, de retrouver des alignements et des structures corrects.

Nous avons donc transformé la grammaire de la figure 3.11 en 2-grammaire, en gérant les coûts d'alignement suivant le modèle affine. Un bonus fixe a également été attribué aux appariements alignés. La 2-grammaire obtenue comporte 4 non-terminaux et 1396 règles au lieu des 25 non-terminaux et 4356 règles de la 2-grammaire du modèle de Zuker. Les ARN de transfert précédents ont été utilisés pour tester le comportement de cette 2-grammaire.

Exemple 3.9 *ARN da4880 sur la première bande et da0260 sur la deuxième. L'alignement et les structures trouvés sont :*

```
AGGGUUGUAGUUA----AUUAUAACAUUUGAUUUGCAUUCAAAAAGUAUUGAAU---AUUCAAUCUACCUUA---
(((((    ((((        )))) (((((       )))))    (((((       )))))  )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((    ((((        ))))((((((       )))))) (((((((       ))))))))))))
```

L'alignement et les structures corrects sont :

```
AGGGUUGUAGUUAAUU----AUAACAUUUGAUUUGCAUUCAAAAAGUAUUG-AAU-AUU-CAAUCUACCUUA---
((((( (  ((((        )))) (((((       )))))    ((((         ))))) )))))
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        ))))((((((       ))))))   (((((       ))))))))))))
```

Sur l'ARN da0260, les bases de l'extrémité 3' de la boucle interne du bras accepteur ont été appariées dans le bras TΨC. Elles ont été laissées non appariées sur l'ARN da4880. L'appariement supplémentaire fermant l'épingle à cheveux du bras TΨC de da4880 est normal en l'absence d'information complémentaire sur la structure tertiaire de l'ARN.

Exemple 3.10 *ARN da0260 sur la première bande et da0420 sur la deuxième. L'alignement et les structures trouvés sont :*

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAG-GGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        ))))((((((       ))))))    (((((       ))))))))))))
GGGCCCAUAGCUC-AGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA---
(((((((  ((((        )))) (((((((   )))))))     (((((       ))))))))))))
```

L'alignement et les structures corrects sont :

```
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAG-GGAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((  ((((        ))))((((((       ))))))    (((((       ))))))))))))
GGGCCCAUAGCUCAGU-GGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA---
(((((((  ((((        )))) (((((       )))))     (((((       ))))))))))))
```

l'alignement et la structure trouvés par la grammaire sont exacts, à l'exception des deux paires supplémentaires dans la boucle de l'anticodon, normales en l'absence d'information complémentaire sur la structure tertiaire de da0420. Notons que la structure trouvée pour da0260 dans cet alignement est différente de celle trouvée dans l'exemple précédent.

Exemple 3.11 *ARN da4880 sur la première bande et da0420 sur la deuxième. L'alignement et les structures trouvés sont :*

```
AGGGUUGUAGUUAA---UUAUAACAUUUGAUUUGCAUUCAAAAAG-UAUUGAAU---AUUCAAUCUACCUUA
(((((    ((((       )))) (((((       )))))     (((((       )))))  )))))
GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA
(((((((  ((((       )))) (((((((   )))))))     (((((       ))))))))))))
```

L'alignement et les structures corrects sont :

```
AGGGUUGUAGUUAAUU---AUAACAUUUGAUUUGCAUUCAAAAAG-UAUUG-AAU-AUU-CAAUCUACCUUA
((((( (  ((((       )))) (((((       )))))     ((((         )))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA
(((((((  ((((       )))) (((((       )))))     (((((       ))))))))))))
```

L'alignement et la structure trouvés par la 2-grammaire sont exacts, à l'exception d'une paire adjacente à la boucle interne dans le bras accepteur de da4880, et d'une paire supplémentaire dans le bras $T\Psi C$ de da4880. La structure de da0420 est la même ici que dans l'exemple précédent.

En résumé, les alignements et repliements trouvés par la 2-grammaire issue de la grammaire de la figure 3.11 sont d'excellente qualité, quasi-parfaits s'il n'est pas tenu compte des défauts d'alignement dans les boucles[7]. Tous ces alignements ont été obtenus en utilisant environ 40 Moctets de mémoire et 10 minutes de temps CPU (sur un Pentium 120 Mhz). Les gains par rapport à la 2-grammaire originale sont donc significatifs (5 fois moins de mémoire, 36 fois moins de CPU) et montrent qu'il n'est pas nécessaire d'utiliser un modèle thermodynamique sophistiqué pour calculer de bons repliements si ce modèle est épaulé par des informations d'alignement. Ce résultat est conforme à l'intuition car les alignements et repliements d'ARN de référence (avec lesquels tous les produits d'algorithmes d'alignement et/ou de repliement sont comparés) sont généralement affinés manuellement sans connaissance approfondie des énergies de boucles ou d'appariements, mais simplement à partir de quelques remarques simples :

- une paire est préférable à une absence de paire,
- l'alignement des paires doit être favorisé,
- les petites boucles sont préférables aux grandes boucles,
- la multiplication des gaps est à éviter.

[7] Les alignements "corrects" utilisés en référence dans les trois exemples précédents sont obtenus après alignement de plusieurs centaines d'ARN et prise en compte des pseudo-nœuds entre les bases des boucles D et $T\Psi C$ dans la structure tertiaire des ARN de transfert.

La généralisation des tests précédents sur 50 paires d'ARN choisies au hasard dans la partie I de (Sprinzi *et al.*, 1996) a permis d'obtenir des résultats de qualité similaire dans la totalité des cas[8]. Le choix aléatoire d'une paire formée d'un ARN de la partie I et d'un ARN de la partie III de (Sprinzi *et al.*, 1996) n'a pas toujours permis de retrouver des alignements et des structures corrects, sans doute pour deux raisons :

- les ARN de la partie I comportent tous quatre bras, alors que les ARN de la partie III ont en général un bras en moins,
- les paramètres de notre 2-grammaire simplifiée (pénalités de boucles, d'alignement, énergies des appariements) ont été choisis au jugé et n'ont pas été optimisés[9].

Notons pour mémoire que l'empilement manquant des ARN de l'exemple 3.4 trouvé par la 2-grammaire du modèle de Zuker est également trouvé par la 2-grammaire simplifiée :

```
GGGCGAAUA----------------GAGGGUUCGAGUCCCUCUUUGUCCACCA
(((((((                  (((((       ))))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCUGGGUUGGAAUCCCAGUGGGUCC---A
(((((((  ((((       )))) (((((       ))))))))))))
```

3.5 Alignement des ARN – Modèle de Corpet et Michot

Dans la section précédente, nous avons décrit un modèle d'alignement et de repliement simultanés de deux ARN proposé par Sankoff. Lorsque les deux ARN ont des tailles respectives m et n, le modèle de Sankoff débouche sur des méthodes (équations de programmation dynamique ou MTSAG) dont les complexités sont $O(m^3n^3)$ en temps et $O(m^2n^2)$ en espace. Ces complexités étant élevées, Corpet et Michot ont proposé un modèle différent d'alignement et de repliement simultanés (Corpet & Michot, 1994). Ce modèle les a conduit à écrire, une fois de plus, des équations de programmation dynamique puis, à partir de ces équations, un algorithme dont la complexité est $O(m^2n^3)$ en temps et $O(m^2n^2)$ en espace. Nous allons voir que la description du modèle de Corpet et Michot sous la forme d'une MTSAG permet d'obtenir sans effort un analyseur syntaxique dont la complexité est $O(mn^3)$ en temps et $O(mn^2)$ en espace, soit un gain d'un facteur m.

Le principe du modèle de Corpet et Michot est simple : plutôt que d'aligner et de replier simultanément deux ARN dont la structure est inconnue, il suffit d'aligner et de replier un ARN dont la structure est inconnue contre un ARN dont la structure est déjà connue. Les coûts d'alignement sont choisis affines et l'alignement des structures est favorisé en accordant un bonus à toute paire de bases du deuxième ARN alignée contre une paire de bases déjà appariées dans la première structure. Le bonus accordé

[8] À condition que les deux ARN ne soient pas identiques à plus de 90%, auquel cas l'analyse comparative n'apporte rien (James *et al.*, 1989).

[9] Le fait d'obtenir des résultats cohérents sur les ARNt avec des paramètres non optimisés montre à quel point la 2-grammaire est robuste.

est fonction des deux bases impliquées. Il est plus élevé pour les paires putatives telles (G,C), (A,U) ou (G,U) que pour les autres. D'une certaine manière, ce modèle ne “voit” pas d'autres appariements que ceux donnés dans la structure connue.

Partant de ce principe simple, Corpet et Michot déduisent un ensemble d'équations de programmation dynamique (figure 3.12). La complexité de ces équations n'a rien à envier à celle des équations proposées par Sankoff pour son modèle et il est particulièrement difficile de retrouver le modèle initial à partir des seules équations. Les équations sont ensuite traduites en un algorithme de calcul dont la complexité est $O(m^2n^3)$ en temps et $O(m^2n^2)$ en espace (Corpet & Michot, 1994).

Pour pouvoir décrire le modèle de Corpet et Michot à l'aide d'une MTSAG 2-bandes, il faut commencer par trouver un moyen de transformer la structure connue du premier ARN en information utilisable par une grammaire. En fait, la transformation est assez simple puisqu'il suffit de remplacer chacune des bases du premier ARN par un des symboles '(', ')' (pour les bases appariées) ou '.' pour les bases non-appariées. Les bases sous-jacentes aux nouveaux symboles deviennent leurs attributs physiques. La première bande contient ainsi le code, sous forme d'expression correctement parenthésée, du premier ARN, celui dont la structure est connue (figure 3.13). La deuxième bande contient le deuxième ARN laissé tel quel.

Il reste maintenant à transformer le modèle en grammaire à deux bandes. Remarquons tout d'abord que la première bande est entièrement décrite par la grammaire suivante :

$$\begin{aligned} S &\to \text{'.'} \mid D \mid S\,\text{'.'} \mid S\,D \\ D &\to \text{'('}\,S\,\text{')'} \end{aligned}$$

Cette grammaire LR(1) et non-récursive à droite traduit le fait que la première bande est une structure récursivement formée d'une suite de '.' et de structures fermées par deux parenthèses correctement appariées. Nous cherchons à transformer cette grammaire en MTSAG 2-bandes alignant la première bande contre la deuxième avec pénalité affine de gap. Dans cette future MTSAG, les terminaux 2-bandes sont divisés en 3 catégories :

- alignements avec $\left[{\text{'.'} \atop \bullet}\right]$, $\left[{\text{'('} \atop \bullet}\right]$ et $\left[{\text{')'} \atop \bullet}\right]$,
- délétions avec $\left[{\text{'.'} \atop -}\right]$, $\left[{\text{'('} \atop -}\right]$ et $\left[{\text{')'} \atop -}\right]$,
- insertions avec $\left[{- \atop \bullet}\right]$.

Le symbole $\bullet$ représente ici n'importe quelle base du second ARN.

Pour obtenir la version 2-bandes de la grammaire précédente, il suffit de remplacer les terminaux 1-bande par les différentes combinaisons de terminaux 2-bandes préservant la structure de la première bande. La grammaire obtenue est la suivante :

$$S \to \left[{\text{'.'} \atop \bullet}\right] \mid \left[{\text{'.'} \atop -}\right] \mid \left[{- \atop \bullet}\right] \mid D \mid S\left[{\text{'.'} \atop \bullet}\right] \mid S\left[{\text{'.'} \atop -}\right] \mid S\left[{- \atop \bullet}\right] \mid S\,D$$

$$D \to \left[{\text{'('} \atop \bullet}\right] S \left[{\text{')'} \atop \bullet}\right] \mid \left[{\text{'('} \atop -}\right] S \left[{\text{')'} \atop \bullet}\right] \mid \left[{\text{'('} \atop \bullet}\right] S \left[{\text{')'} \atop -}\right] \mid \left[{\text{'('} \atop -}\right] S \left[{\text{')'} \atop -}\right]$$

Soit $a_0 \dots a_m$ un premier ARN A, dont on connaît la structure secondaire, et soit φ la fonction de $\{0, \dots, m\}$ dans $\{\emptyset, 0, \dots, m\}$ telle que $\big(a_k, a_{\varphi(k)}\big)$ est une paire de la structure secondaire de A, ou $\varphi(k) = \emptyset$ si a_k n'est pas appariée. Soit $b_0 \dots b_n$ un second ARN, dont on cherche la structure et l'alignement contre le premier.
Pour $0 \leqslant i \leqslant k \leqslant m$ et $0 \leqslant j \leqslant l \leqslant n$, soit $C(i,j,k,l)$ le score maximum d'alignement entre $a_i \dots a_k$ et $b_j \dots b_l$ en supposant que les extrémités sont alignées. Pour $0 \leqslant i \leqslant \varphi(i) \leqslant k \leqslant m$ et $0 \leqslant j \leqslant l \leqslant m$, soit $D(i,j,k,l)$ le score maximum d'alignement entre les mêmes sous-séquences en supposant que les extrémités sont alignées et que la base $a_{\varphi(i)}$ est alignée avec un élément de $b_j \dots b_l$ (i.e. pas de délétion).
Soit $S(i,j)$ la valeur de similarité pour un alignement de a_i et b_j et soit $P(j,l)$ le bonus accordé pour un appariement entre b_j et b_l. Alors, pour $i < k$ et $j < l$:

$$C(i,j,k,l) = \max \begin{cases} S(i,j) + N^{+}(C;i,j,k,l) & \text{si } \varphi(i) \notin [i,k] \\ S(i,j) + P(j,l) + N^{+}(C;i,j,k,l) & \text{si } \varphi(i) = k \\ \max\big(D(i,j,k,l), S(i,j) + N^{+}(C;i,j,k,l)\big) & \text{si } i < \varphi(i) < k \end{cases}$$

et pour $i < \varphi(i) < k$ et $j+1 < l$

$$D(i,j,k,l) = \max \begin{cases} S(k,l) + N^{-}(D;i,j,k,l) \quad \text{si } \forall h \in [\varphi(i),k], \varphi(h) \notin [\varphi(i),k] \\ \max\limits_{j<l'<l} \Big\{ C\big(i,j,\varphi(i),l'\big) \\ \qquad\qquad + C\big(\varphi(i),l',k,l\big) - S\big(\varphi(i),l'\big) \Big\} \quad \text{si } \exists h, \varphi(i) < h < \varphi(h) \leqslant k \end{cases}$$

Le coût des gaps est donné par $-\beta \times (\text{taille du gap}) - \alpha$ et

$$N^{+}(F;i,j,k,l) = \max \begin{cases} F(i+1,j+1,k,l) \\ \max\limits_{i<i'\leqslant k} F(i',j+1,k,l) - (i'-i-1)\beta - \alpha \\ \max\limits_{j<j'\leqslant l} F(i+1,j',k,l) - (j'-j-1)\beta - \alpha \end{cases}$$

$$N^{-}(F;i,j,k,l) = \max \begin{cases} F(i,j,k-1,l-1) \\ \max\limits_{i\leqslant k'<k} F(i,j,k',l-1) - (k'-k-1)\beta - \alpha \\ \max\limits_{j\leqslant l'<l} F(i,j,k-1,l') - (l'-l-1)\beta - \alpha \end{cases}$$

Finalement, les conditions initiales sont

- $C(i,j,i,j) = S(i,j)$ pour $0 \leqslant i \leqslant m$ et $0 \leqslant j \leqslant n$;
- $C(i,j,k,j) = -\infty$ pour $0 \leqslant i < k \leqslant m$ et $0 \leqslant j \leqslant n$;
- $C(i,j,i,l) = -\infty$ pour $0 \leqslant i \leqslant m$ et $0 \leqslant j < l \leqslant n$;
- $D(i,j,\varphi(i),l) = C(i,j,\varphi(i),l)$ pour $0 \leqslant i < \varphi(i) \leqslant m$ et $0 \leqslant j < l \leqslant n$;
- $D(i,j,k,j+1) = -\infty$ pour $0 \leqslant i < \varphi(i) < k \leqslant m$ et $0 \leqslant j < n$.

Figure 3.12: Équations de programmation dynamique données dans (Corpet & Michot, 1994), avec quelques adaptations mineures.

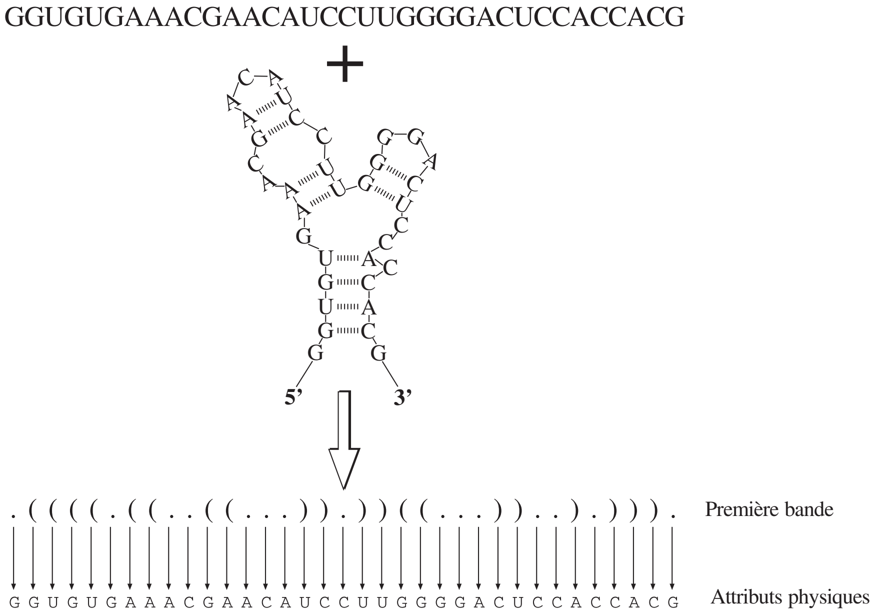


Figure 3.13: Un ARN dont la structure est connue peut facilement être converti en une première bande contenant uniquement des parenthèses et des points et dont les symboles ont pour attribut physique la base de départ. En fait, on passe d'une représentation où la structure secondaire est une annotation de la structure primaire à une représentation où la structure primaire est une annotation de la structure secondaire.

Pour gérer les pénalités affines d'alignement, il faut savoir si un terminal de catégorie insertion prolonge une insertion existante. De même, il faut savoir si un terminal de catégorie délétion prolonge une délétion existante. Des attributs physiques sont donc ajoutés à chaque non-terminal pour stocker ces informations. Ces attributs physiques (`l` et `r` respectivement) valent 1 si le non-terminal se termine (à gauche et à droite respectivement) par une délétion, 0 par un alignement et -1 par une insertion. La MTSAG de la figure 3.14 est alors obtenue.

La projection de cette MTSAG sur la première bande étant trivialement réductible en une grammaire LR(1) non-récursive à droite[10], l'analyseur généré par MTSAG2C a une complexité de $O(mn^3)$ en temps et $O(mn^2)$ en espace. L'utilisation des MTSAG à la place des équations de programmation dynamique nous a donc permis de :

- décrire plus facilement le modèle,
- obtenir un code plus efficace, et ce sans effort puisque MTSAG2C se charge automatiquement de générer l'analyseur syntaxique.

La meilleure complexité théorique va de pair avec une plus grande performance pratique (i.e. les constantes cachées ne viennent pas tout gâcher) puisque l'analyseur généré à partir de la MTSAG n'a mis que 2 secondes et 4,5 Moctets (sans contrainte d'alignement) pour aligner et replier un ARN de transfert contre un ARN de transfert déjà replié, alors que le code d'origine aimablement fourni par Florence Corpet a mis 12 secondes et 14 Moctets pour obtenir le même résultat.

[10] C'est la grammaire $S \rightarrow$ '.' | S '.' | $S\ D \quad D \rightarrow$ '(' S ')'.

```
%nb_tapes 2

%empty _
%attribute "int" idx
%token A { $$.idx = 0; }
%token C { $$.idx = 1; }
%token G { $$.idx = 2; }
%token U { $$.idx = 3; }
%token '(' ')' '.'

%abbreviation • A C G U

%attribute "int" E
%attribute "short" l r
%terminal_attribs {}
%default_nonterminal_attribs < E l r>

%default_constraint
        {}
        { if ($$.E > $!.E)
             {$!.E = $$.E; $!.l = $$.l; $!.r = $$.r; }
%%

Structure
    : ['.' •]
        { $$.E = MATCH; $$.l = $$.r = 0; }
    | ['.' _]
        { $$.E = GAP_D + EXT_D;
          $$.l = $$.r = 1; }
    | [_ •]
        { $$.E = GAP_I + EXT_I;
          $$.l = $$.r = -1; }
    | DoubleStrand
        { $$.E = $1.E;
          $$.l = $1.l; $$.r = $1.r; }
    | Structure ['.' •]
        { $$.E = $1.E + MATCH;
          $$.l = $1.l; $$.r = 0; }
    | Structure ['.' _]
        { $$.E = $1.E + EXT_D;
          $$.l = $1.l; $$.r = 1;
          if ($1.r <= 0) $$.E += GAP_D; }
    | Structure [_ •]
        { $$.E = $1.E + EXT_I;
          $$.l = $1.l; $$.r = -1;
          if ($1.r >= 0) $$.E += GAP_I; }
    | Structure DoubleStrand
        { $$.E = $1.E + $2.E;
          $$.l = $1.l; $$.r = $2.r;
          if ($1.r < 0 && $2.l < 0)
               $$.E += -GAP_I;
          if ($1.r > 0 && $2.l > 0)
               $$.E += -GAP_D; };
DoubleStrand
    : ['(' •] Structure [')' •]
        { $$.E = $2.E + 2*MATCH +
               PAIR[$1[2].left[0].idx][$3[2].left[0].idx];
          $$.l = $$.r = 0; }
    | ['(' _] Structure [')' •]
        { $$.E = $2.E + MATCH + EXT_D;
          if ($2.l <= 0) $$.E += GAP_D;
          $$.l = 1; $$.r = 0; }
    | ['(' •] Structure [')' _]
        { $$.E = $2.E + MATCH + EXT_D;
          if ($2.r <= 0) $$.E += GAP_D;
          $$.l = 0; $$.r = 1; }
    | ['(' _] Structure [')' _]
        { $$.E = $2.E + 2 * EXT_D;
          if ($2.l <= 0) $$.E += GAP_D;
          if ($2.r <= 0) $$.E += GAP_D;
          $$.l = 1; $$.r = 1; };
```

Figure 3.14: Dans cette MTSAG à deux bandes, les coûts d'alignement sont constants et valent respectivement `MATCH` pour l'alignement de deux bases, `GAP_D` pour la création d'une délétion et `EXT_D` pour son extension, `GAP_I` pour la création d'une insertion et `EXT_I` pour son extension. Il aurait été possible, comme le font Corpet et Michot, de créer des coûts d'alignement dépendant de la position en stockant ces coûts sous forme d'attributs physiques des tokens de la première bande.

3.6 Alignement des ARN – Modèles de Bafna *et al.*

Bafna *et al.* (1995) ont étudié plusieurs problèmes de calcul de similarité entre deux chaînes d'ARN, en tenant compte à la fois de la structure primaire des chaînes d'ARN et de leur structure secondaire. Nous appellerons chaîne augmentée une chaîne d'ARN pour laquelle l'information de structure est disponible.

Les premiers problèmes considérés par Bafna *et al.* concernent la recherche d'une chaîne augmentée (le motif) dans une autre chaîne augmentée plus longue. Nous ne reviendrons ni sur ces problèmes ni sur les solutions proposées car elles utilisent des variantes de l'algorithme de Knuth-Morris-Pratt (Knuth *et al.*, 1977), bien plus adaptées que nos grammaires à la résolution efficace de ce type de problème. Nous allons nous concentrer sur l'adaptation aux MTSAG du second type de problème considéré par Bafna *et al.* : l'alignement de chaînes augmentées. Ce type d'alignement avait déjà été abordé par d'autres auteurs (Shapiro, 1988; Konings & Hogeweg, 1989), mais de manière très partielle car ils ne tenaient pas correctement compte de la structure secondaire dans leurs recherches d'alignements. Nous présenterons d'abord différents problèmes d'alignement, puis nous comparerons les performances des algorithmes d'analyse syntaxique aux performances des solutions proposées par Bafna *et al.*

À l'instar de Bafna, nous supposerons dans la suite que la structure d'une chaîne augmentée ne contient pas de pseudo-nœud.

Remarque 3.1 En l'absence de pseudo-nœud, une chaîne augmentée peut-être écrite sous la forme d'une expression correctement parenthésée dont chaque symbole contient la base sous-jacente sous forme d'attribut (cf. section précédente) :

```
(((..(((..))).((..))..)))..   expression
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
GCAGUCAUGAAUGCGCAAGCUAUGUCC   attributs
```

Avant d'énoncer les quatre problèmes d'alignement étudiés, nous avons besoin de définir les notions de sous-chaîne augmentée et super-chaîne augmentée.

Définition 3.3
Soient s et t deux chaînes augmentées d'ARN, de structures respectives S et T. La chaîne s est une super-chaîne augmentée de t s'il existe un alignement **sans mismatch** *de s sur la première bande et t sur la seconde bande tel que la première bande est sans gap et tel que les bases appariées de t sont alignées avec des bases appariées de s. Si s est une super-chaîne augmentée de t, alors t est une sous-chaîne augmentée de s.*

Exemple 3.12 *La première chaîne augmentée suivante est une super-chaîne augmentée de la deuxième.*

```
GCAGUCAUGAAUGCGCAAGCUAUGUCC
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
(((..(((..))).((..))..)))..   première chaîne
-(-..(-(..).).((.-))---)...   deuxième chaîne
 ↓ ↓↓↓ ↓↓↓↓↓↓↓↓↓↓ ↓↓   ↓↓↓↓
 C GUC UGAAUGCGCA GC   GUCC
```

Problème	**temps d'exécution**	**espace**
Alignement de deux chaînes augmentées	$O(m^2n^2)$	$O(m^2n^2)$
Plus longue sous-chaîne augmentée commune	$O(m^2n^2)$	$O(m^2n^2)$
Plus courte super-chaîne augmentée commune	$O\big(m^2n^2(m^2+n^2)\big)$	$O(m^2n^2)$
Alignement séquence contre chaîne augmentée	$O(m^2n^2+mn^3)$	$O(m^2n^2)$

Table 3.3: Résultats obtenus par Bafna *et al.*

Problème	**temps d'exécution**	**espace**
Alignement de deux chaînes augmentées	$O(m^2n^2)$	$O(m^2n^2)$
Plus longue sous-chaîne augmentée commune	$O(m^2n^2)$	$O(m^2n^2)$
Plus courte super-chaîne augmentée commune	$O(mn)$	$O(mn)$
Alignement séquence contre chaîne augmentée	$O(mn^3)$	$O(mn^2)$

Table 3.4: Résultats obtenus à l'aide des MTSAG.

Les quatre problèmes d'alignement étudiés par Bafna *et al.* sont :

1. alignement de deux chaînes augmentées avec pénalité linéaire de gaps,
2. plus longue sous-chaîne augmentée commune à deux chaînes augmentées,
3. plus courte super-chaîne augmentée commune à deux chaînes augmentées,
4. alignement d'une séquence d'ARN contre une chaîne augmentée.

Une solution sous forme d'équations de programmation dynamique est proposée dans (Bafna *et al.*, 1995) pour chacun de ces problèmes. Les complexités en espace de ces solutions ne sont pas données, mais la forme des équations montre immédiatement que toutes les solutions proposées ont une complexité en espace de $O(m^2n^2)$, où m et n sont les longueurs respectives de la première et deuxième chaîne augmentée. Les complexités des solutions proposées sont regroupées dans la table 3.3. L'application des MTSAG aux quatre problèmes précédents fournit des analyseurs syntaxiques dont les complexités sont regroupées dans la table 3.4.

En fait, le quatrième problème est une version simplifiée de celui traité par Corpet et Michot (cf. section précédente) : dans un cas comme dans l'autre, les appariements trouvés dans la séquence sont contraints à être présents dans la chaîne augmentée. Par contre, Bafna *et al.* ne considèrent que des pénalités linéaires de gap, alors que Corpet et Michot utilisent des pénalités affines. Les équations de programmation dynamique proposées par Corpet et Michot ont une complexité en temps de $O(m^2n^3)$, alors que celles proposées par Bafna ont une complexité en temps de $O(m^2n^2+mn^3)$ car ils constatent avec justesse que la structure particulière du problème est utilisable pour optimiser les calculs. Mais l'optimisation obtenue n'est pas aussi bonne qu'avec les MTSAG qui permettent d'obtenir une complexité de $O(mn^3)$.

Le problème d'alignement de deux chaînes augmentées avec pénalité linéaire de gap est défini de la manière suivante. Soient deux chaînes augmentées $s_1 \dots s_m$ et $t_1 \dots t_n$ de structures respectives S et T, et deux fonctions $\gamma(u, v)$ et $\delta(u, u', v, v')$ (dites de score d'alignement et de score d'alignement d'appariement). Il faut trouver un alignement $\left[\begin{smallmatrix} a_1 \dots a_k \\ b_1 \dots b_k \end{smallmatrix}\right]$ tel que $a_1 \dots a_k = s_1 \dots s_m$ et $b_1 \dots b_k = t_1 \dots t_n$ (i.e. les alignements peuvent contenir des gaps) qui maximise la relation suivante:

$$\sum_{1 \leqslant i \leqslant k} \gamma(a_i, b_i) + \sum_{1 \leqslant i < j \leqslant k} \delta(a_i, a_j, b_i, b_j)$$

Pour mériter son nom, la fonction δ doit retourner $-\infty$ si (a_i, a_j) ou (b_i, b_j) ne sont pas appariées. La fonction γ est chargée d'introduire les pénalités linéaires de gaps.

La recherche de la plus longue sous-chaîne augmentée commune à deux chaînes est un sous-problème du problème précédent puisqu'elle se ramène aux définitions suivantes de γ et δ.

$$\gamma(a_i, b_i) = \begin{cases} 1 & \text{si } a_i = b_i \\ 0 & \text{si } a_i = - \text{ ou } b_i = - \\ -\infty & \text{sinon} \end{cases}$$

$$\delta(a_i, a_j, b_i, b_j) = \begin{cases} 1 & \text{si } a_i = b_i \text{ et } a_j = b_j \text{ et } (a_i, a_j) \in S \text{ et } (b_i, b_j) \in T \\ -\infty & \text{sinon} \end{cases}$$

Ces deux problèmes sont aisément décrits par une MTSAG grâce au codage proposé en remarque 3.1. La MTSAG de la figure 3.15 pourra convenir à cet effet. Notons que les projections de cette MTSAG sur les deux bandes sont ambiguës, mais l'introduction de la contrainte de sélection permet de maintenir une complexité en temps de $O(m^2n^2)$, tandis que la complexité en espace reste égale à son maximum théorique de $O(m^2n^2)$.

Les cinq lignes suivantes sont un alignement de la plus longue sous-chaîne augmentée commune à deux chaînes augmentées et de ces deux chaînes augmentées. Les cinq lignes sont décomposées comme suit :

1. attributs de la première chaîne ;
2. structure de la première chaîne ;
3. structure de la sous-chaîne augmentée commune ;
4. structure de la deuxième chaîne ;
5. attributs de la deuxième chaîne.

```
GGGCGA  AUAG UGUCAGC  GGG AGCA CAC CA GAC UU GCAAU  CU GGUA GGG  AG GGUUCGAGUCC CUCU  UUGUCCACCA
((((((--...(-(((....--....-.)))-)((-((-((.-..-....)--))-))).-.((--((-((.......))-))))--))))))....
(((--(---..(-(--..---.--..-)-)---(--(-.-..-....---)--)--.-.----((-((.......))-))----)--)))....
(((--((((..(((--(..-....-..)-)))--.(-((-((((...))-)))-))--...--..(((((.......)))))-))))--)))....
GGG  AGUAUAGCU  CAG UUGG UAG AGC  GC UG CCUUUGCAA GGC AG  AUG  UCAGCGGUUCGAGUCCGCU UACU  UCCACCA
                                      ↑↑               ↑↑
                                      ij               kl
```

```
%{
#define INFINITY 10000
enum {IDX_A = 0, IDX_C, IDX_G, IDX_U,
        IDX_VAL};

#define MAX_NESTING 1000
int nesting_pos[MAX_NESTING];
int nesting = 0;
%}
%nb_tapes 2
%empty _
%attribute "int" idx pair score

%token '.'
{ $$.idx = yylex (); }

%token '('
{
  nesting_pos[nesting++] = tape_len;
  if (nesting == MAX_NESTING)
    {
      fprintf (stderr, "Max level reached\n");
      exit (1);
    }
  $$.idx = yylex ();
}

%token ')'
{
  $$.pair = nesting_pos[–nesting];
  tape[$$.pair].pair = tape_len;
  $$.idx = yylex ();
}

%abbreviation • _ '(' ')' '.'
```

```
%terminal_attribs
{ if ($$[1].len)
    if ($$[2].len)
      $$.score = γ ($$[1].left[0].idx, $$[2].left[0].idx);
    else
      $$.score = γ ($$[1].left[0].idx, -);
  else
      $$.score = γ (-, $$[2].left[0].idx);
}
%default_nonterminal_attribs <score>

%default_constraint
{ if ($$.score < -INFINITY / 2) return 0; }
{ if ($$.score > $! .score) $! .score = $$.score; }

%%
A   : [• •]
        { $$.score = $1.score; }
    | D
        { $$.score = $1.score; }
    | A [• •]
        { $$.score = $1.score + $2.score; }
    | A D
        { $$.score = $1.score + $2.score; } ;

D   : ['(' '('] A [')' ')']
        {
          if ($1[1].left[0].pair! = $3[1].left ||
              $1[2].left[0].pair! = $3[2].left)
            $$.score = -INFINITY;
          else
            $$.score = $1.score + $2.score + $3.score
                + δ ($1[1].left[0].idx, $3[1].left[0].idx,
                     $1[2].left[0].idx, $3[2].left[0].idx);
        } ;
%%
```

Figure 3.15: Lors de la lecture de chaque bande, il faut stocker la base sous-jacente dans l'attribut physique *idx* des tokens grâce à une valeur retournée par la fonction *yylex* de l'analyseur lexical (non représenté). De plus, chaque token '(' ou ')' contient dans l'attribut physique *pair* la position sur la bande de la parenthèse correspondante. Cette position est calculée au moment de la lecture des tokens grâce à la table *nestion_pos*. Aucune détection des bandes mal parenthésées n'est proposée dans la grammaire de cette figure. Ensuite, les pénalités de gap étant linéaires, elles peuvent être calculées une fois pour toute pour chaque terminal dans %terminal_attribs. La contrainte d'optimisation est bien entendu une contrainte de maximisation, tandis que la contrainte de sélection rejette toutes les sous-dérivations non-conformes, c'est-à-dire celles dont le score est trop bas (δ et γ sont censées renvoyer -INFINITY en cas d'alignement invalide).

En pratique, la plus longue sous-chaîne augmentée commune est obtenue en alignant deux chaînes augmentées grâce à la MTSAG de la figure 3.15 et en remplaçant chaque colonne par

- un `'-'` si la colonne contient au moins un `'-'`,
- une `'('` si la colonne contient uniquement des `'('` et si les `')'` correspondantes sont alignées,
- une `')'` si la colonne contient uniquement des `')'` et si les `'('` correspondantes sont alignées,
- un `'.'` sinon.

Si la procédure habituellement utilisée pour les séquences normales était généralisable, alors la plus courte super-chaîne augmentée commune à deux chaîne augmentées s'obtiendrait à partir du même alignement en remplaçant cette fois chaque colonne par :

- une `'('` si la colonne contient au moins une `'('` ;
- un `'.'` sinon.

L'attribut de chaque symbole est la base commune aux symboles de la colonne s'il n'y a pas de gap, ou la base d'une des deux chaînes si l'autre chaîne contient un gap.

Malheureusement, et comme le font remarquer Bafna *et al.*, il n'y a aucune raison pour que les appariements obtenus ne contiennent pas de pseudo-nœud. Ainsi, dans l'exemple précédent, il existe 4 colonnes $i < j < k < l$ telles que les parenthèses des colonnes i et k sont appariées dans la première chaîne, tandis que celles des colonnes j et l sont appariées dans la deuxième chaîne. Il faut donc revoir la méthode de calcul de l'alignement pour l'adapter à la recherche de la plus courte super-chaîne augmentée commune. Pour résoudre ce problème, Bafna *et al.* proposent une série d'équations de programmation dynamique dont la complexité en temps est $O\big(m^2n^2(m^2+n^2)\big)$. Cette complexité est très loin d'être optimale puisque la MTSAG de la figure 3.16 résout ce problème en temps et espace $O(mn)$.

Théorème 3.1 (Plus courte super-chaîne augmentée commune)
Soient $s_1 \dots s_m$ et $t_1 \dots t_n$ deux chaînes augmentées de structures respectives S et T. Il existe un algorithme de complexité $O(mn)$ en temps et $O(mn)$ en espace pour trouver la plus courte super-chaîne augmentée commune à ces deux chaînes.

Démonstration : l'existence d'une super-chaîne augmentée commune résulte du fait que la concaténation de $s_1 \dots s_m$ et $t_1 \dots t_n$ fournit trivialement une super-chaîne augmentée commune.

La grammaire décrite figure 3.16 fournit effectivement un algorithme de recherche de la plus courte super-chaîne augmentée commune car toute dérivation du symbole de départ de score fini fournit obligatoirement un alignement d'où une super-chaîne augmentée commune est déductible. La contrainte d'optimisation assure que la super-chaîne trouvée est bien la plus courte.

Les complexités en temps et espace de $O(mn)$ se déduisent trivialement du fait que les grammaires projetées sont LR(1) et non-récursives à droite. □

La MTSAG construite se contente pourtant de traduire la méthode de construction d'une super-chaîne indiquée plus haut et de respecter la contrainte d'absence de pseudo-nœud. Ce résultat montre à quel point il est parfois délicat d'établir des équations de programmation dynamique efficaces en présence d'un modèle ayant une structure non triviale, alors que l'écriture d'une grammaire correspondante ne se heurte pas aux mêmes difficultés.

Les cinq lignes suivantes sont un alignement des mêmes chaînes augmentées que dans l'exemple précédent et de leur plus courte super-chaîne augmentée commune. La super-chaîne se trouve sur la troisième ligne.

```
GGGCGA  AUAG UGUCAGC  GGG A GCA C   A CCAGA CUUGCAAUCUGG UAGG  G  AG GGUUCGAGUCC CUCU   UUGUCCACCA
(((((--...(-(((...--...-.-)))-)---(-(((((-.......)))))-)..(--(--((-((.......))-))))---))))))....
((((((((..((((((.........)))))).(((((((((.(...))))))))))(..(..(((((......)))))))))))))))....
(((--((((..(((--(..-....-..)--))).((-(((---(-(...))---)))-))-.....(((((.......)))))--))))--)))....
GGG  AGUAUAGCU  CAG UUGG UAG  AGCGCU GCC   U UUGCAA   GGC AG AUGUCAGCGGUUCGAGUCCGCU  UACU  UCCACCA
```

On pourra comparer cet alignement à celui donnant la plus longue sous-chaîne augmentée, que nous rappelons ici pour mémoire.

```
GGGCGA  AUAG UGUCAGC  GGG AGCA CAC CA GAC UU GCAAU  CU GGUA GGG  AG GGUUCGAGUCC CUCU  UUGUCCACCA
(((((--...(-(((....--....-.)))-)((-((-((.-..-.....)--))-))).-.((--((-((.......))-))))--))))))....
(((--(---..(-(--...---.--..-)-)---(--(-.-..-.......---)--)--.-.----((-((.......))-))----)--)))....
(((--((((..(((--(..-.....-..)-)))--.(-((-((((...))-)))-))--...--..(((((.......)))))-))))--)))....
GGG  AGUAUAGCU  CAG UUGG UAG AGC  GC UG CCUUUGCAA GGC AG  AUG  UCAGCGGUUCGAGUCCGCU UACU  UCCACCA
```

Dans les deux cas, on remarquera que les alignements de chaînes augmentées ne semblent pas fournir des solutions biologiquement agréables au problème d'alignement de structures secondaires d'ARN. Ce n'était d'ailleurs pas le but de Bafna *et al.*

3.7 Alignement d'arbres de dérivation

Nous avons affirmé dans la section 3.3.1 p. 96 que l'alignement d'un ensemble de structures secondaires obtenues à l'aide d'un algorithme de repliement a priori n'est pas suffisant pour déterminer la structure consensus de cet ensemble, en raison des erreurs contenues dans les structures initiales. Nous allons montrer dans cette section que cette opinion n'est pas dictée par une faiblesse des MTSAG en ce qui concerne les modèles de comparaison ou d'alignement d'arbres de structures secondaires. Nous illustrerons notre propos en appliquant les MTSAG à deux modèles déjà décrits dans la littérature (Shapiro & Zhang, 1990; Jiang *et al.*, 1994). Nous verrons que les MTSAG produites sont plus claires et produisent des algorithmes plus performants que ceux des auteurs de ces modèles.

Le premier modèle est celui proposé par Shapiro et Zhang en 1990 (Shapiro & Zhang, 1990). Ce modèle repose sur la transformation d'une structure secondaire en arbre ordonné (i.e. l'ordre des fils d'un sommet est significatif), chaque sommet de l'arbre codant une boucle de la structure secondaire (figure 3.17). Un parcours préfixe de l'arbre de la figure 3.17 fournit *l'expression préfixe parenthésée* suivante :

$$(N(R(I(R(M(R(B(R(M(R(H))(R(H))))))(R(H)))))))$$

```
%{
#define INFINITY 10000
enum {IDX_A = 0, IDX_C, IDX_G, IDX_U,
        IDX_VAL};

#define MAX_NESTING 1000
int nesting_pos[MAX_NESTING];
int nesting = 0;
%}
%nb_tapes 2
%empty _
%attribute "int" idx pair score

%token '.'
{ $$.idx = yylex (); }

%token '('
{
  nesting_pos[nesting++] = tape_len;
  if (nesting == MAX_NESTING)
     {
       fprintf (stderr, "Max level reached\n");
       exit (1);
     }
  $$.idx = yylex ();
}

%token ')'
{
  $$.pair = nesting_pos[--nesting];
  tape[$$.pair].pair = tape_len;
  $$.idx = yylex ();
}

%abbreviation • _ '.'
%terminal_attribs
{ if ($$[1].len && $$[2].len)
    if ($$[1].left[0].idx==$$[2].left[0].idx) $$.score=1;
    else $$.score = INFINITY;
  else $$.score = 1;
}
```

```
%default_nonterminal_attribs <score>

%default_constraint
{ if ($$.score > INFINITY / 2) return 0; }
{ if ($$.score < $! .score) $! .score = $$.score; }

%%
A  : [• •]
        { $$.score = $1.score; }
   | D
        { $$.score = $1.score; }
   | A [• •]
        { $$.score = $1.score + $2.score; }
   | A D
        { $$.score = $1.score + $2.score; } ;

D  : ['(' '('] A [')' ')']
        {
          if ($1[1].left[0].pair! = $3[1].left ||
              $1[2].left[0].pair! = $3[2].left)
            $$.score = INFINITY;
          else
            $$.score=$1.score+$2.score+$3.score+1;
        }
   | ['(' •] A [')' •]
         {
           if ($1[1].left[0].pair! = $3[1].left)
             $$.score = INFINITY;
           else
             $$.score=$1.score+$2.score+$3.score+1;
         }
   | [• '('] A [• ')']
        {
          if ($1[2].left[0].pair! = $3[2].left)
            $$.score = INFINITY;
          else
            $$.score=$1.score+$2.score+$3.score+1;
        } ;
%%
```

Figure 3.16: L'en-tête de cette MTSAG est très semblable à celui de la figure 3.15. Les changements les plus importants proviennent de la redéfinition de l'abréviation • et de la contrainte d'optimisation, qui est cette fois une contrainte de minimisation. Les fonctions γ et δ ont été remplacées par les constantes appropriées.

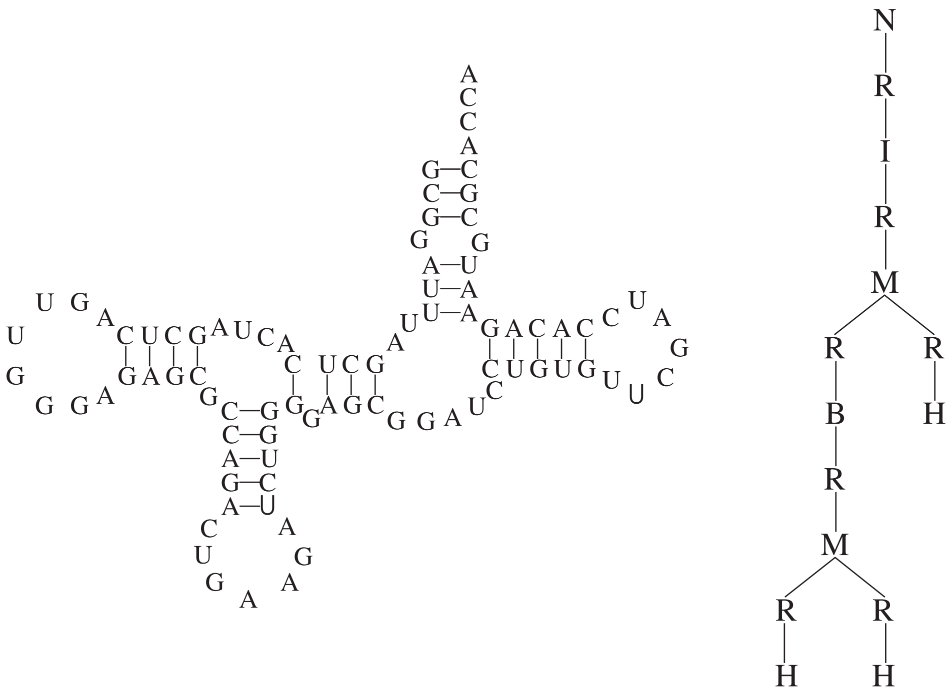


Figure 3.17: Représentation en arbre (à droite) d'une structure secondaire (à gauche). Les épingles à cheveux sont représentées par un H, les boucles intérieures par un I, les boucles proéminentes par un B, les boucles multiples par un M, les empilements par un R, la boucle extérieure par un N.

Shapiro et Zhang définissent en premier lieu une forme restreinte de la notion d'édition d'arbre ordonné à partir de trois opérations élémentaires : le réétiquetage d'un sommet, la délétion d'un sommet, l'insertion d'un sommet. Un coût est attribué à chacune de ces opérations élémentaires, le coût d'une séquence d'éditions élémentaires étant la somme des coûts élémentaires. La distance entre deux arbres T_1 et T_2 est alors définie comme étant le coût minimal d'une séquence d'édition transformant T_1 en T_2.

La notion de "mapping" d'un arbre vers un autre est ensuite définie (cf. figure 3.18).

Définition 3.4 (mapping)
Soient T_1 et T_2 deux arbres de structures secondaires. Un mapping de T_1 vers T_2 est un triplet (M, T_1, T_2), où M est un ensemble de couples (i, j) satisfaisant les conditions suivantes :

1. *i est un sommet de T_1 et j est un sommet de T_2.*
2. *Pour chaque paire de couples $(i_1, j_1) \in M$ et $(i_2, j_2) \in M$:*
   (a) *$i_1 = i_2$ si et seulement si $j_1 = j_2$;*
   (b) *i_1 est un ancètre de i_2 si et seulement si j_1 est un ancêtre de j_2 ;*
   (c) *i_1 est à gauche de i_2 (i.e. i_1 et i_2 ont le même parent et i_1 précède i_2, ce qui est traditionnellement représenté graphiquement en notant i_1 à gauche de i_2) si et seulement si j_1 est à gauche de j_2 ;*

Le coût d'un mapping est défini comme étant la somme des coûts :

1. $\gamma(i \to j)$ pour les sommets i et j tels que $(i, j) \in M$ (sommets édités),
2. $\gamma(i \to \epsilon)$ pour les sommets i tels qu'il n'existe pas j tel que $(i, j) \in M$ (sommets supprimés),
3. $\gamma(\epsilon \to j)$ pour les sommets j tels qu'il n'existe pas i tel que $(i, j) \in M$ (sommets insérés).

Enfin, Shapiro et Zhang montrent que la distance entre deux arbres T_1 et T_2 est égale au coût d'un mapping de coût minimal entre deux arbres, puis ils proposent un algorithme de programmation dynamique permettant de trouver un mapping de coût minimal en temps $O(|T_1| \times |T_2| \times L_1 \times L_2)$ et en espace $O(|T_1| \times |T_2|)$, où L_1 est la profondeur de T_1 et L_2 la profondeur de T_2.

Pour obtenir une MTSAG adaptée à ce modèle, remarquons que les expressions préfixes parenthésées représentant les arbres de structure sont décrites par la grammaire LR(1) non-récursive à droite suivante :

$$\begin{aligned} S &\to (\ N\ S'\ ) \mid (\ I\ S'\ ) \mid (\ B\ S'\ ) \mid (\ R\ S'\ ) \mid (\ M\ S'\ ) \mid (\ H\ ) \\ S' &\to S \mid S'\ S \end{aligned}$$

La définition 3.4 signifie en fait que les expressions préfixes parenthésées sont alignées en conservant la structure définie par les parenthèses (l'alignement de deux expressions

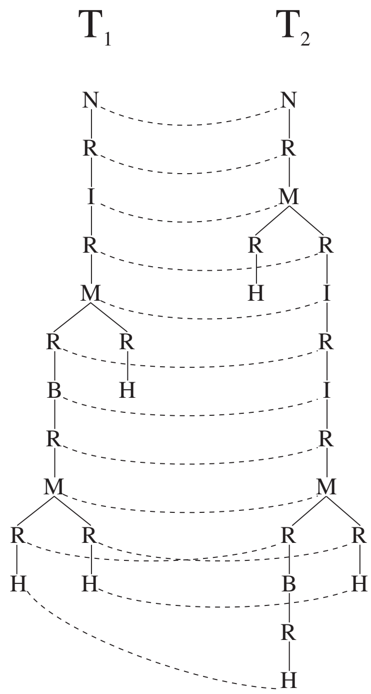


Figure 3.18: Illustration d'un mapping (optimal) entre deux arbres. Les sommets touchés par un arc sont des sommets édités. Les sommets de T_1 non touchés par un arc sont des sommets supprimés et les sommets de T_2 non touchés par un arc sont des sommets insérés.

```
%{
#define INF 1000000

int edit[6][6] = {
        /*   NI,     B,    H,    M,    R */
     /* N */{ 0,    INF, INF, INF, INF, INF},
     /* I */ { INF, 0,     3,    8,    8,    INF},
     /* B */{ INF, 3,     0,    8,    8,    INF},
     /* H */{ INF, 8,     8,    0,    8,    INF},
     /* M */{ INF, 8,     8,    8,    0,    INF},
     /* R */{ INF, INF, INF, INF, INF, 0}};

int delete[6] = {INF, 5, 5,100, 75, 5};
int insert[6] = {INF, 5, 5,100, 75, 5};
%}

%nb_tapes 2

%empty _
%attribute "int" idx
%token N { $$.idx = 0; }
%token I  { $$.idx = 1; }
%token B { $$.idx = 2; }
%token H { $$.idx = 3; }
%token M { $$.idx = 4; }
%token R { $$.idx = 5; }
%token '(' ')'

%abbreviation • N I B H M R
```

```
%attribute "int" edit
%terminal_attribs {}
%default_nonterminal_attribs <edit>

%default_constraint
        { if ($$.edit > INF / 2) return 0; }
        { if ($$.edit < $! .edit) $! .edit = $$.edit; }
%%

tree : ['(' '('] [• •] sequence [')' ')']
       { $$.edit = $3.edit + edit[$2[1].left[0].idx]
                                  [$2[2].left[0].idx]; }
     | [_ '('] [_ •] sequence [_ ')']
       { $$.edit = $3.edit + insert[$2[2].left[0].idx]; }
     | ['(' _] [• _] sequence [')' _]
       { $$.edit = $3.edit + delete[$2[1].left[0].idx]; }
     | ['(' '('] [H H] [')' ')']
       { $$.edit = edit[$2[1].left[0].idx]
                       [$2[2].left[0].idx]; }
     | ['(' _] [H _] [')' _]
       { $$.edit = delete[$2[1].left[0].idx]; }
     | [_ '('] [_ H] [_ ')']
       { $$.edit = insert[$2[2].left[0].idx]; } ;

sequence : tree            { $$.edit = $1.edit; }
         | sequence tree   { $$.edit = $1.edit +
                                       $2.edit; } ;
%%
```

Figure 3.19: MTSAG réalisant l'alignement optimal de deux arbres de structures ou, de manière équivalente, la recherche d'un mapping optimal des deux structures. Les valeurs utilisées pour la fonction γ dans les tables edit, delete et insert sont les mêmes que celles utilisées dans (Shapiro & Zhang, 1990)

correspond en quelque sorte à une super-chaîne augmentée commune à deux chaînes augmentées (cf. définition 3.3 p. 122)). Il suffit donc de transformer les règles précédentes en règles 2-bandes codant les différentes combinaisons d'édition, d'insertion et de délétion. La MTSAG décrivant ces alignements ainsi que les fonctions d'évaluation d'attributs associées aux coûts γ est représentée figure 3.19.

Cette MTSAG ayant des projections trivialement réductibles en grammaires LR(1) non-récursives à droite, l'analyse syntaxique est réalisée en temps et en espace $O(|T_1| \times |T_2|)$, soit un gain d'un facteur $L_1 \times L_2$ en temps. L'alignement optimal des deux arbres dont le mapping optimal est représentée en figure 3.18 est le suivant :

```
(N(R(I------(R(M(R(B(R(M(R----(H)--)(R(H))))))(R(H)))))))
(N(R(M(R(H))(R(I(R(I(R(M(R(B(R(H))))(R(H))))))------)))))
```

Le second modèle est celui proposé par Jiang *et al.* en 1994 (Jiang *et al.*, 1994). Dans ce modèle, la notion de mapping entre arbres disparaît pour laisser la place à la notion d'alignement d'arbres. Lorsque les arbres sont ordonnés, l'algorithme proposé par les auteurs, et rappelé figure 3.20, fournit un alignement optimal de deux arbres ordonnés en temps $O(|T_1| \times |T_2| \times D_1 \times D_2)$, où D_1 et D_2 sont respectivement les degrés de T_1

et T_2 (le degré d'un arbre est le nombre maximum de fils de tout sommet de l'arbre). Les alignements fournis par cet algorithme sont équivalents aux mappings optimaux fournies par l'algorithme de (Shapiro & Zhang, 1990), bien que ce fait ne soit pas mentionné dans (Jiang *et al.*, 1994). L'intérêt principal de l'algorithme d'alignement est d'être plus performant sur les arbres ordonnés que l'algorithme de recherche de mapping optimal, et d'être généralisable à des arbres non ordonnés (Zhang, 1993; Jiang *et al.*, 1994).

Comme la MTSAG de la figure 3.19 utilisait déjà les alignements pour rechercher des mappings optimaux, elle peut être employée quasiment telle quelle à la recherche d'alignements d'arbres, à condition de coder ces arbres sous la forme de chaînes parenthésées obtenues par parcours préfixe. La MTSAG est également sensiblement plus facile à comprendre que les équations de programmation dynamique de la figure 3.20 et elle conserve une meilleure complexité en temps. Par contre, la MTSAG ne peut pas être employée pour aligner des arbres non ordonnés, contrairement aux récurrences de la figure 3.20 qui peuvent être généralisées.

3.8 Modèles de Markov cachés (Hidden Markov Models)

Les années 1994–1996 ont été marquées par l'explosion du formalisme des modèles de Markov cachés (ou Hidden Markov Models, ou HMM) en bio-informatique. Les raisons de l'engouement pour ce formalisme sont multiples. En effet, les HMM sont aussi bien utilisables pour la détection de motifs (Fujiwara *et al.*, 1994) que pour l'alignement multiple de séquences (Krogh *et al.*, 1994a; Baldi *et al.*, 1994; Eddy, 1995) ou la recherche de séquences dans les bases de données (Krogh *et al.*, 1994b; Lowe & Eddy, 1996). En ce qui concerne les alignements multiples, les HMM ont plusieurs avantages par rapport aux méthodes d'alignement multiple classiques [11] :

- ils sont fondés sur une théorie probabiliste solide,
- la connaissance de la phylogénie des séquences à aligner n'est pas absolument nécessaire,
- les insertions et délétions sont traitées avec un coût probabiliste variable, fonction de la position,
- les informations expérimentales peuvent être incorporées au modèle a priori,
- le coût d'alignement d'un ensemble de séquences à un HMM est proportionnel au nombre de séquences à aligner, alors qu'il est à croissance exponentielle dans la plupart des autres méthodes existantes.

Nous allons tout d'abord rappeler le principe de fonctionnement des HMM, puis nous proposerons une implémentation sous la forme d'une MTSAG 1-bande plutôt que

[11] Nous renvoyons le lecteur aux références données en section 3.1 p. 87 pour l'alignement multiple et (Eddy, 1996) pour un état de l'art concernant les HMM utilisés en bio-informatique.

Soient T_1 et T_2 deux arbres ordonnés. Soient i un sommet de T_1 et j un sommet de T_2. Les degrés de i et j sont notés m_i et n_j respectivement. Les fils de i et j sont notés $i_1, \ldots, i_{m_i}$, et $j_1, \ldots, j_{n_j}$ respectivement. Le sous-arbre de T_1 (resp. T_2) de sommet i (resp. j) est noté $T_1[i]$ (resp. $T_2[j]$). L'arbre vide est noté θ. Soit $F_1[i_s, i_t]$ la forêt consistant en les sous-arbres $T_1[i_s], \ldots, T_1[i_t]$. La forêt $F_1[i_1, i_{m_i}]$ est également notée $F_1[i]$. $F_2[j_s, j_t]$ et $F_2[j]$ sont définis de manière similaire. Le coût d'alignement des sommets i et j est noté $\gamma(i \rightarrow j)$, comme dans la définition 3.4 p. 130. La distance d'alignement de deux sous-arbres est notée $D(T_1[i], T_2[j])$ et la distance d'alignement de deux forêts F_1 et F_2 est notée $D(F_1, F_2)$. Les équations de programmation dynamique suivantes permettent de calculer $D(T_1, T_2)$.

$$
\begin{aligned}
D(\theta, \theta) &= 0 \\
D(F_1[i], \theta) &= \sum_{k=1}^{m_i} D(T_1[i_k], \theta) \\
D(T_1[i], \theta) &= D(F_1[i], \theta) + \gamma(i \rightarrow \epsilon) \\
D(\theta, F_2[j]) &= \sum_{k=1}^{n_j} D(\theta, T_2[j_k]) \\
D(\theta, T_2[j]) &= D(\theta, F_2[j]) + \gamma(\theta \rightarrow j) \\
D(T_1[i], T_2[j]) &= \min \begin{cases} D(\theta, T_2[j]) + \min_{1 \leqslant r \leqslant n_j} \big(D(T_1[i], T_2[j_r]) - D(\theta, T_2[j_r])\big) \\ D(T_1[i], \theta) + \min_{1 \leqslant r \leqslant m_i} \big(D(T_1[i_r], T_2[j]) - D(T_1[i_r], \theta)\big) \\ D(F_1[i], F_2[j]) + \gamma(i \rightarrow j) \end{cases}
\end{aligned}
$$

et pour tous s et t tels que $1 \leqslant s \leqslant m_i$ et $1 \leqslant t \leqslant n_j$

$$
D(F_1[i_1, i_s], F_2[j_1, j_t]) = \min \begin{cases} D(F_1[i_1, i_{s-1}], F_2[j_1, j_t]) + D(T_1[i_s], \theta) \\ D(F_1[i_1, i_s], F_2[j_1, j_{t-1}]) + D(\theta, T_2[j_t]) \\ D(F_1[i_1, i_{s-1}], F_2[j_1, j_{t-1}]) + D(T_1[i_s], T_2[j_t]) \\ \gamma(\epsilon \rightarrow j_t) + \min_{1 \leqslant k < s} \big(D(F_1[i_1, i_{k-1}], F_2[j_1, j_{t-1}]) + \\ \qquad D(F_1[i_k, i_s], F_2[j_t])\big) \\ \gamma(i_s \rightarrow \epsilon) + \min_{1 \leqslant k < t} \big(D(F_1[i_1, i_{s-1}], F_2[j_1, j_{k-1}]) + \\ \qquad D(F_1[i_s], F_2[j_k, j_t])\big) \end{cases}
$$

Figure 3.20: Algorithme de programation dynamique utilisé par Jiang *et al.* pour aligner deux arbres ordonnés.

sous la forme habituelle d'équations de programmation dynamique. Finalement, nous montrerons que toute MTSAG 1-bande représentant un HMM peut être factorisée en une MTSAG 2-bandes indépendante du HMM. Ce type de factorisation se révèlera particulièrement utile lors de la généralisation des HMM aux grammaires stochastiques (section suivante).

Les HMM ne forment en fait qu'un sous-ensemble des automates finis non-déterministes stochastiques, c'est-à-dire des automates finis non-déterministes dont chaque transition s'est vue affecter une probabilité en plus du symbole d'étiquetage habituel. Ces probabilités vont servir à calculer la probabilité des chemins empruntés par l'automate. La particularité des HMM utilisés pour l'alignement de séquences biologiques provient de leur distribution spécifique d'états. En effet, les états de tels HMM peuvent être regroupés en m colonnes (correspondant aux colonnes consensus d'un alignement multiple), chaque colonne i ($1 \leqslant i \leqslant m$) contenant trois états de types différents.

- Un état d'alignement, ou matching state M_i. Cet état a toutes ses transitions vers les trois états de la colonne suivante et une transition ne peut être empruntée qu'en lisant un symbole de la bande d'entrée (i.e. les transitions ne sont pas des ϵ-transitions). Le passage par un tel état correspond à la présence d'une lettre dans la colonne consensus i d'un alignement multiple.

- Un état de délétion D_i. Cet état a lui aussi toutes ses transitions vers les trois états de la colonne suivante et une transition empruntée ne lit aucun symbole de la bande d'entrée (i.e. les transitions sont obligatoirement des ϵ-transitions). Le passage par un tel état correspond à la présence d'un gap dans une colonne consensus i d'un alignement multiple.

- Un état d'insertion I_i. Cet état a toutes ses transitions vers les trois états de sa colonne, lui même compris. Les transitions ne sont pas des ϵ-transitions. Les passages par un tel état correspondent à l'insertion de lettres avant la colonne consensus i d'un alignement multiple.

En plus des états des m colonnes précédentes, il faut ajouter une colonne 0 contenant uniquement un état de départ S effectuant des ϵ-transitions vers les états de la première colonne, et une colonne $m+1$ contenant uniquement un état d'insertion I_{m+1} et un état de fin E. La figure 3.21 est une synthèse des définitions précédentes. Dans la suite, nous noterons q_i^c (avec $c = 1, 2, 3$) un état de la colonne i, avec $q_i^1 = M_i$, $q_i^2 = D_i$ et $q_i^3 = I_i$. Par convention, $q_0^2 = S$, $q_{m+1}^2 = E$ et $q_{m+1}^3 = I_{m+1}$.

Dans un état q_i^c donné, la probabilité de passer à l'état $q_{i'}^{c'}$ en lisant x (où x est un symbole ou ϵ) est $P(q_{i'}^{c'}, x \mid q_i^c)$. L'alphabet Σ des symboles est celui de la séquence analysée, c'est-à-dire celui des acides aminés pour les protéines (20 éléments) ou celui des acides nucléiques pour les ADN (4 éléments). En réalité, la probabilité $P(q_{i'}^{c'}, x \mid q_i^c)$ est décomposée en un produit de deux probabilités :

$$P(q_{i'}^{c'}, x \mid q_i^c) = \underbrace{P(q_{i'}^{c'} \mid q_i^c)}_{\text{transition}} \underbrace{P(x \mid q_i^c)}_{\text{émission}}$$

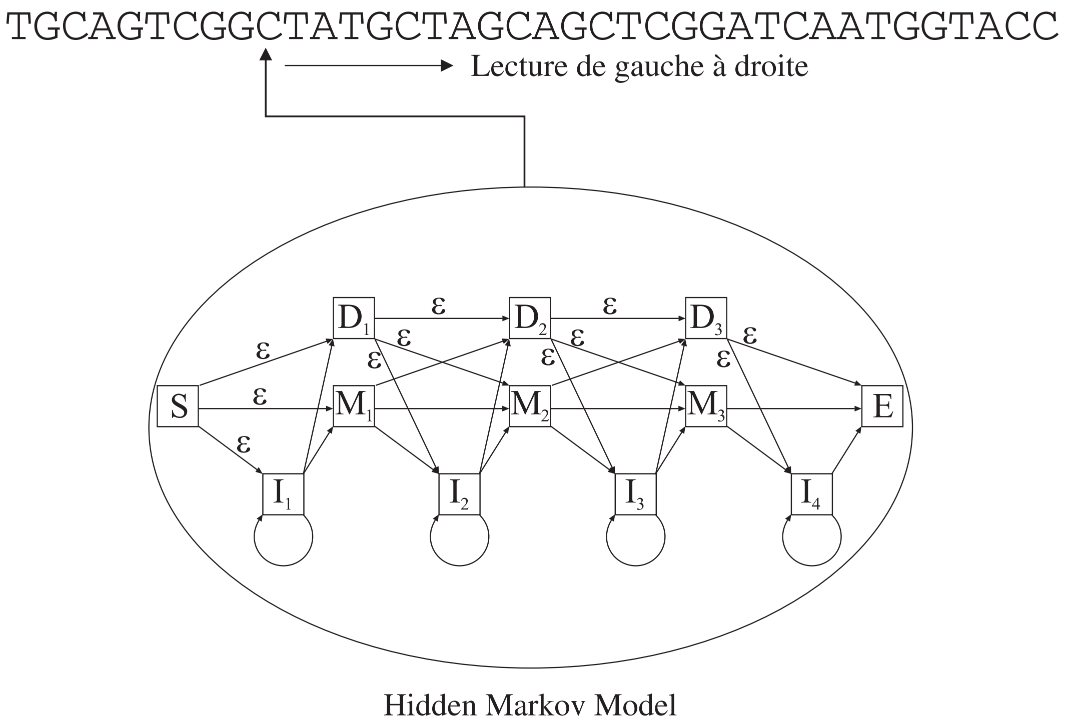


Figure 3.21: Les transitions du HMM ont reçu un label ϵ uniquement lorsqu'il s'agit d'ϵ-transitions. Les autres transitions sont en fait un ensemble de $|\Sigma|$ transitions, où $|\Sigma|$ est la taille de l'alphabet utilisé pour décrire la séquence d'entrée (une transition par symbole possible). Une probabilité (non représentée) est assignée à chaque transition, que ce soit une ϵ-transition ou non.

Au total, $|\Sigma|+1$ probabilités $P(x \mid q_i^c)$, dites d'émission de symbole, et une probabilité $P(q_{i'}^{c'} \mid q_i^c)$, dite de transition, sont donc attribuées à chaque état. Ces probabilités vérifient bien entendu les relations de normalisation suivantes :

$$\sum_{q_{i'}^{c'} \text{ destination de } q_i^c} P(q_{i'}^{c'} \mid q_i^c) = 1 \qquad\qquad \sum_{x \in \Sigma \cup \{\epsilon\}} P(x \mid q_i^c) = 1$$

Par définition des différents états, $P(\epsilon \mid q_i^2) = P(\epsilon \mid D_i) = 1$, $P(\epsilon \mid q_i^1) = P(\epsilon \mid M_i) = 0$ et $P(\epsilon \mid q_i^3) = P(\epsilon \mid I_i) = 0$.

Étant donnés une séquence et un HMM, la suite d'états partant de l'état initial S et arrivant à l'état final E en ayant lu la totalité de la séquence forme une chaîne de Markov acceptante. Le nom Hidden Markov Model provient du fait que, en réalité, seule la séquence biologique est connue, pas l'automate. C'est tout le problème car la présence d'états d'insertion dans un HMM permet à n'importe quel HMM d'accepter n'importe quelle séquence (i.e. il existe toujours une chaîne acceptante). Toujours est-il que certains HMM sont jugés meilleurs que d'autres pour décrire une famille donnée de séquences. Quels sont donc les critères permettant d'estimer la qualité d'un HMM ?

Soit $a_1 \ldots a_n$ une séquence, $q_{i_0}^{c_0} \ldots q_{i_k}^{c_k}$ une chaîne acceptante d'un HMM pour cette séquence et $x_1, \ldots, x_k$ la suite correspondante de symboles lus[12]. Certains x_i peuvent être égaux à ϵ mais l'ensemble des x_i vérifie bien sûr $a_1 \ldots a_n = x_1 \ldots x_k$ et $k \geqslant n$. La probabilité de la séquence selon cette chaîne acceptante est le produit des probabilités des transitions empruntées :

$$P(a_1 \ldots a_n \mid q_{i_0}^{c_0} \ldots q_{i_k}^{c_k}) = \prod_{j=1}^{k} P(q_{i_j}^{c_j}, x_j \mid q_{i_{j-1}}^{c_{j-1}}) = \prod_{j=1}^{k} P(q_{i_j}^{c_j} \mid q_{i_{j-1}}^{c_{j-1}}) P(x_j \mid q_{i_{j-1}}^{c_{j-1}})$$

La probabilité d'une séquence est la somme des probabilités de la séquence selon les différentes chaînes acceptantes possibles :

$$P(a_1 \ldots a_n) = \sum_{\substack{q_{i_0}^{c_0} \ldots q_{i_k}^{c_k} \text{ est une chaîne} \\ \text{acceptante de } a_1 \ldots a_n}} P(a_1 \ldots a_n \mid q_{i_0}^{c_0} \ldots q_{i_k}^{c_k}) \tag{3.1}$$

Pour une famille donnée de séquences, un bon HMM est un HMM qui maximise la probabilité de chaque séquence de la famille et minimise celle des séquences n'appartenant pas à la famille. Ainsi, les séquences n'appartenant pas à la famille sont susceptibles d'être rejetées par simple seuillage des probabilités de séquence. L'apprentissage d'un bon HMM est un problème délicat car le nombre important de degrés de liberté peut conduire l'algorithme d'apprentissage à faire converger un HMM vers un modèle sans capacité de généralisation car trop paramétré pour l'ensemble d'apprentissage (overfit) ou vers un modèle non optimal (optimum local). Le choix d'un ensemble d'apprentissage est également délicat. Nous ne donnerons pas ici de bibliographie complète des différents algorithmes d'apprentissage habituellement utilisés mais le lecteur intéressé pourra consulter la bibliographie de (McClure *et al.*, 1996) ou (Eddy, 1996).

Partant d'un bon HMM, l'alignement des séquences est obtenu en cherchant pour chaque séquence la chaîne acceptante la plus probable et en alignant les chaînes acceptantes obtenues colonne par colonne (d'après les colonnes du HMM d'origine). La recherche de séquences dans les bases de données est réalisée en choisissant un seuil de probabilité de séquence appartenant au modèle et en retenant uniquement les séquences de la base dont les probabilités dépassent ce seuil.

Jusqu'à présent, nous avons expliqué à quoi servent $P(a_1 \ldots a_n)$ ou les chaînes acceptantes les plus probables, sans expliquer comment ces deux quantités sont calculées efficacement. Il est aisé de deviner que la programmation dynamique est habituellement mise à contribution à cet effet. Plutôt que d'écrire des relations de programmation dynamique évidentes, nous donnons dans la figure 3.22 un exemple de MTSAG 1-bande pour le HMM de la figure 3.21. Cette MTSAG permet de calculer simultanément la probabilité d'une chaîne et la chaîne acceptante la plus probable, simplement à partir

[12]Étant donnés $a_1 \ldots a_n$ et $q_{i_0}^{c_0} \ldots q_{i_k}^{c_k}$, il est toujours possible de retrouver $x_1 \ldots x_k$ car tout état q_i^c doit soit lire obligatoirement un symbole (état M_i ou I_i), soit ne rien lire (état D_i). La contrainte $a_1 \ldots a_n = x_1 \ldots x_k$ suffit alors à lever les ambiguïtés de transition.

de la description des états et des probabilités de l'automate. L'analyseur syntaxique généré à partir de cette grammaire calcule l'arbre de dérivation le plus probable (i.e. la chaîne acceptante la plus probable). L'attribut physique `p` à la racine de cet arbre contient la probabilité de la dérivation la plus probable, tandis que l'attribut physique `sum` contient la probabilité $P(a_1 \ldots a_n)$ de la séquence de départ. La MTSAG 1-bande décrite ici est facilement généralisable à des HMM plus longs.

La grammaire G d'un HMM étant linéaire à gauche, les complexités d'analyse sont $O\big(|G|n\big)$ en temps et en espace pour une séquence de taille n. Cet ordre de grandeur est comparable au $O\big(|HMM|n\big)$ habituel puisque la taille de la grammaire est proportionnelle à la taille du HMM correspondant. Cette proportionnalité est d'ailleurs déplaisante car la syntaxe d'une grammaire utilisable par MTSAG2C n'est pas particulièrement concise. De plus, tout changement dans le nombre de colonnes du HMM impose un changement de grammaire et une recompilation de l'analyseur syntaxique généré. Pour supprimer ces deux inconvénients, nous allons factoriser les HMM grâce à l'introduction d'une bande de lecture supplémentaire.

Nous avons déjà mentionné la décomposition en colonnes des états d'un HMM. D'une colonne à l'autre, les transitions ont une configuration fixe et seules leurs probabilités changent. Nous pouvons donc remplacer chacune des colonnes par un symbole unique contenant les probabilités des transitions sous forme codée (figure 3.23), la séquence de ces symboles formant une première bande (figure 3.24). La configuration des transitions est remplacée par un automate à deux bandes unique (figure 3.25) qui reconstitue les probabilités de transition grâce à la première bande et la séquence à analyser grâce à la deuxième bande. Le changement de colonne dans un HMM correspond à la lecture d'un caractère sur la première bande. Finalement, l'automate à deux bandes est transformé en MTSAG 2-bandes (figure 3.26). Cette MTSAG ayant des projections linéaires à gauche sur les deux bandes, les complexités d'analyse en temps et en espace sont $O(mn)$ où m, la longueur de la première bande, est égale au nombre de colonnes du HMM de départ. Les complexités théoriques restent donc les mêmes, que ce soit en utilisant des équations de programmation dynamique, une MTSAG 1-bande, ou une MTSAG 2-bandes.

Comme nous pouvons le constater, la mise à jour d'un HMM avec une MTSAG 2-bandes fixe est beaucoup plus agréable qu'avec une MTSAG 1-bande, puisqu'elle se réduit à la mise à jour d'une première bande. En pratique, et dans le cadre des HMM que nous avons présentés dans cette section, les temps de calculs seront bien plus élevés avec l'analyseur généré à partir de la MTSAG 2-bandes ou de la MTSAG 1-bande qu'avec un programme spécialement écrit à partir des équations de programmation dynamique. Nous retrouvons là un phénomène déjà constaté avec l'alignement de séquences : lorsque la MTSAG n'est pas assez complexe, les équations de programmation dynamique sont plus performantes, bien que moins pratiques à manipuler (p. 89).

En revanche, la généralisation du formalisme de base des HMM pour créer des automates plus puissants (Kulp *et al.*, 1996) pourra sans doute profiter de la souplesse apportée par les MTSAG 2-bandes, au moins pendant la phase de développement. Ainsi, nous verrons dans la section suivante que la généralisation des HMM aux grammaires stochastiques est plus efficacement traitée par une MTSAG 2-bandes que par la représentation standard sous forme de MTSAG 1-bande.

```
%{
#include <stdlib.h>
enum {IDX_A=0, IDX_C, IDX_G, IDX_T};

float *emit_table (float a, float b, float c, float d)
{float *p = (float *) malloc (4 * sizeof (float));
 p[0] = a;  p[1] = b;  p[2] = c;  p[3] = d;
 return p;
}
%}

%empty _
%attribute "int" idx
%token A { $$.idx = IDX_A; }
%token C { $$.idx = IDX_C; }
%token G { $$.idx = IDX_G; }
%token T { $$.idx = IDX_T; }
%abbreviation • A C G T

%attribute "float" p sum
%terminal_attribs {}
%default_nonterminal_attribs < p sum >

%default_constraint
{}{ $! .sum += $$.sum;
    if ($$.p > $! .p)
      $! .p = $$.p;
  }
%rule_model N T %generic f1 <"float" "float *">
{ $$.p   = $1.p   * $#[2][$2.left[0].idx] * $#[1];
  $$.sum = $1.sum * $#[2][$2.left[0].idx] * $#[1];
}
%rule_model N %generic f2 <"float">
  $$.p   = $1.p   * $#[1];
  $$.sum = $1.sum * $#[1];
}

%%

E   : I4 •   % f1<"1.0/2.0"
               "emit_table(0.25, 0.25, 0.25, 0.25)">
    | D3     % f2 <"1.0/2.0">
    | M3 •   % f1<"2.0/3.0"
               "emit_table(0.25, 0.25, 0.25, 0.25)">;

I4  : I4 •   % f1<"1.0/2.0"
               "emit_table(0.25, 0.25, 0.25, 0.25)">
    | D3     % f2 <"1.0/2.0">
    | M3 •   % f1<"1.0/3.0"
               "emit_table(0.25, 0.25, 0.25, 0.25)">;

M3 : I3 •   % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | D2     % f2 <"1.0/3.0">
   | M2 •   % f1<"2.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">;

I3  : I3 •  % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | D2     % f2 <"1.0/3.0">
   | M2 •   % f1<"1.0/6.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">;

D3 : I3 •   % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | D2     % f2 <"1.0/3.0">
   | M2 •   % f1<"1.0/6.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">;

M2 : I2 •   % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | D1     % f2 <"1.0/3.0">
   | M1 •   % f1<"2.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">;

I2  : I2 •  % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | D1     % f2 <"1.0/3.0">
   | M1 •   % f1<"1.0/6.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">;

D2 : I2 •   % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | D1     % f2 <"1.0/3.0">
   | M1 •   % f1<"1.0/6.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">;

M1 : I1 •   % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | S      % f2 <"2.0/3.0">;

D1 : I1 •   % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | S      % f2 <"1.0/6.0">;

I1  : I1 •  % f1<"1.0/3.0"
              "emit_table(0.25, 0.25, 0.25, 0.25)">
   | S      % f2 <"1.0/6.0">;

S   : •     { $$.p = 1;  $$.sum = 1; };
%%
```

Figure 3.22: La MTSAG suivante est une version tronquée de la MTSAG disponible dans la distribution de MTSAG2C. Tous les symboles ont la même probabilité d'être émis (0,25). Le premier paramètre des fonctions génériques correspond à la probabilité de passer d'un état à un autre. Le deuxième paramètre de la fonction générique f1 représente la table des probabilités d'émission des différents symboles (ordre A, C, G, T dans la table). La contrainte d'optimisation effectue simultanément la somme des probabilités de dérivation et la maximisation de la probabilité de la dérivation retenue.

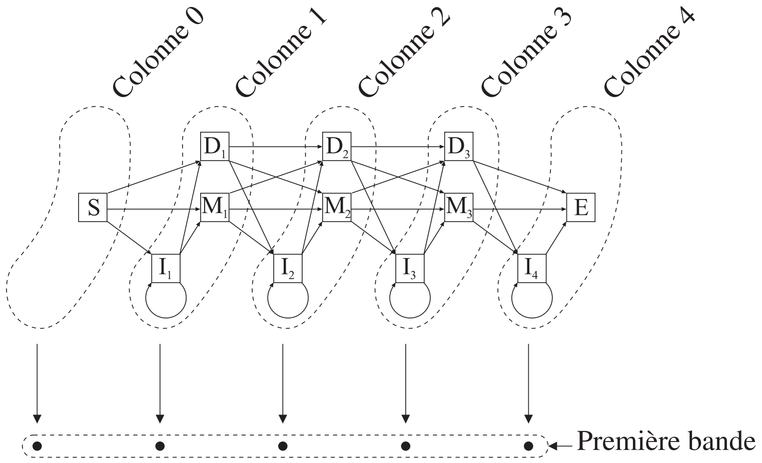


Figure 3.23: Décomposition du HMM de la figure 3.21 en colonnes et codage de ces colonnes sous la forme d'une bande ne contenant que le caractère '•' répété autant de fois qu'il y a de colonnes. En pratique, les probabilités des transitions seront codées sous la forme d'attributs physiques de ces caractères '•'.

3.9 Grammaires non-contextuelles stochastiques

Les grammaires non-contextuelles stochastiques, ou SCFG, sont une généralisation des HMM dans le sens où l'automate sous-jacent passe de l'automate fini non-déterministe à une bande à un automate à pile non-déterministe à une bande. Nous avons déjà décrit le cadre habituel d'utilisation des SCFG dans la section 1.7.3 p. 64. Les résultats principaux de cette section antérieure sont les suivants :

- Les SCFG servent à replier, aligner et discriminer les ARN d'une famille donnée, grâce au codage de la structure consensus de la famille sous forme de règles de grammaire, et à l'ajout de probabilités de dérivation optimisées pour la famille ;
- La grammaire G obtenue a une taille $|G|$ proportionnelle au nombre de bases de la structure consensus ;
- Les SCFG sont typiquement décrites à l'aide d'une méta-grammaire de haut niveau, puis générées à l'aide d'un programme spécialisé ;
- Les MTSAG mono-bande peuvent être utilisées pour écrire les SCFG et obtenir ainsi automatiquement un analyseur syntaxique calculant pour tout séquence A l'arbre de dérivation de A le plus probable ainsi que la probabilité de A ;
- La complexité d'analyse d'un ARN de longueur n suivant une MTSAG G est $O(|G|n^3)$ en temps et $O(|G|n^2)$ en espace ;
- Les analyseurs syntaxiques générés par MTSAG2C ont des performances très satisfaisantes.

```
. 0          0          0          0
  0          0          0
  0.6666666  0.1666666  0.1666666
  0          0          0
. 0.25       0.25       0.25       0.25
  0.6666666  0.1666666  0.1666666
  0.3333333  0.3333333  0.3333333
  0.3333333  0.3333333  0.3333333
. 0.25       0.25       0.25       0.25
  0.6666666  0.1666666  0.1666666
  0.3333333  0.3333333  0.3333333
  0.3333333  0.3333333  0.3333333
. 0.25       0.25       0.25       0.25
  0          0.6666666  0.3333333
  0          0.5        0.5
  0.3333333  0.3333333  0.3333333
. 0.25   0.25   0.25   0.25
  0      0      0
  0      1      0
  0      0.5    0.5
```

```
%{
%}
delim          [ \t]
ws             {delim}+
digit          [0-9]
fraction       "."{digit}*
exposant       [Ee][+-]?{digit}+
mantisse       [+-]?{digit}+{fraction}?
double         {mantisse}{exposant}?
%%
"A"|"a"        { return A; }
"C"|"c"        { return C; }
"G"|"g"        { return G; }
"T"|"t"        { return T; }
"."            { return '.'; }
{double}       { return VALUE; }

«EOF»          { return 0; }
.              {}
%%
```

Figure 3.24: Exemple de première bande (à gauche) codant les probabilités données dans la grammaire de la figure 3.22. Les quatre premières valeurs après le **.** correspondent aux probabilités d'émission de A, C, G, T respectivement. Les 9 valeurs suivantes correspondent aux probabilités de transition de M, D, I vers M, D, I. Cette première bande est directement utilisable par la grammaire de la figure 3.26 grâce à l'analyseur lexical associé (à droite). Lorsque l'analyseur lit un réel, il retourne le lexème VALUE, utilisé par l'analyseur syntaxique lors du calcul des attributs physiques du token '**.**' (cf. figure 3.26).

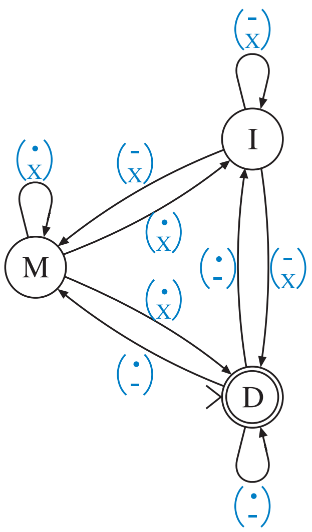


Figure 3.25: Factorisation du HMM de la figure 3.23. Les symboles de la deuxième séquence sont représentés par un X. L'état D est l'état de départ et l'état d'arrivée car les états S et E du HMM original sont assimilables à des états de délétion.

```
%{
#include <stdio.h>
#include <stdlib.h>
#define VALUE -1

extern char *yytext;
void read_values (int n, float **p)
{ int i;

  *p = (float *) malloc (n * sizeof (float));
  for (i = 0;  i < n;  i++)
    { if (yylex () == VALUE)
        sscanf (yytext, "%f", &(*p)[i]);
      else
        { fprintf(stderr, "Erreur de syntaxe\n");
          exit (1);
        }
    }
}

enum {IDX_A = 0, IDX_C, IDX_G, IDX_T };
enum {M = 0, D, I};
%}

%nb_tapes 2
%empty _

%attribute "int" idx
%token A { $$.idx = IDX_A;  }
%token C { $$.idx = IDX_C;  }
%token G { $$.idx = IDX_G;  }
%token T { $$.idx = IDX_T;  }
%abbreviation • A C G T

%attribute "float *" emit
%attribute "float **" transition
%token '.'
{ $$.transition = (float **)malloc(3*sizeof (float *));
  read_values (4, &$$.emit);
  read_values (3, &$$.transition[M]);
  read_values (3, &$$.transition[D]);
  read_values (3, &$$.transition[I]);
}

%attribute "float" p sum
%terminal_attribs {}
%default_nonterminal_attribs <p sum>
```

```
%default_constraint
{}{ $! .sum += $$.sum;
     if ($$.p > $! .p)
       $! .p = $$.p;
   }

%rule_model N [T T]
%generic f1 <"int" "int">
{
  if ($1[1].right[1].lex_code! = 0)
    { float p;

      if ($2[2].len)
        p = $1[1].right[1].emit[$2[2].left[0].idx];
      else p = 1;

      p *= $1[1].right[1].transition[$#[1]][$#[2]];
      $$.p     = $1.p     * p;
      $$.sum = $1.sum * p;
    }
  else $$.p = $$.sum = 0;
}

%rule_model [T T]
%generic f2 <"int" "int">
{ float p;

  p = $1[1].right[0].transition[$#[1]][$#[2]];
  $$.p = $$.sum = p;
}

%%

D   : M  ['.' •]    % f1 <"M" "D">
    | D  ['.' _]    % f1 <"D" "D">
    | I  [_ •]      % f1 <"I" "D">
    |    ['.' _]    % f2 <"D" "D"> ;

M   : M  ['.' •]    % f1 <"M" "M">
    | D  ['.' _]    % f1 <"D" "D">
    | I  [_ •]      % f1 <"I" "M">
    |    ['.' _]    % f2 <"D" "M"> ;

I   : M  ['.' •]    % f1 <"M" "I">
    | D  ['.' _]    % f1 <"D" "D">
    | I  [_ •]      % f1 <"I" "I">
    |    ['.' _]    % f2 <"D" "I"> ;

%%
```

Figure 3.26: Cette MTSAG 2-bandes possède un en-tête de taille importante. En fait, la majeure partie de cet en-tête contient le code de lecture des attributs physiques `emit` et `transition` des tokens `'.'` de la première bande. La fonction générique f1 effectue la mise à jour des probabilités en fonction de la transition choisie (le premier paramètre est l'état source et le deuxième paramètre l'état destination). La fonction f2 est utilisée uniquement pour les transitions initiales. La syntaxe d'accès aux différents attributs physiques est expliquée en détail en annexe A.

```
S                Branch       molecule     end
end              Loop         1
molecule         Helix        7            rest1
rest1            Branch       btw12        rest2
btw12            Loop         2
rest2            Branch       d-arm        rest3
d-arm            Helix        4            d-arm-loop
d-arm-loop       VariableLoop 8
rest3            Branch       btw23        rest4
btw23            Loop         1
rest4            Branch       anti-codon   rest5
anti-codon       Helix        5            anti-codon-loop
anti-codon-loop  Loop         7
rest5            Branch       variable     t-arm
variable         VariableLoop 8
t-arm            Helix        5            t-arm-loop
t-arm-loop       Loop         7
```

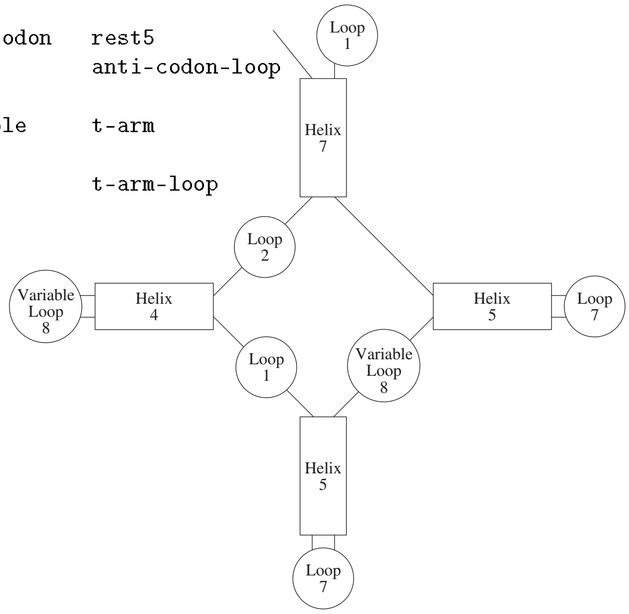


Figure 3.27: Méta-grammaire de la SCFG à 75 non-terminaux et 660 productions utilisée pour les ARN de transfert (en haut à gauche) et représentation graphique équivalente (en bas à droite).

Le premier inconvénient des SCFG telles que nous les avons décrites (et telles qu'elles sont décrites dans (Sakakibara *et al.*, 1994a) ou (Sakakibara *et al.*, 1994b)) est une conséquence de leur taille proportionnelle à la taille de la structure consensus à représenter. Cette proportionnalité est rendue nécessaire par le principe même de ces grammaires, mais elle induit des grammaires ingérables manuellement. L'usage d'une pseudo-grammaire de haut niveau (figure 3.27) et d'un générateur transformant cette pseudo-grammaire en SCFG permet de contourner ce problème. Cependant, le générateur utilisé n'est jamais décrit formellement, et pourtant son influence sur la puissance des SCFG générées est non négligeable. Ainsi, la déclaration `d-arm Helix 4 d-arm-loop` de la figure 3.27 peut être interprétée par le générateur comme une instruction pour générer des règles correspondant à une hélice appelée `d-arm` de longueur 4 fixe, ou de longueur 4 en moyenne mais pouvant contenir des délétions ou des insertions. La taille et les performances des grammaires obtenues dans les deux cas ne seront certainement pas les mêmes. Qui plus est, le générateur prend typiquement la

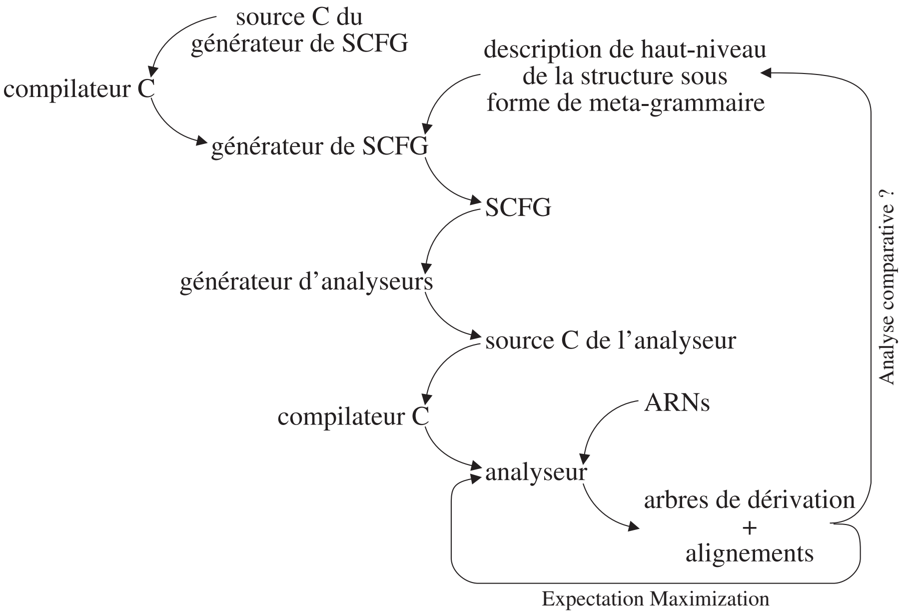


Figure 3.28: Cycle de développement habituel d'une SCFG. L'apprentissage d'une SCFG est un processus itératif à deux niveaux. Le premier niveau consiste à optimiser les probabilités des règles de la grammaire à l'aide d'une étape du type "Expectation Maximization" (Sakakibara *et al.*, 1994a). La deuxième étape pourrait (aucune réalisation n'a encore été décrite) utiliser un algorithme d'analyse comparative pour regénérer une grammaire de haut niveau à partir des alignements et structures obtenus avec la grammaire initiale.

forme d'un programme C qui doit être écrit, déverminé, expliqué et maintenu. Sans la pression de la publication, les deux points précédents (explication et maintenance) peuvent aisément se transformer en vœux pieux.

Le deuxième inconvénient de la présentation usuelle des SCFG provient de la nécessité de faire évoluer les règles de la SCFG lors de son apprentissage, et non plus seulement les probabilités des règles. Tout changement de grammaire (que ce soit un changement de la méta-grammaire ou un changement de la SCFG générée) impose une regénération de l'analyseur syntaxique et une recompilation de cet analyseur (figure 3.28). C'est un processus pour le moins malcommode.

Pour supprimer ces deux inconvénients, nous adaptons aux SCFG le principe utilisé dans la section précédente pour les HMM : nous codons la SCFG sur une première bande, qui constituera donc une nouvelle forme de description de haut-niveau, et remplaçons le générateur de SCFG par une MTSAG à deux bandes (figure 3.29). De cette manière, le générateur de SCFG est remplacé par une MTSAG fixe, relativement courte et facile à écrire ou à modifier, tandis que la recompilation de l'analyseur syntaxique n'est plus nécessaire lors d'un changement de méta-grammaire.

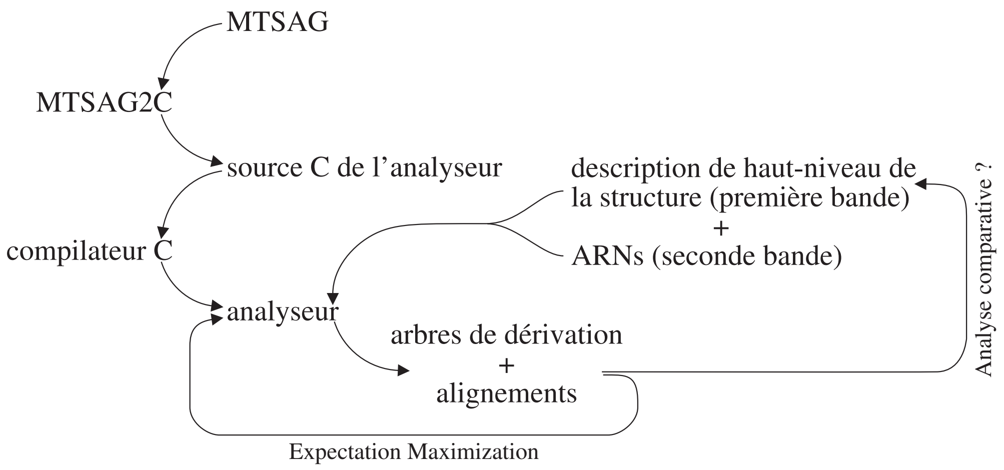


Figure 3.29: Cycle de développement d'une SCFG à l'aide d'une MTSAG à deux bandes. L'apprentissage de la SCFG est toujours un processus itératif à deux niveaux, mais cette fois la génération d'une nouvelle description de haut niveau ou méta-grammaire se résume à la génération d'une première bande, sans recompilation de l'analyseur.

3.9.1 MTSAG 2-bandes et ARN de transfert

Comme première illustration de principe, nous allons transformer la méta-grammaire de la figure 3.27 en première bande. Puis nous remplacerons le générateur de SCFG utilisé avec cette méta-grammaire par une MTSAG à deux bandes lisant la description de haut niveau sur la première bande et l'ARN à aligner et replier sur la deuxième bande.

La méta-grammaire de la figure 3.27 comporte quatre types de déclarations interprétées de manière spécifique dans le cas qui nous intéresse :

Branch Cette déclaration permet d'introduire des boucles multiples dans les structures. Comme les règles associées à ces branchements se contentent de multiplier les probabilités des sous-structures des boucles multiples (fonction générique f1 de la figure 1.13 p. 68), nous coderons implicitement cette déclaration dans la MTSAG.

Helix k Le générateur interprète cette déclaration comme étant celle d'une hélice de longueur fixe k, chaque paire de l'hélice ayant une certaine probabilité de générer une des seize combinaisons de bases appariées. Nous représenterons cette déclaration sur la première bande par k couples successifs de parenthèses imbriquées. Les seize probabilités associées aux seize combinaisons seront stockées comme attributs physiques de la parenthèse ouvrante de chaque couple.

Loop k Il s'agit de la déclaration d'une portion simple brin de longueur fixe k, chaque position de la portion ayant une certaine probabilité de générer une des quatre bases possibles. Nous représenterons cette déclaration sur la première bande par k

points `'.'` successifs. Les quatre probabilités associées aux quatre bases possibles seront stockées comme attributs physiques de ce point.

VariableLoop k Cette déclaration est la plus complexe car c'est celle d'une boucle de taille variable allant de 1 à un certain maximum et dont la taille consensus est k. Le générateur doit écrire un certain nombre de règles autorisant les délétions et les insertions à chaque position de la boucle. Ces boucles variables se comportent comme des HMM. Nous les représenterons par k étoiles `'*'`. Les probabilités associées aux états du HMM sous-jacent seront stockées comme attributs physiques de ces étoiles, comme dans la section 3.8 (nous avions alors utilisé le symbole `'.'` à la place de `'*'`).

Une fois choisis ces codages, la méta-grammaire de la figure 3.27 est transformée en la première bande suivante (les probabilités n'ont pas été représentées mais elles sont en fait écrites en même temps que les symboles suivant une syntaxe proche de celle utilisée dans la figure 3.24 p. 141) :

```
(((((((..((((********)))).(((((.......)))))********(((((.......)))))))))))).
```

Ensuite, nous devons exprimer dans la MTSAG 2-bandes le fait qu'un alignement entre un ARN et une première bande est une suite d'alignements de portions fixes, portions variables et hélices. Nous utiliserons les règles suivantes à cet effet :

```
RNA         : FixedStrand
            | VariableStrand
            | DSEndedRNA
                  { $$.p = $1.p; }
            | DSEndedRNA FixedStrand
            | DSEndedRNA VariableStrand
                  { $$.p = $1.p * $2.p; } ;

DSEndedRNA  : Helix
                  { $$.p = $1.p; }
            | RNA Helix
                  { $$.p = $1.p * $2.p; } ;
```

Le non-terminal DSEndedRNA a été introduit dans la grammaire précédente pour éviter l'apparition de dérivations

$$\text{RNA} \to^* \alpha \text{ FixedStrand FixedStrand } \beta$$

ou

$$\text{RNA} \to^* \alpha \text{ VariableStrand VariableStrand } \beta,$$

qui ont pour inconvénient de décomposer une même boucle en deux entités différentes.

Chaque hélice est une suite de paires de bases correctement imbriquées, sans insertion ni délétion. Une hélice est terminée par un ensemble de structures équivalent à l'ensemble défini par le non-terminal `RNA`.

```
Helix : ['(' •/"("] RNA [')' •/")"]
      | ['(' •/"("] Helix [')' •/")"]
            { $$.p = $2.p * $1[1].left[0].proba_ds[$1[2].left[0].idx][$3[2].left[0].idx]; };
```

Les portions simple brin de longueur fixe sont facilement décrites car il s'agit d'alignements sans insertion ni délétion :

```
FixedStrand   : ['.' •]
                      { $$.p = $1[1].left[0].proba_ss[$1[2].left[0].idx]; };
              | FixedStrand ['.' •]
                      { $$.p = $1.p * $2[1].left[0].proba_ss[$2[2].left[0].idx]; };
```

Les portions simple brin de longueur variable sont plus complexes à gérer car il faut exprimer le fait qu'à chaque position '*', des bases supplémentaires peuvent être insérées, ou que la position peut être supprimée.

```
VariableStrand : MatchInsertOrDelete
                      { $$.p = $1.p; }
               | VariableStrand MatchInsertOrDelete
                      { $$.p = $1.p * $2.p; };

MatchInsertOrDelete
               | ['*' •]
                      { $$.p = $1[1].left[0].proba_match[$1[2].left[0].idx]; }
               : ['*' •] InsertBases
                      { $$.p = $2.p * $1[1].left[0].proba_end_insert[$2[2].right[0].idx]; }
               | ['*' _]
                      { $$.p = $1[1].left[0].proba_delete[0]; }

InsertBases    : [_ •]
                      { $$.p = $1[1].right[0].proba_begin_insert[$1[2].left[-1].idx]; }
               | InsertBases [_ •]
                      { $$.p = $1.p * $1[1].right[0].proba_insert[$1[2].right[0].idx]; }
```

La synthèse des règles précédentes, ainsi que l'écriture des déclarations associées, en particulier celles de lecture des attributs physiques des tokens de la première bande, fournit la MTSAG de la figure 3.30. Une grande partie de l'en-tête de cette MTSAG est très semblable à l'en-tête de la MTSAG utilisée pour la description 2-bandes des HMM (figure 3.26). Les codes des fonctions d'évaluation des attributs des non-terminaux dans la grammaire précédente sont sans aucun doute d'un abord difficile, mais une lecture de l'annexe A permettra de les décrypter.

La projection de la MTSAG sur la première bande fournit une grammaire LR(1) non-récursive à droite. La complexité d'analyse est donc $O(mn^3)$ en temps et $O(mn^2)$ en espace, où m est la longueur de la première bande et n la longueur de la deuxième bande, c'est-à-dire le nombre de bases de l'ARN à analyser. Comme la taille m de la première bande est proportionnelle à la taille de la structure consensus qu'elle doit modéliser, les complexités obtenues sont identiques aux complexités de l'analyseur mono-bande[13]. Les performances pratiques des deux analyseurs ont été comparées

[13] Ces complexités et le codage utilisé sur la première bande ne sont pas sans rappeler les complexités et le codage du modèle de Corpet et Michot (p. 117).

```
%{
/* voir l'en-tête de la figure 3.26 */
%}

%nb_tapes 2
%empty _

%attribute "int" idx
%token C { $$.idx = 0; }
%token G { $$.idx = 1; }
%token U { $$.idx = 2; }
%token A { $$.idx = 3; }
%abbreviation • A C G T

%attribute "float *" proba_ds
%token '('
{ $$.proba_ds = (float **)malloc(4*sizeof (float *));
  read_values (4, &$$.proba_ds[0]);
  read_values (4, &$$.proba_ds[1]);
  read_values (4, &$$.proba_ds[2]);
  read_values (4, &$$.proba_ds[3]);
}
%token ')'

%attribute "float *" proba_ss
%token '.'  { read_values (4, &$$.proba_ss); }

%attribute "float *" proba_match
%attribute "float *" proba_delete
%attribute "float *" proba_begin_insert
%attribute "float *" proba_insert
%attribute "float *" proba_end_insert
%token '*'
{ float *d;
  read_values (4, &$$.proba_match);
  read_values (1, &$$.proba_delete);
  read_values (4, &$$.proba_begin_insert);
  read_values (4, &$$.proba_insert);
  read_values (4, &$$.proba_end_insert);
  read_values (4, &d);  $$.max_len = *d;
}

%attribute "float" p
%terminal_attribs {}
%default_nonterminal_attribs <p>

%default_constraint
{}
{ if ($$.p > $! .p) $! .p = $$.p; }

%%
```

```
RNA
    : FixedStrand
    | VariableStrand
    | DSEndedRNA      { $$.p = $1.p; }
    | DSEndedRNA FixedStrand
    | DSEndedRNA VariableStrand
        { $$.p = $1.p * $2.p; } ;

DSEndedRNA
    | RNA Helix       { $$.p = $1.p * $2.p; } ;
    : Helix           { $$.p = $1.p; }

Helix
    : ['(' •/"("] RNA [')' •/")"]
    | ['(' •/"("] Helix [')' •/")"]
        { $$.p = $2.p *
          $1[1].left[0].proba_ds[$1[2].left[0].idx]
                                 [$3[2].left[0].idx]; };

FixedStrand
    : ['.' •]
        { $$.p = $1[1].left[0].proba_ss
                                 [$1[2].left[0].idx]; };
    | FixedStrand ['.' •]
        { $$.p = $1.p * $2[1].left[0].proba_ss
                                 [$2[2].left[0].idx]; };

VariableStrand
    : MatchInsertOrDelete
        { $$.p = $1.p; }
    | VariableStrand MatchInsertOrDelete
        { $$.p = $1.p * $2.p; } ;

MatchInsertOrDelete
    : ['*' •] InsertBases
        { $$.p = $2.p*$1[1].left[0].proba_end_insert
                                 [$2[2].right[0].idx]; }
    | ['*' •]
        { $$.p = $1[1].left[0].proba_match
                                 [$1[2].left[0].idx]; }
    | ['*' _]
        { $$.p = $1[1].left[0].proba_delete[0]; }

InsertBases
    : [_ •]
        { $$.p = $1[1].right[0].proba_begin_insert
                                 [$1[2].left[-1].idx]; }
    | InsertBases [_ •]
        { $$.p = $1.p + $1[1].right[0].proba_insert
                                 [$1[2].right[0].idx]; }

%%
```

Figure 3.30: Extrait de MTSAG 2-bandes utilisable pour aligner et replier les ARN dont la structure consensus est décrite sur une première bande à l'aide des méta-symboles `'('`, `')'`, `'.'` et `'*'`. La syntaxe d'accès aux différents attributs physiques est expliquée en détail en annexe A. Pour des raisons d'espace sur la page, cette grammaire ne contient pas d'attribut physique `sum` (calcul de la probabilité d'une séquence), mais les modifications à apporter pour ajouter cet attribut sont triviales.

```
(((((((..((((********)))).(((((.......)))))********(((((.......))))))))))))).
CCUUCUGUAGCUCAAUUGGUAGAGCAUGUGACUGUAGAGUAUGCGGGUAUCACAGGGUCGCUGGUUCGAUUCCGGCCGGAAGG
```

(a) Première bande et ARN de 83 bases tels qu'ils sont fournis à l'analyseur 2-bandes (i.e. non alignés).

```
(((((((..((((........)))).(((((.......)))))...............(((((.......))))))))))))).
CCUUCUGUAGCUCAAUUGGUAGAGCAUGUGACUGUAGAGUAUGCGGGUAUCACAGGGUCGCUGGUUCGAUUCCGGCCGGAAGG
```

(b) Structure de l'ARN de la deuxième bande telle qu'elle est déterminée par l'analyseur mono-bande. L'alignement avec les non-terminaux de la grammaire (i.e. la structure consensus) n'est pas directement visible.

```
(((((((..((((********)))).(((((.......)))))*******----------*(((((.......))))))))))))).
CCUUCUGUAGCUCAAUUGGUAGAGCAUGUGACUGUAGAGUAUGC--GG-GUAUCACAGGGUCGCUGGUUCGAUUCCGGCCGGAAGG
```

(c) Alignement des deux bandes du (a) par l'analyseur 2-bandes. La structure de l'ARN de la deuxième bande est obtenue en lisant directement les parenthèses correspondantes sur la première bande.

Figure 3.31: Comparaison des alignements et résultats obtenus par l'analyseur mono-bande et l'analyseur 2-bandes à partir de la même structure consensus de départ et du même ARN de 83 bases (ARN DY6050 extrait d'une compilation de RNAt (Sprinzi *et al.*, 1996)).

(sur une station DEC Alpha 250 4/266) en utilisant un représentant de la famille des ARN de transfert (figure 3.31) et les résultats obtenus sont regroupés dans la table suivante :

	1-bande	2-bandes
RNAt de 83 bases		
temps en secondes :	0,25	0,12
espace en Moctets :	1,7	0,8

Ces temps et espaces sont représentatifs de la famille des ARN de transfert et évoluent peu d'un ARN de transfert à l'autre.

L'analyseur utilisant la MTSAG 2-bandes est plus rapide et prend moins de place en mémoire que celui utilisant la MTSAG 1-bande équivalente et pourtant l'analyseur 2-bandes génère trois fois plus d'états attribués $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ que l'analyseur 1-bande (non représenté). Il semble en fait que cette accélération provienne essentiellement de la diminution du nombre de non-terminaux à gérer (7 au lieu de 75 dans cet exemple). Par ailleurs, l'alignement entre ARN et structure consensus n'est plus une conséquence cachée dans l'arbre de dérivation trouvé par l'analyseur syntaxique mono-bande (figure 3.31(b)), mais il apparaît explicitement dans le résultat fourni (figure 3.31(c)). Et dès l'instant où une MTSAG possède deux bandes, nous pouvons y ajouter des

S	2BranchRNec	5'end	rest1	forBulge2	2BranchRNec	restS3	bulge2
5'end	Loop	10		restS3	Helix	6	loop3
rest1	2BranchRNec	trunkHelix	rest3	loop3	Loop	7	
trunkHelix	Helix	6	rest2	bulge2	Loop	1	
rest2	2BranchRNec	stem1	top&Right	rest3	2BranchRNec	SmSite	rest4
stem1	Helix	5	forBulge1	SmSite	Loop	14	
forBulge1	2BranchRNec	bulge1	restStem1	rest4	2BranchRNec	stem4	OHend
bulge1	Loop	1		stem4	Helix	5	forBulge3
restStem1	Helix	5	loop1	forBulge3	2BranchRNec	bulge3	restS4
loop1	Loop	10		bulge3	Loop	1	
top&Right	2BranchRNec	stem2	stem3	restS4	Helix	6	loop4
stem2	Helix	16	loop2	loop4	Loop	4	
loop2	Loop	10		OHend	Loop	2	
stem3	Helix	3	forBulge2				

Figure 3.32: Méta-grammaire de la structure consensus des snRNA de la famille U1.

contraintes d'alignement (cf. section 3.2 p. 93) pour améliorer les performances de l'analyseur 2-bandes. La table suivante est une synthèse des résultats obtenus.

	1-bande	2-bandes	2-bandes + contraintes d'alignement
RNAt de 83 bases			
temps en secondes :	0,25	0,12	0,075
espace en Moctets :	1,7	0,8	0,7

Ces temps et espaces sont typiques des temps obtenus avec différents représentants de la famille des ARN de transfert.

3.9.2 MTSAG 2-bandes et petits ARN nucléaires

Nous avons appliqué la même stratégie à la SCFG utilisée par Underwood pour décrire la structure consensus des petits ARN nucléaires (snRNA) de la famille U1 (Underwood, 1994). La méta-grammaire de cette structure consensus est rappelée en figure 3.32 et une représentation en est donnée figure 3.33. Tout d'abord, nous pouvons constater que la méta-grammaire utilise une déclaration *2BranchRNec* à la place de la déclaration *Branch* de la méta-grammaire des ARN de transfert. Cette déclaration *2BranchRNec* signifie que la partie gauche d'un branchement peut être omise avec une certaine probabilité, et que dans tous les cas, la partie droite reste nécessaire. Nous avons besoin d'autres méta-symboles sur la première bande pour représenter ces probabilités d'omission. Nous avons choisi les crochets `'['` et `']'` car ils symbolisent bien l'imbrication des paires de sous-structures. Ensuite, toutes les boucles de la méta-grammaire (déclaration *Loop*) sont cette fois ci de taille variable et les délétions de paires dans les hélices (déclaration *Helix*) sont permises. Cette flexibilité *non explicitement documentée dans la méta-grammaire* est apportée par le code du générateur de SCFG qui interprète la méta-grammaire plus librement que dans le cas des ARN de transfert. La première bande équivalente à la méta-grammaire de la figure 3.32 est représentée figure 3.34. Une fois cette première bande écrite, il ne reste plus qu'à construire une MTSAG 2-bandes l'interprétant en autorisant les boucles de

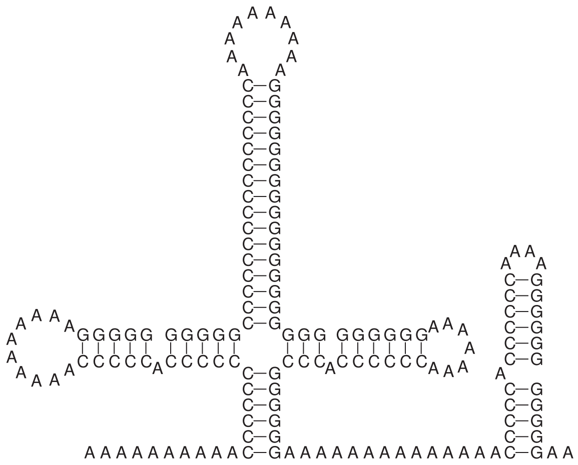


Figure 3.33: Structure secondaire consensus des snRNA de la famille U1.

taille variable, les délétions dans les hélices, et les suppressions de sous-structures entières. Cette MTSAG possède 10 non-terminaux et ressemble beaucoup à celle de la figure 3.30. Les règles de la grammaire sont reproduites en figure 3.35. L'en-tête et les fonctions d'évaluation d'attributs sont semblables à ceux de la grammaire de la figure 3.30. Ils doivent simplement être adaptés au contexte plus général de la grammaire et à la plus grande flexibilité de dérivation.

Les résultats de la comparaison des performances sur DEC Alpha 250 4/266 entre l'analyseur utilisé par Underwood[14], notre propre analyseur 1-bande, et l'analyseur 2-bandes avec contraintes d'alignement, sont regroupés dans la table suivante. La comparaison a utilisé un snRNA de 182 bases dont l'alignement avec la première bande est reproduit en figure 3.36.

	Underwood	1-bande	2-bandes + contraintes d'alignement
182 bases snRNA			
temps en secondes :	≈ 6	6,5	3,8
espace en Moctets :	≈ 128	25	10

Ces temps et espaces sont typiques des temps obtenus avec différents représentants de la famille des snRNA.

[14] Le temps réel étant d'approximativement 30 secondes sur une Decstation 5000/240 (Underwood, 1994) et il a été divisé par 5 pour le rendre comparable aux temps de calcul sur une DEC Alpha 250 4/266.

```
[
     ..........
     [
          ((((((
                [
                     (((((
                          [
                               .
                               (((((
                                    ..........
                               )))))
                          ]
                     )))))
                     [
                          (((((((((((((((
                                           ..........
                          )))))))))))))))
                          (((
                             [
                                 ((((((
                                        .......
                                 ))))))
                                 .
                             ]
                          )))
                     ]
                ]
          ))))))
          [
              ..............
              [
                   (((((
                        [
                             .
                             ((((((
                                   ....
                             ))))))
                        ]
                   )))))
                   ..
              ]
          ]
     ]
]
```

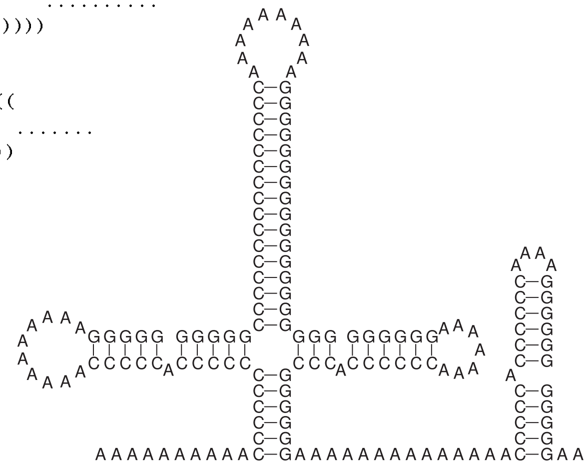


Figure 3.34: Représentation éclatée de la première bande équivalente à la méta-grammaire de la figure 3.32. La version 2D de la structure associée est rappelée à droite. Les probabilités ne sont pas représentées.

```
BranchRNec: ['[' _] All All [']' _]
          | ['[' _] LoopMat All [']' _]
          | ['[' _] All LoopMat [']' _]
          | ['[' _] SkipAll All [']' _]
          | ['[' _] SkipAll LoopMat [']' _]
          | ['[' _] SkipLoop All [']' _]
          ;

All       : DSMatch
          | DSDelete
          | BranchRNec
          ;

DSMatch   : ['(' ./"("] DSMatch [')' ./")"]
          | ['(' ./"("] DSDelete [')' ./")"]
          | ['(' ./"("] LoopMat [')' ./")"]
          | ['(' ./"("] BranchRNec [')' ./")"]
          ;

DSDelete  : ['(' _] DSMatch [')' _]
          | ['(' _] DSDelete [')' _]
          | ['(' _] LoopMat [')' _]
          | ['(' _] BranchRNec [')' _]
          ;

LoopIns   : RealIns
          | LoopDel RealIns
          | LoopMat RealIns
          ;

RealIns   : [_ .]
          | RealIns [_ .]
          ;

LoopDel   : ['.' _]
          | LoopIns ['.' _]
          | LoopDel ['.' _]
          | LoopMat ['.' _]
          ;

LoopMat   : ['.' .]
          | LoopIns ['.' .]
          | LoopDel ['.' .]
          | LoopMat ['.' .]
          ;

SkipLoop  : ['.' _]
          | SkipLoop ['.' _]
          ;

SkipAll   : ['(' _] SkipAll [')' _]
          | ['(' _] SkipLoop [')' _]
          | ['[' _] SkipAll SkipAll [']' _]
          | ['[' _] SkipLoop SkipAll [']' _]
          | ['[' _] SkipAll SkipLoop [']' _]
          ;
```

Figure 3.35: Règles de la MTSAG 2-bandes équivalente au générateur de SCFG utilisé par Underwood. Les non-terminaux commençant par *Skip* servent à absorber les symboles de la première bande inutiles lorsque la branche gauche d'un *BranchRNec* est omise.

```
[
    ..........
    UUACUUACCU
    [
        ((((((
        GGACGG
             [
                 (((((
                 GGUCA
                      [
                          -.
                          AU
                          (((((---..........)))))
                          --UGGCGAUCAUGAAGGUCCA--
                      ]
                 )))))
                 UGGCC
                 [
                     ((((((((((((((((--.----.----.-.-.-.....))))))))))))))))
                     ------UAGGUUGGUAACCUCCAUUGCACUUAGGAGAGGUGCCUACCUA------
                     (((
                     AGA
                        [
                            ((((((.......))))))
                            UCGGCC--CAAGUGGCCGA
                            .
                            A
                        ]
                     )))
                     UCU
                 ]
             ]
        ))))))
        ACGUCC
        [
            ..............
            AUAAUUUGUUGCUG
            [
                (((((
                AGGGG
                     [
                         -.
                         GC
                         ((((((....))))))
                         -CUGCGUUCGCGCGG-
                     ]
                )))))
                CCCCU
                ..
                GC
            ]
        ]
    ]
]
```

Figure 3.36: Représentation éclatée de l'alignement entre la première bande de la figure 3.34 et un snRNA de 182 bases.

3.9.3 Discussion

Nous avons montré que les MTSAG 2-bandes renouvellent le concept de grammaire stochastique. Les avantages des MTSAG 2-bandes par rapport à l'implémentation monobande des SCFG sont les suivants :

- le générateur de SCFG est remplacé par une MTSAG 2-bandes plus facile à écrire et à documenter,
- une MTSAG 2-bandes est indépendante de la famille d'ARN à modéliser,
- la description de haut niveau d'une structure consensus est aussi facile à écrire sous la forme d'une première bande spéciale que sous la forme d'une méta-grammaire,
- le changement de structure consensus, et donc de première bande, n'impose pas de recompilation de la grammaire,
- l'alignement entre structure consensus et ARN est un produit explicite de l'analyse syntaxique,
- les contraintes d'alignement peuvent aisément être utilisées avec les grammaires 2-bandes,
- les grammaires 2-bandes se prêtent naturellement à l'introduction de points d'ancrage destinés à faciliter le traitement de longs ARN (cf. section 3.11).

Tout ces avantages sont autant d'arguments tendant à montrer que les MTSAG 2-bandes pourraient constituer le formalisme nécessaire à la réalisation d'un algorithme d'apprentissage automatique d'une structure consensus à partir d'ARN non alignés et non repliés d'une même famille (cf. section 3.10).

Terminons cette section par une discussion autour de l'interprétation qu'il est possible de donner à cette notion de première bande représentant une structure et de deuxième représentant une donnée, le tout analysé grâce à une MTSAG 2-bandes. Tout se passe en fait comme si la MTSAG 2-bandes formait un méta-code compilé (par MTSAG2C, puis un compilateur C) chargé d'interpréter un code (la première bande) et de lire des données (la deuxième bande), alors que la MTSAG 1-bande est une version compilée du code de la première bande. On peut donc s'étonner qu'une MTSAG 2-bandes soit plus efficace en interprétant un code qu'une MTSAG 1-bande exécutant le code compilé. Cette anomalie montre en fait que MTSAG2C, et l'algorithme GCP sous-jacent, sont potentiellement améliorables dans leur étape d'analyse de la grammaire et d'extraction des informations pertinentes accélérant le travail de l'analyseur syntaxique.

3.10 Modèles de covariance

Parallèlement à l'introduction des SCFG par Haussler *et al*, Eddy et Durbin proposèrent une généralisation des HMM destinée à prendre en compte les appariements dans les ARN (Eddy & Durbin, 1994). Cette généralisation est appelée *modèles de covariance*[15] (ou Covariance Models, ou CM). Les auteurs de cette généralisation reconnaissent que tout CM peut sans doute être écrit sous la forme d'une SCFG. Cependant, les CM présentent une souplesse, une simplicité de description et une unité de structure qui les rendent plus facile à écrire qu'une SCFG équivalente. Cette facilité d'écriture se traduit par l'existence d'un algorithme d'apprentissage d'un CM à partir d'une famille d'ARN non alignés et non repliés. L'algorithme d'alignement et de repliement selon un CM, et dont le résultat est nécessaire à l'algorithme d'apprentissage, est représenté par des équations de programmation dynamique dont les complexités sont $O(mn^3)$ en temps et $O(mn^2)$ en espace (où m est la taille du CM et n la longueur de l'ARN). Sans surprise, nous allons appliquer les MTSAG aux CM selon un principe équivalent à celui détaillé dans la section précédente pour les SCFG. Nous retrouverons ainsi les mêmes complexités.

Techniquement, un CM est basé sur une décomposition en arbre binaire du squelette de la structure secondaire consensus de la famille d'ARN représentée (figure 3.37). Cela signifie que les structures secondaires représentables par un CM sont soumises aux mêmes contraintes que dans les autres modèles, et en particulier les paires de bases formées doivent être correctement imbriquées.

Les nœuds utilisés dans l'arbre binaire sont de 8 types :

- le nœud racine,
- les nœuds simples droits,
- les nœuds simples gauches,
- les nœuds doubles,
- les nœuds bifurcation,
- les nœuds début de branche gauche,
- les nœuds début de branche droite,
- les nœuds fin de branche.

Intuitivement, les nœuds simples et doubles correspondent à des colonnes d'un alignement multiple. Les nœuds doubles sont utilisés pour représenter les paires de bases dans la structure, alors que les nœuds simples sont utilisés pour représenter les bases non-appariées. Les nœuds bifurcation, début de branche et fin de branche permettent d'exprimer les branchements dans les structures secondaires.

Pour autoriser les insertions, délétions et mismatches dans la structure inflexible définie par un CM, les nœuds sont en fait décomposés en un ensemble de 6 états, à

[15] Le nom *covariance* provient de la technique d'apprentissage d'un modèle de covariance par calcul des covariations dans les paires de colonnes d'un alignement multiple d'ARN.

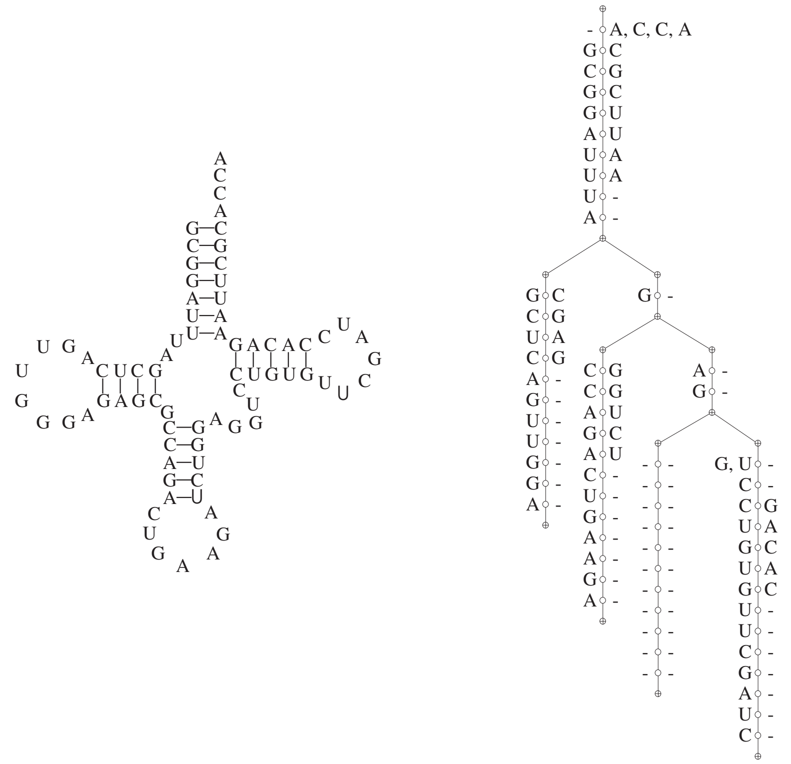


Figure 3.37: Structure secondaire de l'ARN de transfert DF6280 de la compilation (Sprinzi *et al.*, 1996) (à gauche) et décomposition de cette structure en arbre binaire (à droite) suivant les nœuds du CM créé pour les ARN de transfert par Eddy et Durbin (fichier `TRNA2.cm` de la distribution du logiciel COVE (ftp://genome.wustl.edu/pub/eddy/cove.tar.Z)). Pour cet ARN de transfert, la branche verticale de l'arbre responsable du codage de l'hélice dans la boucle variable d'un ARN de transfert est vide.

l'instar des colonnes d'un HMM qui sont décomposées en 3 états (match, délétion et insertion). Les 6 états utilisés sont :

- MATP (Match Pair), MATL (Match Left) et MATR (Match Right) pour les matches ;
- INSL (Insert Left), INSR (Insert Right) pour les insertions ;
- DEL pour les délétions.

Tous les nœuds contiennent en théorie les six états précédents. Cinq états (MATP, MATL, MATR, INSL, INSR) peuvent émettre des symboles à gauche ou à droite. Les transitions entre états ont une configuration fixe donnée dans la figure 3.38 par les transitions entre états du premier nœud double vers le deuxième nœud double. Certains types de nœuds ont des probabilités de transition nulles vers et partant de certains états. Ces états deviennent alors inutiles et peuvent donc être supprimés (figure 3.38). Ainsi, un nœud simple gauche ne contient en pratique que les états MATL, DEL et INSL. D'ailleurs un HMM est un CM sans nœud bifurcation et contenant uniquement des nœuds simples gauches (ou, de manière équivalente, uniquement des nœuds simples droits).

L'alignement d'une séquence sur un CM, c'est-à-dire le calcul de la séquence de transitions ayant la probabilité la plus élevée, est réalisable grâce à un algorithme de programmation dynamique (Eddy & Durbin, 1994). La conversion d'un CM en MTSAG passe par le codage du CM sous la forme d'une première bande variable, interprétée par une MTSAG 2-bandes fixe. Nous avons choisi d'utiliser un couple de parenthèses pour représenter l'alignement d'une paire de bases contre un nœud double. Les parenthèses obtenues après conversion de tous les nœuds doubles sont correctement imbriquées par construction. Nous avons ensuite utilisé un `L` pour représenter l'alignement d'une base contre un nœud simple gauche et un `R` pour l'alignement contre un nœud simple droit[16]. Les bifurcations sont représentées par deux sous-chaînes séparées par un `|` et encadrées par des crochets :

`[` *gauche* `|` *droite* `]`

Les nœuds "fin de branche" sont représentés par un caractère `*`, alors que le nœud racine est représenté par une paire de crochets encadrant la première bande obtenue. Chaque symbole de cette première bande, à l'exception du symbole `*`, possède des attributs physiques dans lesquels sont stockées les probabilités de transition et d'émission de base des états du nœud associé à ce symbole. Les 36 premières probabilités sont les probabilités de transition d'un des 6 états du nœud vers un des 6 états du nœud suivant. Certaines probabilités sont nulles lorsque l'état source ou l'état destination ne sont pas des états valides pour le type de nœud considéré. Les 32 probabilités suivantes sont les probabilités d'émission de base et elles se décomposent comme suit : 4 valeurs

[16] Notons qu'il n'était pas nécessaire de différencier les nœuds simples et doubles puisque les nœuds simples ne sont que des nœuds doubles dont certains états sont inutiles. Mais l'analyse syntaxique est plus efficace grâce à cette distinction.

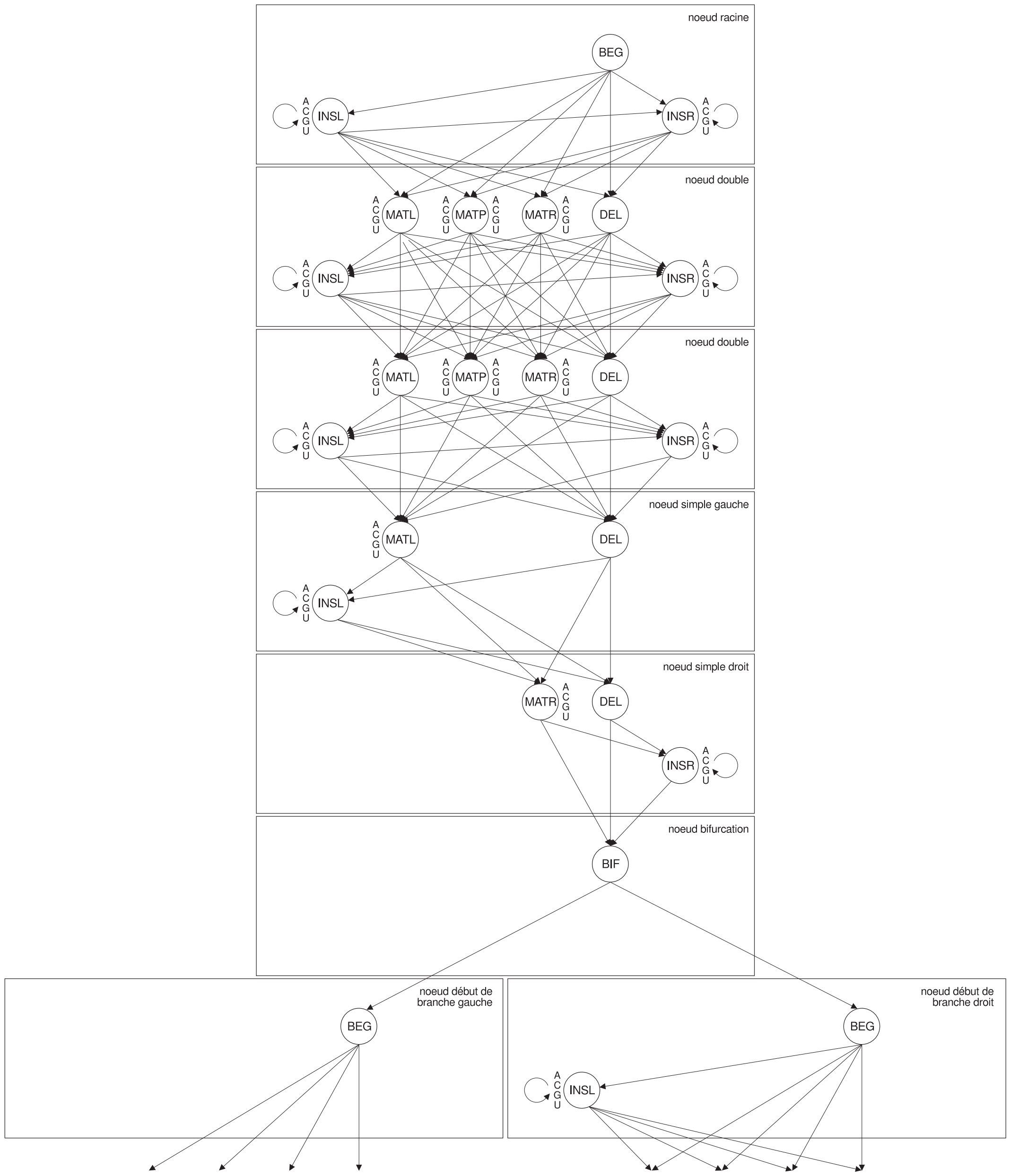


Figure 3.38: Découpage en états des différents types de nœuds. Aucun nœud de fin de branche n'est représenté, mais un tel nœud ne contient qu'un état DEL. Il n'y a que six états différents car les états BIF et BEG ne sont que des appellations différentes de l'état DEL.

```
[
    (((((((
           LL
           [
               ((((
                   LLLLLLLL*
               ))))
           |
               L
               [
                   (((((
                        LLLLLLL*
                   )))))
               |
                   LL
                   [
                        (((((((
                              LLLL*
                        )))))))
                   |
                        LL
                        (((((
                            LLLLLLL*
                        )))))
                   ]
               ]
           ]
    )))))))
    R
]
```

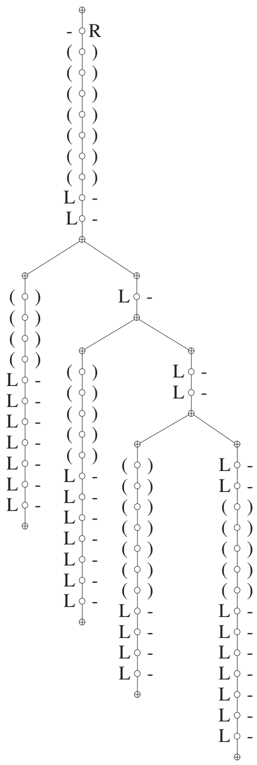


Figure 3.39: Représentation éclatée (à gauche) de la première bande équivalente au CM des ARN de transfert contenu dans le fichier **TRNA2.cm** de la distribution du logiciel COVE. Le CM équivalent est rappelé à droite. Les probabilités associées à chaque symbole n'ont pas été représentées.

pour INSL, 4 valeurs pour INSR, 16 valeurs pour les 16 couples de MATP, 4 valeurs pour MATL, 4 valeurs pour MATR. L'ordre adopté suit celui adopté par le logiciel COVE pour les probabilités des CM. La première bande donnée en figure 3.39 est un exemple de description de CM utilisable par une MTSAG.

Une fois choisie la méthode de codage des CM sous forme de première bande, il est aisé de construire une MTSAG interprétant cette première bande pour l'aligner contre un ARN donné sur une deuxième bande. La MTSAG doit coder le motif des transitions d'un état à l'autre. Une MTSAG possible est donnée en figures 3.40 et 3.41. La projection de cette MTSAG sur la première bande fournit une grammaire ambiguë. Pourtant, les complexités d'analyse sont $O(mn^3)$ en temps et $O(mn^2)$ en espace, où m est la taille de la première bande analysée (i.e. la taille du CM) et n celle de la deuxième bande (i.e. la taille de l'ARN). L'explication de ce résultat est simple : l'ambiguïté de la grammaire projetée sur la première bande est due aux différents étiquetages possibles des sommets de l'unique arbre de dérivation de chaque séquence. Le remplacement des différentes étiquettes possibles (DEL, MATP, MATL, MATR, INSL, INSR) par une seule étiquette (NODE) fournit la grammaire équivalente[17] :

ROOT → `'['` NODE `']'`
NODE → `'('` NODE `')'`
| `'L'` NODE
| NODE `'R'`
| `'['` NODE `'|'` NODE `']'`
| `'*'`

Cette grammaire est encore ambiguë, mais elle est beaucoup plus simple à comprendre que la grammaire projetée d'origine[18]. Il n'est pas difficile de constater que l'analyseur GCP pour cette grammaire est linéaire en temps et en espace : la seule ambiguïté potentiellement génératrice de non-linéarité provient de la présence simultanée d'une règle récursive à droite (NODE → `L` NODE) et d'une règle récursive à gauche (NODE → NODE `R`). De plus, pour que l'analyseur GCP analyse effectivement des chaînes en temps non-linéaire il faudrait que des suites non-bornées de nœuds de type `L` puissent être adjacentes à des suites non-bornées de nœuds de type `R` dans une branche de l'arbre binaire du CM associé. En pratique, lorsqu'un nœud de type `L` est adjacent à un nœud de type `R` dans un arbre, ces deux nœuds sont fusionnés en un seul nœud de type double (représenté par deux parenthèses).

La comparaison entre l'analyseur produit à partir de la MTSAG présentée dans cette section et le noyau d'alignement du logiciel COVE de Eddy et Durbin montre que, pour ce type de CM, les MTSAG ne présentent pas d'avantage déterminant : les analyseurs utilisent un peu moins de mémoire mais sont sensiblement plus lents que COVE. Une MTSAG est en revanche plus facile à modifier que les équations de programmation dynamique de COVE. En particulier, il devrait être facile d'ajouter le traitement des points d'ancrage à la MTSAG (cf. section suivante) pour pouvoir aligner et replier des ARN de plusieurs centaines de bases.

[17]Le nombre d'états $[\Delta \rightarrowtail \alpha]$ produits par une analyse pour la grammaire projetée d'origine est borné par une constante multipliée par le nombre d'états produits par une analyse pour la grammaire équivalente.

[18]Il est possible de construire une MTSAG dont la projection sur la première bande est cette grammaire simplifiée. Mais il faut alors coder les états de chaque nœud sous la forme d'attributs physiques du non-terminal NODE. L'écriture des fonctions d'évaluation d'attributs est donc plus complexe, mais la MTSAG obtenue est transformée par MTSAG2C en un analyseur syntaxique plus rapide que celui de la grammaire présentée dans cette section.

```
%{
/* voir l'en-tête de la figure 3.26 pour
   la définition de la fonction read_values */
enum {DEL = 0, MATP, MATL, MATR,
        INSL, INSR};
#define NEGINFINITY -999999
%}

%nb_tapes 2
%empty _

%attribute "int" idx
%token A {$$.idx = 0; }
%token C {$$.idx = 1; }
%token G {$$.idx = 2; }
%token T {$$.idx = 3; }
%abbreviation • A C G T

%attribute "int **" transit matp
%attribute "int *"  matl matr insl insr
%token '*'
%token '(' ')' 'L' 'R' '[' ']' '|'
{
  $$.transit = (int **) malloc (6 * sizeof (int));
  read_values (6, &$$.transit[0]);
  read_values (6, &$$.transit[1]);
  read_values (6, &$$.transit[2]);
  read_values (6, &$$.transit[3]);
  read_values (6, &$$.transit[4]);
  read_values (6, &$$.transit[5]);
  read_values (4, &$$.insl);
  read_values (4, &$$.insr);
  $$.matp = (int **) malloc (4 * sizeof (int));
  read_values (4, &$$.matp[0]);
  read_values (4, &$$.matp[1]);
  read_values (4, &$$.matp[2]);
  read_values (4, &$$.matp[3]);
  read_values (4, &$$.matl);
  read_values (4, &$$.matr);
}

%attribute "int" p
%terminal_attribs {}
%default_nonterminal_attribs <p>

%default_constraint
{ if ($$.p < NEGINFINITY / 2)
   return 0;
}
{ if ($$.p > $! .p)
   $! .p = $$.p;
}

%rule_model [T T] N [T T]
%generic f1 <"int" "int">
{
  $$.p = $2.p + $1[1].left[0].transit[$#[1]][$#[2]];
  if ($1[2].len)
     {
      if ($3[2].len)
         $$.p += $1[1].left[0].matp[$1[2].left[0].idx]
                                   [$3[2].left[0].idx];
      else
         $$.p += $1[1].left[0].matl[$1[2].left[0].idx];
  }
  else if ($3[2].len)
     $$.p += $1[1].left[0].matr[$3[2].left[0].idx];
}

%rule_model [T T] N
%generic f2 <"int" "int">
{
  $$.p = $2.p + $1[1].left[0].transit[$#[1]][$#[2]];
  if ($1[2].len)
     $$.p += $1[1].left[0].matl[$1[2].left[0].idx];
}

%rule_model N [T T]
%generic f3 <"int" "int">
{
  $$.p = $1.p + $2[1].left[0].transit[$#[1]][$#[2]]
  if ($2[2].len)
     $$.p += $2[1].left[0].matr[$2[2].left[0].idx];
}

%rule_model N
%generic f4 <"int" "int">
{
  $$.p = $1.p + $1[1].left[-1].transit[$#[1]][$#[2]];
}

%rule_model [T T] N
%generic f5 <"int" "int">
{
   $$.p = $2.p + $2[1].left[-1].transit[$#[1]][$#[2]]
                + $2[1].left[-1].insl[$1[2].left[0].idx];
}

%rule_model N [T T]
%generic f6 <"int" "int">
{
   $$.p = $1.p + $1[1].right[1].transit[$#[1]][$#[2]]
                + $1[1].right[1].insr[$2[2].left[0].idx];
}
```

Figure 3.40: En-tête de la MTSAG utilisée pour les CM. La colonne de gauche contient notamment les déclarations des tokens et de leurs attributs physiques. La colonne de droite contient les déclarations des fonctions génériques utilisées dans les règles de grammaire de la figure 3.41. Les probabilités lues par la fonction read_values sont en fait des logarithmes de probabilités, convertis en représentation en virgule fixe avec trois chiffres après la virgule (i.e. des entiers). Cette conversion en entier est celle utilisée par l'outil COVE de Eddy.

```
ROOT
  : ['[' _] DEL   [']' _] %f1<"DEL" "DEL">
  | ['[' _] MATP  [']' _] %f1<"DEL" "MATP">
  | ['[' _] MATL  [']' _] %f1<"DEL" "MATL">
  | ['[' _] MATR  [']' _] %f1<"DEL" "MATR">
  | ['[' _] INSL  [']' _] %f1<"DEL" "INSL">
  | ['[' _] INSR  [']' _] %f1<"DEL" "INSR">
  ;

MATP
  : ['(' •] DEL   [')' •] %f1<"MATP" "DEL">
  | ['(' •] MATP  [')' •] %f1<"MATP" "MATP">
  | ['(' •] MATL  [')' •] %f1<"MATP" "MATL">
  | ['(' •] MATR  [')' •] %f1<"MATP" "MATR">
  | ['(' •] INSL  [')' •] %f1<"MATP" "INSL">
  | ['(' •] INSR  [')' •] %f1<"MATP" "INSR">
  ;

MATL
  : ['(' •] DEL   [')' _] %f1<"MATL" "DEL">
  | ['(' •] MATP  [')' _] %f1<"MATL" "MATP">
  | ['(' •] MATL  [')' _] %f1<"MATL" "MATL">
  | ['(' •] MATR  [')' _] %f1<"MATL" "MATR">
  | ['(' •] INSL  [')' _] %f1<"MATL" "INSL">
  | ['(' •] INSR  [')' _] %f1<"MATL" "INSR">
  | ['L' •] DEL           %f2<"MATL" "DEL">
  | ['L' •] MATP          %f2<"MATL" "MATP">
  | ['L' •] MATL          %f2<"MATL" "MATL">
  | ['L' •] MATR          %f2<"MATL" "MATR">
  | ['L' •] INSL          %f2<"MATL" "INSL">
  ;

MATR
  : ['(' _] DEL   [')' •] %f1<"MATR" "DEL">
  | ['(' _] MATP  [')' •] %f1<"MATR" "MATP">
  | ['(' _] MATL  [')' •] %f1<"MATR" "MATL">
  | ['(' _] MATR  [')' •] %f1<"MATR" "MATR">
  | ['(' _] INSL  [')' •] %f1<"MATR" "INSL">
  | ['(' _] INSR  [')' •] %f1<"MATR" "INSR">
  | DEL ['R' •]           %f3<"MATR" "DEL">
  | MATP ['R' •]          %f3<"MATR" "MATP">
  | MATL ['R' •]          %f3<"MATR" "MATL">
  | MATR ['R' •]          %f3<"MATR" "MATR">
  | INSR ['R' •]          %f3<"MATR" "INSR">
  ;

INSL
  : [_ •] DEL             %f5<"INSL" "DEL">
  | [_ •] MATP            %f5<"INSL" "MATP">
  | [_ •] MATL            %f5<"INSL" "MATL">
  | [_ •] MATR            %f5<"INSL" "MATR">
  | [_ •] INSL            %f5<"INSL" "INSL">
  | [_ •] INSR            %f5<"INSL" "INSR">
  ;

INSR
  : DEL [_ •]             %f6<"INSR" "DEL">
  | MATP [_ •]            %f6<"INSR" "MATP">
  | MATL [_ •]            %f6<"INSR" "MATL">
  | MATR [_ •]            %f6<"INSR" "MATR">
  | INSR [_ •]            %f6<"INSR" "INSR">
  ;

DEL
  : ['(' _] DEL   [')' _] %f1<"DEL" "DEL">
  | ['(' _] MATP  [')' _] %f1<"DEL" "MATP">
  | ['(' _] MATL  [')' _] %f1<"DEL" "MATL">
  | ['(' _] MATR  [')' _] %f1<"DEL" "MATR">
  | ['(' _] INSL  [')' _] %f1<"DEL" "INSL">
  | ['(' _] INSR  [')' _] %f1<"DEL" "INSR">
  | ['L' _] DEL           %f2<"DEL" "DEL">
  | ['L' _] MATP          %f2<"DEL" "MATP">
  | ['L' _] MATL          %f2<"DEL" "MATL">
  | ['L' _] MATR          %f2<"DEL" "MATR">
  | ['L' _] INSL          %f2<"DEL" "INSL">
  | DEL ['R' _]           %f3<"DEL" "DEL">
  | MATP ['R' _]          %f3<"DEL" "MATP">
  | MATL ['R' _]          %f3<"DEL" "MATL">
  | MATR ['R' _]          %f3<"DEL" "MATR">
  | INSR ['R' _]          %f3<"DEL" "INSR">
  | ['[' _] BEGL ['|' _] BEGR [']' _]
                          { $$.p = $2.p + $4.p; }
  | ['*' _]
                          { $$.p = 0; }
  ;

BEGL
  : MATP                  %f4<"DEL" "MATP">
  | MATL                  %f4<"DEL" "MATL">
  | MATR                  %f4<"DEL" "MATR">
  | DEL                   %f4<"DEL" "DEL">
  ;

BEGR
  : MATP                  %f4<"DEL" "MATP">
  | MATL                  %f4<"DEL" "MATL">
  | MATR                  %f4<"DEL" "MATR">
  | DEL                   %f4<"DEL" "DEL">
  | INSL                  %f4<"DEL" "INSL">
  ;
```

Figure 3.41: Règles de la MTSAG utilisée pour les CM. Les non-terminaux correspondent aux différents états d'un CM. Les règles se contentent d'exprimer les combinaisons des transitions d'un état vers un autre en fonction des tokens-instructions de la première bande. Lorsqu'un symbole doit être lu sur la deuxième bande, un • est mis à la place de _ sur la deuxième bande, en correspondance avec le token-instruction de la première bande.

3.11 MTSAG et points d'ancrage

Dans le meilleur des cas, les complexités de calcul de la structure secondaire d'un ARN de longueur n sont de $O(n^3)$ en temps et $O(n^2)$ en espace, et ce pour le modèle thermodynamique. Mais ce modèle a depuis longtemps révélé ses insuffisances, même si sa formulation grammaticale (cf. p. 58) élimine certains de ses inconvénients. Les meilleurs modèles utilisables à l'heure actuelle pour replier et aligner les ARN ont des complexités allant de $O(n^6)$ en temps et $O(n^4)$ en espace (modèle de Sankoff, p. 95, modèle de Sankoff généralisé, p. 108), à $O(n^4)$ en temps et $O(n^3)$ en espace (modèle de Corpet et Michot p. 117, grammaires stochastiques p. 140, modèles de covariance p. 156). Dans toutes ces complexités, le facteur limitant n'est pas le temps (bien qu'une complexité de $O(n^6)$ en temps soit quelque peu déraisonnable) mais bien l'espace. De nos jours, il n'est plus rare de trouver des stations de travail ayant 256 Moctets de mémoire vive. Une telle capacité permet d'envisager avec sérénité de replier un ARN de deux ou trois mille bases à l'aide du modèle thermodynamique, de complexité quadratique en espace, mais guère plus de 300 bases à l'aide des autres modèles, de complexité au moins cubique en espace.

Comme le fait remarquer à juste titre Grate (1995a), une des méthodes permettant aux modèles de traiter des ARN plus longs consiste à séparer les ARN en sous-séquences de taille plus petite, puis à utiliser l'hypothèse simplificatrice d'absence de pseudo-nœud dans les structures secondaires pour construire la structure globale à partir des structures des sous-séquences. Cette méthode est en fait un exemple typique du principe "diviser pour régner". La décomposition d'une séquence initiale en sous-séquences plus petites est réalisée grâce aux informations d'appariement déjà disponibles à divers endroits de la séquence (figure 3.42). Les informations d'appariement peuvent être obtenues à partir d'alignements multiples préexistants (Chan *et al.*, 1991; Han & Kim, 1993; Eddy & Durbin, 1994) ou d'autres méthodes (Winker *et al.*, 1990). La complexité de ces méthodes de recherche d'appariements est en général cubique en le nombre d'hélices potentielles, ou bien quadratiques avec des heuristiques gloutonnes raisonnables.

Le formalisme des MTSAG permet l'introduction de points d'ancrage dans n'importe quel modèle de manière élémentaire, quel que soit le nombre de bandes utilisées. Nous illustrerons notre propos avec la grammaire 2-bandes utilisée pour le modèle d'alignement et de repliement de Corpet et Michot (cf. section 3.5), dont nous rappelons les règles ci-dessous :

```
Structure  : ['.' •]                    DoubleStrand : ['(' •] Structure [')' •]
           | ['.' _]                                 | ['(' _] Structure [')' •]
           | [_ •]                                   | ['(' •] Structure [')' _]
           | DoubleStrand                            | ['(' _] Structure [')' _]
           | Structure ['.' •]
           | Structure ['.' _]
           | Structure [_ •]
           | Structure DoubleStrand
```

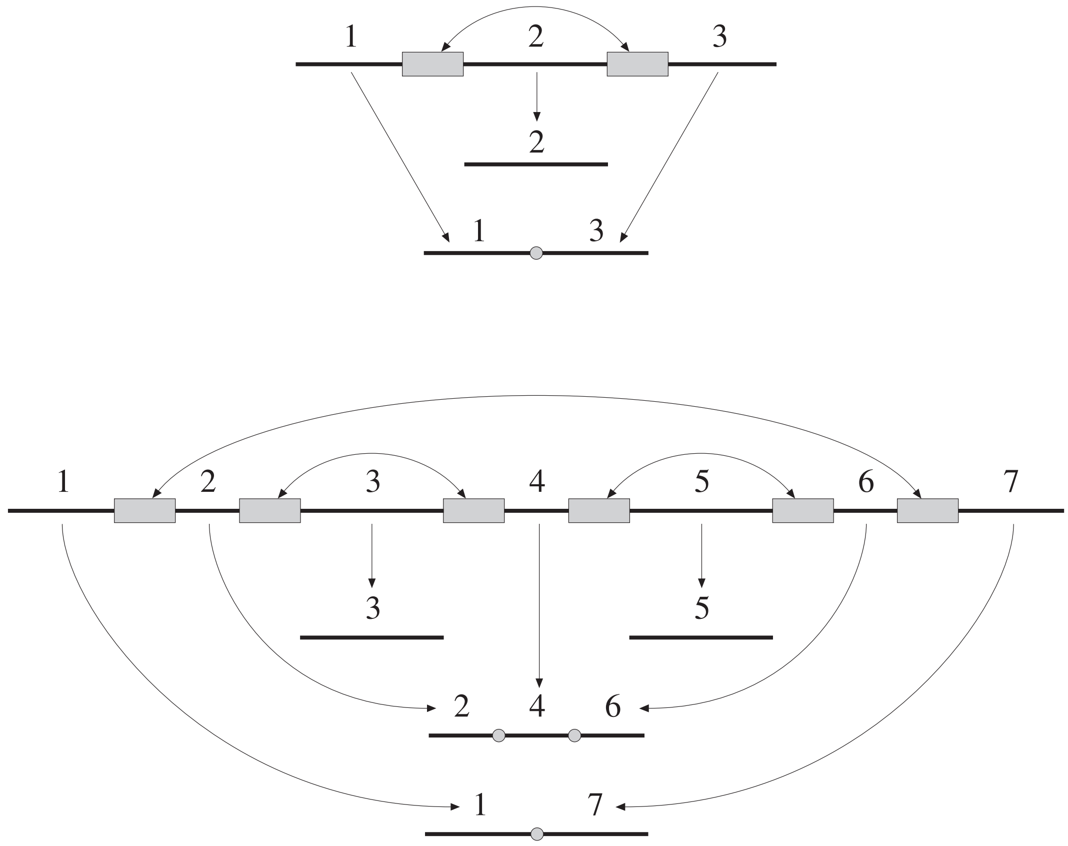


Figure 3.42: Quand l'appariement de deux parties d'une séquence a été établi (rectangles gris du schéma du haut), la région intérieure 2 devient indépendante des régions 1 et 3. Le calcul de la structure secondaire de la région 2 peut donc se faire indépendamment du calcul de la structure secondaire des régions 1 et 3 groupées. Le problème de départ a été décomposé en deux sous problèmes. Sur le schéma du bas, trois hélices ont été trouvées, et les sept sous-séquences ainsi déterminées décomposent le problème de départ en quatre sous-problèmes de tailles plus petites.

Le non-terminal Structure est le non-terminal de départ de cette grammaire. La première bande est un codage particulier d'un ARN modèle dont on connaît déjà la structure et la deuxième bande est l'ARN dont on cherche à déterminer la structure et l'alignement contre le premier ARN. Pour introduire des points d'ancrage dans la MTSAG, il suffit d'utiliser deux tokens spéciaux (`'['` et `']'` par exemple) dans la grammaire et les séquences à aligner. Les points d'ancrage devant obligatoirement être alignés, il faut ajouter la règle suivante à la grammaire :

Anchor : `['[' '[']` Structure `[']' ']']`

Ensuite, il ne reste plus qu'à utiliser le non-terminal Anchor ainsi défini dans la grammaire d'origine en ajoutant les deux règles suivantes :

Structure : Anchor
| Structure Anchor

La grammaire finale, avec ses fonctions d'évaluation d'attributs, est donnée en figure 3.43.

Nous avons comparé les performances de la MTSAG avant et après introduction de points d'ancrage. Alors que la MTSAG d'origine a mis 2 secondes et 4,5 Moctets pour aligner et replier l'exemple de la figure 3.44(a), la MTSAG avec points d'ancrage n'a mis que 0,1 seconde et 0,7 Moctets pour aligner et replier ce même exemple, après introduction de points d'ancrage (figure 3.44(c)). Les gains obtenus sont donc impressionnants. Pourtant les complexités d'analyse au pire définies selon le théorème 2.2 p. 83 n'ont pas changé[19] et sont toujours de $O(mn^3)$ en temps et $O(mn^2)$ en espace, où m est la longueur de la première bande et n la longueur de la deuxième bande. En fait, l'invariance des complexités au pire n'est pas surprenante car la MTSAG avec points d'ancrage est aussi utilisable sur des exemples ne contenant pas de point d'ancrage, avec les mêmes performances théoriques et pratiques que la MTSAG d'origine. L'ajout d'un mécanisme de gestion des points d'ancrages est donc transparent aux séquences n'en contenant pas. L'algorithme GCP au cœur de tous les analyseurs syntaxiques générés par MTSAG2C utilise automatiquement les points d'ancrage éventuellement spécifiés pour délimiter les zones où le repliement et l'alignement peut être effectué indépendamment des autres zones.

Les complexités moyennes d'analyse pour la MTSAG de la figure 3.43 dépendent en fait de la distribution des points d'ancrages dans les séquences analysées. Si le nombre de bases comprises entre deux points d'ancrage est borné quelles que soient les séquences analysées, alors l'analyse dépend linéairement en temps et en espace de la taille des séquences. Ce résultat est une autre illustration de la remarque 2.5 p. 84. En effet, les tokens d'ancrage `[` et `]` doivent apparaître simultanément sur les deux bandes et introduisent donc des zones de synchronisation dans l'algorithme d'analyse syntaxique.

[19] la grammaire projetée sur la première bande est toujours $LR(1)$ non-récursive à droite tandis que la grammaire projetée sur la deuxième bande reste ambiguë

```
%nb_tapes 2

%empty _
%attribute "int" idx
%token A { $$.idx = 0; }
%token C { $$.idx = 1; }
%token G { $$.idx = 2; }
%token U { $$.idx = 3; }
%token '(' ')' '.' '[' ']'

%abbreviation • A C G U

%attribute "int" E
%attribute "short" l r
%terminal_attribs {}
%default_nonterminal_attribs < E l r>

%default_constraint
        {}
        { if ($$.E > $!.E)
            {$!.E = $$.E; $!.l = $$.l; $!.r = $$.r; }
%%

Structure
    : ['.' •]
        { $$.E = MATCH; $$.l = $$.r = 0; }
    | ['.' _]
        { $$.E = GAP_D + EXT_D;
          $$.l = $$.r = 1; }
    | [_ •]
        { $$.E = GAP_I + EXT_I;
          $$.l = $$.r = -1; }
    | DoubleStrand
    | Anchor
        { $$.E = $1.E;
          $$.l = $1.l; $$.r = $1.r; }
    | Structure ['.' •]
        { $$.E = $1.E + MATCH;
          $$.l = $1.l; $$.r = 0; }
    | Structure ['.' _]
        { $$.E = $1.E + EXT_D;
          $$.l = $1.l; $$.r = 1;
          if ($1.r <= 0) $$.E += GAP_D; }
    | Structure [_ •]
        { $$.E = $1.E + EXT_I;
          $$.l = $1.l; $$.r = -1;
          if ($1.r >= 0) $$.E += GAP_I; }
    | Structure DoubleStrand
    | Structure Anchor
        { $$.E = $1.E + $2.E;
          $$.l = $1.l; $$.r = $2.r;
          if ($1.r < 0 && $2.l < 0)
               $$.E += -GAP_I;
          if ($1.r > 0 && $2.l > 0)
               $$.E += -GAP_D; };

Anchor
    : ['(' •] Structure [')' •]
        { $$.E = $2.E;
          $$.l = $2.l; $$.r = $2.r; };

DoubleStrand
    : ['(' •] Structure [')' •]
        { $$.E = $2.E + 2*MATCH +
               PAIR[$1[2].left[0].idx][$3[2].left[0].idx];
          $$.l = $$.r = 0; }
    | ['(' _] Structure [')' •]
        { $$.E = $2.E + MATCH + EXT_D;
          if ($2.l <= 0) $$.E += GAP_D; }
          $$.l = 1; $$.r = 0; }
    | ['(' •] Structure [')' _]
        { $$.E = $2.E + MATCH + EXT_D;
          if ($2.r <= 0) $$.E += GAP_D; }
          $$.l = 0; $$.r = 1; }
    | ['(' _] Structure [')' _]
        { $$.E = $2.E + 2 * EXT_D;
          if ($2.l <= 0) $$.E += GAP_D; }
          if ($2.r <= 0) $$.E += GAP_D; }
          $$.l = 1; $$.r = 1; };
```

Figure 3.43: Cette MTSAG à deux bandes est une modification mineure de la MTSAG à deux bandes de la figure 3.14 p. 121 tenant compte d'éventuels points d'ancrage dans les séquences à aligner et replier.

```
(((((((..((((...........))))((((((.......))))))...(((((.......))))))))))))
GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA
```

(a) Première bande et ARN de 72 bases fournis à la MTSAG 2-bandes originale du modèle de Corpet.

```
(((((((..((((...........))))((((((.......)))))-)...(((((.......))))))))))))
-GGGCCCAUAGCUC----AGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA
```

(b) Alignement des deux bandes précédentes obtenu à l'aide de la MTSAG de la figure 3.14 p. 121.

```
((((((((..[((((...........))))][((((((.......))))))]...[(((((.......)))))]))))))))
GGGCCCAUA[GCUCAGUGGUAGAGU][GCCUCCUUUGCAAGGAGGAU]GCC[CUGGGUUGGAAUCCCAG]UGGGUCCA
```

(c) Première bande et ARN de 72 bases fournis à la MTSAG 2-bandes avec points d'ancrage de la figure 3.43. Des points d'ancrage, indiqués par les tokens spéciaux `[` et `]`, sont insérés aux endroits adéquats.

```
((((((((..[((((...........))))][((((((.......)))))-)]...[(((((.......)))))]))))))))
-GGGCCCAUA[GCUC----AGUGGUAGAGU][GCCUCCUUUGCAAGGAGGAU]GCC[CUGGGUUGGAAUCCCAG]UGGGUCCA
```

(d) Alignement des deux bandes précédentes obtenu à l'aide de la MTSAG de la figure 3.43

Figure 3.44: Les points d'ancrage insérés dans les deux bandes de la figure (c) n'ont pas été choisis au hasard puisque l'alignement obtenu à l'aide de l'algorithme sans point d'ancrage était déjà connu. Le but de cet exemple est de montrer comment fonctionnent les points d'ancrage et quels sont les gains réalisés, et non pas d'introduire une méthode sophistiquée de choix des points d'ancrage.

Les points d'ancrage peuvent être introduits dans la totalité des grammaires 2-bandes, en s'inspirant de l'exemple donné dans cette section. Il suffit de :

- ajouter des tokens spéciaux définissant les points d'ancrage,
- ajouter un ou plusieurs non-terminaux et règles utilisant ces tokens spéciaux avec d'autres non-terminaux de la grammaire d'origine,
- utliser le ou les non-terminaux ajoutés dans des règles supplémentaires.

Bien sûr, pour que l'ajout d'une gestion des points d'ancrage dans une grammaire soit réellement utile, il faut également introduire des points d'ancrage dans les chaînes analysées.

Cette manière de procéder est beaucoup plus naturelle et souple que les démarches externes au modèle adoptées par Corpet et Michot (Corpet & Michot, 1994) ou Grate

(Grate, 1995a) : les points d'ancrage sont intégrés à la grammaire et directement utilisés s'il y a lieu par l'analyseur syntaxique, sans avoir à découper artificiellement les séquences. Ainsi, il suffit d'ajouter les règles suivantes à la MTSAG 2-bandes du modèle de Sankoff de la p. 104 pour y introduire des points d'ancrage :

Anchor	:	`['['  '[']` Loop `[']'  ']']`
Structure	:	Anchor | Structure Anchor
Index2	:	Anchor
Index2Insert	:	Anchor
Index2Delete	:	Anchor
LoopSeq	:	Anchor | LoopSeq Anchor

Ici, le non-terminal Anchor se comporte sensiblement comme le non-terminal Loop.

3.12 Alignements et repliements fiables des ARN

Nous proposons dans cette section une méthode permettant d'assurer une certaine fiabilité dans les alignements et les repliements d'ARN comportant plusieurs milliers de base. Cette méthode n'a pas encore été implémentée ni testée par manque de temps et/ou de ressources.

Dans le contexte du repliement et de l'alignement de séquences d'ARN initialement non alignées, les discussions des sections précédentes nous conduisent à la récapitulation suivante :

- les ARN ne peuvent être alignés avec précision indépendamment de leur structure. Les méthodes classiques utilisées pour aligner des protéines ou des séquences d'ADN ne sont donc que partiellement efficaces ;
- les méthodes de recherche d'une structure secondaire d'un ARN isolé ne peuvent, dans l'état actuel de la technique, garantir la correction du résultat fourni. Les méthodes fondées sur des grammaires stochastiques, ou sur n'importe quel autre codage de la structure cherchée, ne constituent pas une solution recevable dans ce contexte puisqu'elles nécessitent une préconnaissance de la structure. Elles sont trop coûteuses en l'état lorsque les ARN dépassent une centaine de bases ;
- l'analyse phylogénétique comparative des ARN (c'est-à-dire leur comparaison par le biais d'alignements) reste la meilleure manière de garantir un alignement et un repliement correct des ARN. Elle nécessite cependant de disposer d'au moins une dizaine d'ARN et n'est pas encore devenue un processus entièrement automatique utilisable sur de longs ARN ;
- les méthodes d'alignement et de repliement simultanés produisent des résultats remarquables avec seulement deux ARN mais sont limitées à des ARN de petite taille (moins de cent bases) du fait de leurs complexités en temps et en espace ;
- l'introduction de points d'ancrages dans les MTSAG est un processus simple permettant de réduire fortement les ordres de grandeur des complexités d'analyse puisqu'elles deviennent linéaires en temps et en espace si les zones délimitées par les points d'ancrage sont de taille bornée.

Nous proposons de combiner le meilleur de chaque méthode pour réussir à automatiser totalement le processus habituel d'analyse comparative d'un ensemble $R_1, \ldots, R_p$ de p ARN homologues. La stratégie envisagée est simple :

1. Construction d'un alignement multiple préliminaire ;
2. Recherche de points d'ancrage dans l'alignement multiple ;
3. Alignement et repliement simultanés des zones de petite taille (< 40 bases) en procédant par paires d'ARN ;
4. Détermination d'une structure consensus ;
5. Utilisation de la structure consensus et des points d'ancrage pour former une première bande contre laquelle les ARN sont alignés ;
6. Réitération des étapes 2 à 6 avec l'alignement multiple obtenu ;

Nous allons maintenant commenter plus avant chacune des étapes précédentes.

Construction d'un alignement initial De nombreuses méthodes sont utilisables (cf. références de la section 3.1 ou (Abdeddaim, 1996)). Cependant, une méthode facilitant la détermination de points d'ancrage (étape suivante) est préférable. Il n'est pas nécessaire de garantir l'optimalité de l'alignement multiple obtenu car cet alignement sera de toute façon retouché.

Recherche de points d'ancrage Il existe deux catégories de points d'ancrage. La première est celle des zones dont la probabilité d'alignement correct est forte du fait de leur grande taille et de leur degré de conservation d'un ARN à l'autre dans l'alignement multiple. Il n'est pas toujours possible de savoir quelles sont les bases de ces zones impliquées dans un appariement. La deuxième catégorie est celle des couples de bases dont la probabilité d'appariement est forte du fait du degré de covariation dans les colonnes correspondantes de l'alignement (cf. (Grate, 1995a)). Ces deux catégories ne seront typiquement pas représentées par les mêmes tokens spéciaux de points d'ancrage : le token `|` pourra délimiter les zones de la première catégorie, tandis que les tokens `[` et `]` délimiteront les couples de la deuxième catégorie (cf. section précédente). Cette étape de recherche peut prendre une vingtaine de minutes sur un ARN 23S en utilisant un ordinateur massivement parallèle (Grate, 1995a).

Alignement et repliement simultanés La 2-grammaire obtenue en section 3.4 peut être utilisée sur les $p-1$ paires (R_i, R_{i+1}). Le succès de cette étape nécessite la séparation des ARN en zones de petite taille (< 30 bases) par les points d'ancrages obtenus lors de l'étape précédente[20]. Les zones de taille plus importante pourront être abandonnées dans un premier temps grâce à l'introduction de contraintes

[20] Au besoin, l'algorithme de recherche de points d'ancrage pourra être autorisé à diminuer ses seuils de validité des points d'ancrage, quitte à introduire quelques faux positifs.

de sélection idoines. La quantité de mémoire nécessaire au calcul du repliement pourrait cependant atteindre 800 Moctets pour une paire d'ARN 23S.

Détermination d'une structure consensus Une version modifiée de la grammaire de la figure 3.16 p. 128 tenant compte des points d'ancrage permet de calculer en $p-1$ phases la plus courte structure secondaire commune à toutes les structures trouvées lors de l'étape précédente.

Alignement contre la structure consensus La structure consensus précédente est alignée contre chacun des p ARN. L'alignement multiple obtenu est alors le nouvel alignement de référence grâce auquel les étapes précédentes peuvent être réitérées.

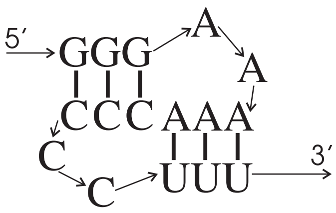
La pierre d'achoppement principale de l'enchaînement précédent réside dans la détermination d'un nombre suffisants de points d'ancrage dans les ARN. Il est nécessaire de vérifier expérimentalement qu'une telle détermination est réaliste pour valider cette méthode. Notons que la recherche fiable de points d'ancrages semble nécessiter un nombre minimum d'ARN. La question de l'alignement et du repliement simultanés d'un petit nombre (< 10) de longs ARN reste donc ouverte.

3.13 Recherche des pseudo-nœuds dans les ARN

Les pseudo-nœuds sont une classe importante de structures d'ARN rencontrées dans virtuellement tous les ARN (van Belkum *et al.*, 1985). Ils sont essentiels à de nombreuses fonctions biologiques (Florentz *et al.*, 1984; Le *et al.*, 1991; ten Dam *et al.*, 1992; Wang *et al.*, 1995; Pleij, 1995). Jusqu'à un passé récent, la plupart des méthodes de repliement d'ARN ont contourné le problème de la détermination des pseudo-nœuds dans les structures secondaires en prétextant que cette détermination faisait partie de l'étude de la structure tertiaire des ARN. Quelques avancées récentes laissent cependant entrevoir des solutions ne requérant pas des temps de calcul prohibitifs. Cary & Stormo (1995) proposent de chercher un "maximum matching" dans un graphe des appariements, mais ils présupposent l'existence d'un alignement parfait, qui ne peut être obtenu que si les pseudo-nœuds sont connus. Brown (1995) montre comment appliquer l'intersection de grammaires non-contextuelles à la recherche de pseudo-nœuds, à condition de déjà connaître la structure de ces pseudo-nœuds. Cette dernière solution n'est pas sans rappeler celle, plus générique, que nous allons proposer dans cette section.

La raison fondamentale pour laquelle les pseudo-nœuds n'ont pas été traités par la plupart des méthodes de repliement connues à ce jour est que les pseudo-nœuds ne forment pas des structures non-contextuelles. Prenons par exemple la structure secondaire de la figure 3.45. Cette structure est représentée abstraitement par le langage L suivant :

$$L = \{ ({}^i u [{}^j v)^i w]^j \mid i > 0, j > 0, \text{ et } u, v, w \text{ sont des mots d'un alphabet quelconque } \}.$$




```
GGGAAAAACCCCCUUU
( ( (     [ [ [ ) ) )     ] ] ]
```

Figure 3.45: En haut, représentation d'un pseudo-nœud. En bas, alignement de la séquence d'ARN et de l'expression parenthésée correspondant aux appariements. Les parenthèses et crochets de l'expression ne sont pas correctement imbriqués.

La notation $(^i$ représente le symbole (répété i fois. Ce langage n'est pas non-contextuel d'après le lemme de Bar-Hillel. Cela veut dire qu'aucune grammaire non-contextuelle ne peut générer L. En revanche, le langage L précédent est l'intersection des langages non-contextuels L_1 et L_2 suivants :

$$L_1 = \{(^* u[^j v)^* w]^j \mid j > 0, \text{ et } u, v, w \text{ sont des mots d'un alphabet quelconque } \}$$
$$L_2 = \{(^i u[^* v)^i w]^* \mid i > 0, \text{ et } u, v, w \text{ sont des mots d'un alphabet quelconque } \}$$

où $(^*$ représente le symbole (répété un nombre quelconque de fois (y compris zéro). Cette propriété est utilisée par Brown (1995) pour trouver un pseudo-nœud dont la structure est fixée à l'aide de deux SCFG (une première SCFG pour L_1 et une deuxième SCFG pour L_2) et de quatre analyses syntaxiques.

Nous allons voir que le même résultat peut être retrouvé à l'aide de grammaires 2-bandes, ce qui est une illustration du théorème 5.5 p. 221. En effet, soit L' le langage 2-bandes suivant :

$$L' = \Big\{ \begin{bmatrix} (^i u[^j v)^* w]^j \\ (^i u[^* v')^i w]^j \end{bmatrix} \;\Big|\; i > 0, j > 0, \text{ et } u, v, v', w \text{ sont des mots d'un alphabet 1-bande quelconque } \Big\}$$

L' est l'ϵ-suppression de l'ensemble d'alignements 2-bandes L'' suivant :

$$L'' = \Big\{ \begin{bmatrix} (\\ (\end{bmatrix}^i \begin{bmatrix} u \\ u \end{bmatrix} \begin{bmatrix} - \\ [\end{bmatrix}^* \begin{bmatrix} - \\ v' \end{bmatrix} \begin{bmatrix} - \\ ) \end{bmatrix}^i \begin{bmatrix} [\\ - \end{bmatrix}^j \begin{bmatrix} v \\ - \end{bmatrix} \begin{bmatrix}) \\ - \end{bmatrix}^* \begin{bmatrix} w \\ w \end{bmatrix} \begin{bmatrix}] \\ ] \end{bmatrix}^j \;\Big|\; i > 0, j > 0, \text{ et } u, v, v', w \text{ sont des mots d'un alphabet 1-bande quelconque } \Big\}$$

Or L'' est généré par la grammaire non-contextuelle 2-bandes[21] de la figure 3.46.

[21] Cette grammaire est volontairement non ϵ-free pour en aérer la présentation, mais elle est trivialement transformable en une grammaire équivalente et ϵ-free.

$$
\begin{array}{ll}
S \to A_0 B_0 & \\
A_0 \to \left[\genfrac{}{}{0pt}{}{(}{(}\right] A_0 \left[\genfrac{}{}{0pt}{}{-}{)}\right] \mid A_1 A_2 A_3 & \qquad B_0 \to \left[\genfrac{}{}{0pt}{}{[}{-}\right] B_0 \left[\genfrac{}{}{0pt}{}{]}{]}\right] \mid B_1 B_2 B_3 \\
A_1 \to A_1 [\,:\,] \mid \epsilon & \qquad B_1 \to B_1 \left[\genfrac{}{}{0pt}{}{.}{-}\right] \mid \epsilon \\
A_2 \to A_2 \left[\genfrac{}{}{0pt}{}{-}{[}\right] \mid \epsilon & \qquad B_2 \to B_2 \left[\genfrac{}{}{0pt}{}{)}{-}\right] \mid \epsilon \\
A_3 \to A_3 \left[\genfrac{}{}{0pt}{}{-}{.}\right] \mid \epsilon & \qquad B_3 \to B_3 [\,:\,] \mid \epsilon
\end{array}
$$

Figure 3.46: Grammaires 2-bandes produisant des alignements potentiellement utilisables pour déterminer des pseudo-nœuds.

$$
\begin{array}{l}
A \to B \mid E \mid A\,B \mid B\,A \mid A\,E \mid E\,A \mid A \bullet \mid \bullet A \mid \bullet A \bullet \\
B \to \left[\genfrac{}{}{0pt}{}{(}{(=1}\right] C \left[\genfrac{}{}{0pt}{}{)}{)=1}\right] \mid \left[\genfrac{}{}{0pt}{}{(}{(=1}\right] D \left[\genfrac{}{}{0pt}{}{)}{)=1}\right] \mid \left[\genfrac{}{}{0pt}{}{(}{(=1}\right] E \left[\genfrac{}{}{0pt}{}{)}{)=1}\right] \\
C \to B \mid C \mid C\,E \mid E\,C \mid C \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] \mid \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] C \mid \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] C \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] \\
D \to B \mid E \mid D\,B \mid B\,D \mid D\,E \mid E\,D \mid D \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] \mid \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] D \mid \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] D \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] \\
E \to E\,E \mid E \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] \mid \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] E \mid \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right] E \left[\genfrac{}{}{0pt}{}{\bullet}{\bullet=1}\right]
\end{array}
$$

Figure 3.47: Grammaire 2-bandes dérivée de la grammaire 1-bande du modèle thermodynamique de Zuker. Pour conserver une figure de taille raisonnable, les non-terminaux ont été abrégés : A pour Structure, B pour Loop, C pour Index2, D pour LoopSeq, E pour SingleStrand. La notation $= 1$ utilisée dans les terminaux 2-bandes signifie que le token de la deuxième bande est égal au token de la première bande (notation de MTSAG2C).

Donc L' est un langage non-contextuel 2-bandes reconnu par un analyseur 2-bandes généré par MTSAG2C. Maintenant, si les deux bandes de L' sont contraintes à être les mêmes, le langage reconnu par l'analyseur devient précisément égal à L sur chaque bande. Nous allons appliquer le principe d'alignement décrit par la grammaire précédente pour tenter de trouver des pseudo-nœuds dans une structure secondaire d'ARN.

La grammaire de la figure 3.47 est une version 2-bandes de la grammaire déjà utilisée pour la généralisation du modèle de Sankoff (cf. p. 113). Cette version 2-bandes est rigoureusement équivalente à la version 1-bande puisque les dérivations doivent être synchronisées sur les deux bandes. L'analyseur 2-bandes pour cette grammaire ne fonctionne donc que s'il est appliqué à des couples de bandes identiques. La grammaire contient un non-terminal E représentant les portions non appariées. Il suffit de déclarer, en substance, que le non-terminal E peut être composé d'une séquence de S (où S est l'axiome de la grammaire de la figure 3.46) pour ajouter une capacité de détermination de pseudo-nœuds à la grammaire. Les fonctions d'évaluation d'attributs utilisées pour les nouvelles règles sont directement déduites de celles appliquées par le modèle thermodynamique. Nous appellerons **pseudo-Zuker** la grammaire S-attribuée 2-bandes obtenue (figure 3.48). Notons que, du fait des contraintes de synchronisation présentes d'une bande à l'autre, la complexité d'analyse suivant la grammaire obtenue est encore $O(n^3)$ en temps et $O(n^2)$ en espace.

$$
\begin{aligned}
A &\rightarrow B \mid E \mid A\,B \mid B\,A \mid A\,E \mid E\,A \mid A\bullet \mid \bullet A \mid \bullet A \bullet\\
B &\rightarrow \left[{}^{(}_{(=1}\right] C \left[{}^{)}_{)=1}\right] \mid \left[{}^{(}_{(=1}\right] D \left[{}^{)}_{)=1}\right] \mid \left[{}^{(}_{(=1}\right] E \left[{}^{)}_{)=1}\right]\\
C &\rightarrow B \mid C \mid C\,E \mid E\,C \mid C \left[{}^{\bullet}_{\bullet=1}\right] \mid \left[{}^{\bullet}_{\bullet=1}\right] C \mid \left[{}^{\bullet}_{\bullet=1}\right] C \left[{}^{\bullet}_{\bullet=1}\right]\\
D &\rightarrow B \mid E \mid D\,B \mid B\,D \mid D\,E \mid E\,D \mid D \left[{}^{\bullet}_{\bullet=1}\right] \mid \left[{}^{\bullet}_{\bullet=1}\right] D \mid \left[{}^{\bullet}_{\bullet=1}\right] D \left[{}^{\bullet}_{\bullet=1}\right]\\
E &\rightarrow E\,E \mid E \left[{}^{\bullet}_{\bullet=1}\right] \mid \left[{}^{\bullet}_{\bullet=1}\right] E \mid \left[{}^{\bullet}_{\bullet=1}\right] E \left[{}^{\bullet}_{\bullet=1}\right]\\
E &\rightarrow S\\
S &\rightarrow A_0 B_0\\
A_0 &\rightarrow \left[{}^{(}_{(}\right] A_0 \left[{}^{-}_{)}\right] \mid A_1 A_2 A_3\\
A_1 &\rightarrow A_1 \left[{}^{\bullet}_{\bullet=1}\right] \mid \epsilon\\
A_2 &\rightarrow A_2 \left[{}^{-}_{\bullet}\right] \mid \epsilon\\
A_3 &\rightarrow A_3 \left[{}^{-}_{\bullet}\right] \mid \epsilon\\
B_0 &\rightarrow \left[{}^{(}_{-}\right] B_0 \left[{}^{)}_{)}\right] \mid B_1 B_2 B_3\\
B_1 &\rightarrow B_1 \left[{}^{\bullet}_{-}\right] \mid \epsilon\\
B_2 &\rightarrow B_2 \left[{}^{\bullet}_{-}\right] \mid \epsilon\\
B_3 &\rightarrow B_3 \left[{}^{\bullet}_{\bullet=1}\right] \mid \epsilon
\end{aligned}
$$

Figure 3.48: Grammaire 2-bandes utilisable pour rechercher des pseudo-nœuds dans les ARN.

Nous avons utilisé cette grammaire pour tenter de déterminer les trois pseudo-nœuds de deux fragments d'ARN situés à l'extrémité 3' de deux virus mosaïques du tabac. Les alignements et pseudo-nœuds de ces deux fragments sont (van Belkum *et al.*, 1987; Garcia-Arenal, 1988) :

```
UAACGGA UUGUGUCCGU AAUCACA CGUGGUGCGUACGAUAACGCAUAGUGUUUUUCCCUCCACUUAAAUCGAAGGG
  (((((  [[[[)))))    ]]]] [[[ ((((((]]]   ))))))[[[[     (((( ]]]]        ))))
UAAAGUCGCUGAAGACUUAAAAUUCAGGGUGGCUGAUACCAAAAUCAGCAGUGGUUGUUCGUCCACUUAAAUAUAACGA
  ((((( [[[[[)))))    ]]]]][[[ ((((((]]]   ))))))[[[[[    ((((]]]]]        ))))
```

Les appariements ont été représentés à l'aide de parenthèses et de crochets afin de mieux visualiser les pseudo-nœuds. Les pseudo-nœuds obtenus à l'aide de la grammaire pseudo-Zuker furent, pour le premier fragment d'ARN :

```
UAACGGAUUGUGUCCGUAAUCACACGUGGUGCGUACGAUAACGCAUAGUGUUUUUCCCUCCACUUAAAUCGAAGGG
  ((((((    ))))))        (((((((((     )))))              ))))
        ((((        ))))                (((((   )))))   ((((            ))))
           ↑                            ↑↑↑↑
```

et pour le second :

```
UAAAGUCGCUGAAGACUUAAAAUUCAGGGUGGCUGAUACCAAAAUCAGCAGUGGUUGUUCGUCCACUUAAAUAUAACGA
  (((((      )))))         ((((     ))))         (((((        )))))
        (((((         )))))    ((((((      ))))))         ((((             ))))
                                    ↑
```

Cette fois-ci, les pseudo-nœuds ont été représentés sous la forme d'appariements séparés sur deux bandes. Comme nous pouvons le constater, tous les pseudo-nœuds ont été déterminés dans le second ARN, mais un seul pseudo-nœud a été trouvé dans le premier ARN. Nous pouvons également constater que certaines bases (marquées par une flèche ↑) sont appariées avec deux autres positions, ce qui est normalement interdit. Dans un cas typique d'appariement triple, le non-terminal S de la grammaire de la figure 3.48 produit en réalité un alignement tel que le suivant :

```
ACGGAU---------UGUGUCCGUAAUCACA
               ((((        ))))
ACGGAUUGUGUCCGU---------AAUCACA
((((((    ))))))
```

Cet alignement est parfaitement conforme aux règles de la grammaire et pourtant une des bases est appariée sur les deux bandes, ce qu'il faudrait normalement interdire. Les appariements triples s'expliquent par le fait que le non-terminal B_0 de la grammaire de la figure 3.48 n'a pas connaissance des appariements trouvés par le non-terminal A_0. Un tel comportement était impossible avec la grammaire L'' car les crochets et parenthèses des appariements étaient clairement distingués. Il serait théoriquement possible d'introduire des contraintes d'alignement dans la grammaire de la figure 3.48 sous la forme de bandes supplémentaires afin d'éviter ces appariements intempestifs.

La grammaire alors obtenue aurait alors 4 bandes et serait inutilisable avec des ARN de plus d'une vingtaine de bases du fait des trop grandes complexités d'analyse en temps, et surtout en espace.

Pour retrouver des pseudo-nœuds corrects sans augmenter trop fortement les complexités, il nous faut procéder autrement. Nous allons donc commencer par aligner et replier simultanément les deux fragments d'ARN à l'aide du modèle de Sankoff généralisé de la section 3.4 p. 114. L'alignement et les structures obtenus après cette première étape sont :

```
UAACGGA UUGUGUCCGU  AAUCACACGUGGUGCGUACGAUAACGCAUAGUGUUUUUCCCUCCACUUAAAUCGAAGGG
  (((((      )))))             ((((((      ))))))         ((((             ))))
UAAAGUCGCUGAAGACUUAAAAUUCAGGGUGGCUGAUACCAAAAUCAGCAGUGGUUGUUCGUCCACUUAAAUAUAACGA
  (((((      )))))             ((((((      ))))))         ((((             ))))
```

Remarquons une fois encore l'intérêt de l'alignement et du repliement simultanés de deux ARN puisque l'alignement et les structures trouvés sont parfaits, à l'exception des pseudo-nœuds, non pris en compte par la grammaire du modèle de Sankoff généralisé.

Maintenant qu'une première série d'appariement a été fixée, nous pouvons de nouveau chercher des pseudo-nœuds en modifiant la grammaire de la figure 3.48 afin qu'une base déjà appariée et marquée comme telle ne soit pas appariée à une autre base que celle déjà déterminée. Les pseudo-nœuds trouvés sur chacun des fragments d'ARN sont alors, pour le premier fragment :

```
UAACGGAUUGUGUCCGUAAUCACACGUGGUGCGUACGAUAACGCAUAGUGUUUUUCCCUCCACUUAAAUCGAAGGG
  (((((     )))))       (((       )))         ((((          ))))
        ((((        ))))    ((((((      ))))))         ((((             ))))
```

et pour le second fragment :

```
UAAAGUCGCUGAAGACUUAAAAUUCAGGGUGGCUGAUACCAAAAUCAGCAGUGGUUGUUCGUCCACUUAAAUAUAACGA
  (((((      )))))         (((       )))         (((((        )))))
        (((((         )))))    ((((((      ))))))         ((((             ))))
```

Cette fois, tous les pseudo-nœuds ont été parfaitement trouvés.

3.14 Conclusion

Nous espérons avoir prouvé tout au long de ce chapitre que les MTSAG et MTSAG2C offrent d'une part un cadre de travail agréable permettant d'exprimer simplement la plupart des modèles existants d'alignement et/ou de repliement de séquences biologiques, et laissent espérer d'autre part la création rapide de nouveaux modèles plus performants. Par ailleurs, les MTSAG 2-bandes autorisent une nouvelle exposition du modèle stochastique et montrent clairement les similitudes existant entre les modèles de covariance de Eddy et Durbin et les grammaires stochastiques de Haussler. Les MTSAG facilitent également l'introduction de mécanismes généraux tels que les contraintes d'alignement ou les points d'ancrage dans un modèle quelconque.

La formulation originelle de la majorité des modèles existants présentés dans ce chapitre utilise des équations de programmation dynamique. Dans les cas les plus élémentaires (alignement par distance d'édition, HMM), la simplicité des équations de programmation dynamique permet encore à l'homme de tenir tête au générateur automatique d'analyseurs syntaxiques ; ce n'est plus le cas dès que la complexité des équations, et donc du modèle, croît. Une MTSAG reste toujours plus facile à comprendre et à écrire que des équations de programmation dynamique puisque la structure du modèle sous-jacent n'est jamais perdue de vue. Il faut cependant tempérer notre enthousiasme en remarquant que la communauté des bio-informaticiens est habituée aux équations de programmation dynamique, mais beaucoup moins aux grammaires, et encore moins aux MTSAG. Que dire de la communauté des biologistes non informaticiens ! Il est donc préférable d'affirmer qu'une MTSAG est plus facile à comprendre que des équations de programmation dynamique dès l'instant où les deux formalismes sont identiquement connus[22].

Fondamentalement, l'analyse syntaxique suivant l'algorithme GCP m-bandes est un processus de type programmation dynamique, optimisé pour tenir compte d'une notion de filtrage top-down (cf. p. 107). Les MTSAG ne sont donc pas formellement plus puissantes que les équations de programmation dynamique. Cela explique que nous ayons pu constater que, dans de nombreux cas, les complexités en temps et en espace de l'analyseur syntaxique automatiquement généré à partir de la MTSAG d'un modèle sont égales aux complexités en temps et en espace de l'algorithme écrit manuellement à partir des équations de programmation dynamique du modèle. Cependant, nous avons rencontré des modèles (Corpet et Michot p. 117, Bafna *et al.* p. 126) suffisamment complexes pour que leurs auteurs ne réussissent pas à écrire ou implémenter des équations de programmation dynamique exprimant leur modèle et ayant une complexité asymptotique aussi bonne que celle de l'analyseur syntaxique. Ceci montre bien que la puissance théorique d'un formalisme n'est pas tout et que sa souplesse et son expressivité doivent entrer en ligne de compte lors de son évaluation. Les MTSAG combinées à l'algorithme GCP implémenté dans MTSAG2C offrent à l'heure actuelle le meilleur compromis performance/puissance/souplesse. En outre, toute amélioration apportée à l'algorithme d'analyse syntaxique ou à MTSAG2C profite immédiatement à toutes les MTSAG, modulo une simple recompilation de l'analyseur syntaxique.

[22] Un individu ne connaissant ni le langage C ni l'assembleur aurait bien du mal à affirmer que le langage C est supérieur à l'assembleur, surtout à la vue de certains programmes écrits dans le cadre de l'obfuscated C contest (concours du programme C le plus illisible).

Chapitre 4

Implémentation de MTSAG2C

La présentation des différentes étapes de l'algorithme GCP m-bandes donnée dans le chapitre 2 est relativement courte car elle tient en moins d'une page (rappel en figure 4.1). Compte tenu de cette brièveté, elle n'explique pas comment créer et gérer efficacement en pratique les états attribués ou les ensembles de non-terminaux attendus. Elle ne met pas non plus en évidence les opérations indépendantes de la MTSAG, et qui peuvent donc être pré-compilées sous forme de tables d'analyse dans l'analyseur syntaxique. Enfin, elle cache un certain nombre de problèmes importants. Pourtant, toute implémentation réussie de l'algorithme GCP se doit de traiter avec efficacité les aspects que nous venons d'évoquer. En effet, il est évident que, pour intéresser des biologistes ou des bio-informaticiens au formalisme grammatical, il faut prouver que les grammaires peuvent conduire *automatiquement* à des programmes rapides, voire même plus rapides que leurs propres programmes à base d'équations de programmation dynamique. Une telle preuve ne peut être apportée qu'à la condition de disposer d'un générateur automatique d'analyseurs syntaxiques performants. Ce générateur existe et c'est MTSAG2C. Il a pu, grâce aux différentes applications présentées au chapitre 3, fournir la preuve du succès et de l'intérêt du formalisme grammatical. Ce succès n'a pas été sans mal et un long chemin sépare l'algorithme GCP m-bandes de la figure 4.1 de l'implémentation actuelle de MTSAG2C. Ce chapitre a pour but de développer quelques aspects techniques importants et/ou instructifs qui n'avaient pas leur place aux chapitres 1 et 2, et non pas d'exposer l'ensemble des choix d'implémentation et d'optimisation présents dans MTSAG2C car cet exposé serait fastidieux et ferait double emploi avec les 250 pages de documentation détaillée[1] du source de MTSAG2C (Lefebvre, 1996b). Nous n'entrerons évidemment pas dans les détails de la programmation C de MTSAG2C.

[1]L'ensemble des fichiers de MTSAG2C, c'est-à-dire aussi bien la grammaire YACC de la syntaxe des MTSAG que le noyau des analyseurs syntaxiques, ont été écrits à l'aide d'un outil de programmation littéraire (Knuth, 1992) appelé noweb (Ramsey, 1994). Cet outil permet de mélanger librement les commentaires LaTeX et le source C en suivant de préférence la structure logique d'un exposé plutôt que la structure d'un programme C. Des programmes spécialisés permettent ensuite de générer soit un programme C compilable, soit un source LaTeX, soit un source HTML. L'usage du concept de programmation littéraire fut déterminant pour la réalisation d'un code C efficace et clair.

Soit Q une file de priorité de triplets $(X, \vec{\imath}, x)$ et sa fonction d'extraction $\mathtt{Extract}$. Au départ, les listes d'analyse m-bandes $T_{\vec{\jmath}}$ sont vides. Soit $\Delta_{\vec{0}} = \{A \in V_N \mid \exists\beta, S \to^* A\beta\}$. Pour $\vec{\jmath}$ allant de $\vec{0}$ (exclus) à $\vec{n}$ (inclus), en ordre croissant d'index m-bandes, exécuter les étapes suivantes :

1. Initialiser la file Q: $Q := \left\{(a, \vec{\imath}, S_A(a)) \,\middle|\, \vec{\imath} < \vec{\jmath} \wedge a = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}} \in \vec{V}_T\right\}$
2. Extraction du triplet à traiter en priorité :
   $(X, \vec{\imath}, x) := \mathtt{Extract}(Q)$
3. Création des états attribués commençant par X :
   $T_{\vec{\jmath}} := T_{\vec{\jmath}} \cup \left\{[\Delta \rightarrowtail X, \vec{\imath}, x] \,\middle|\, [\Delta \rightarrowtail X] = \mathtt{Initial}(\Delta_{\vec{\imath}}, X)\right\}$
4. Prolongement d'états attribués par X :
   $T_{\vec{\jmath}} := T_{\vec{\jmath}} \cup \left\{[\Delta \rightarrowtail \alpha X, \vec{h}, \lambda x] \,\middle|\, \exists[\Delta' \rightarrowtail \alpha, \vec{h}, \lambda] \in T_{\vec{\imath}}, [\Delta \rightarrowtail \alpha X] = \mathtt{Goto}([\Delta' \rightarrowtail \alpha], X)\right\}$
5. Pour chaque état attribué $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ ajouté à $T_{\vec{\jmath}}$ par une des étapes 3 et 4 ci-dessus, et pour chaque $A \in \mathtt{Reduce}([\Delta \rightarrowtail \alpha])$, calculer le triplet $(A, \vec{\imath}, x) := (A, \vec{\imath}, f_{A\to\alpha}(\lambda))$.
   Si $\mathcal{C}^{\mathcal{S}}_A(\vec{\imath}, \vec{\jmath}, x)$ est faux alors passer directement à l'étape 6.
   S'il existe déjà un triplet $(A, \vec{\imath}, x')$ appartenant à Q, remplacer l'attribut x' de ce triplet par $\mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x, x')$.
   Si aucun triplet de ce type n'existe dans Q, ajouter $(A, \vec{\imath}, x)$ à Q.
6. Répéter les étapes 2 à 6 tant que Q n'est pas vide;
7. Calculer $\Delta_{\vec{\jmath}} := \mathtt{Expect}(T_{\vec{\jmath}})$.

Le mot m-bandes $a_{\vec{1}} \dots a_{\vec{n}}$ appartient à $L(\vec{G})$ si et seulement si le triplet $(S, \vec{0}, x)$ a été extrait de Q pendant le calcul de $T_{\vec{n}}$. On dit alors que le mot a été reconnu par l'algorithme et qu'il a un attribut optimal x.

Figure 4.1: Rappel de l'algorithme GCP m-bandes

Le générateur d'analyseurs syntaxiques MTSAG2C est en réalité composé de deux parties (figure 4.2) :

- un programme MTSAG2C chargé de lire la MTSAG, convertir les fonctions d'évaluation d'attributs de la MTSAG en fonctions d'évaluation d'attribut pour l'analyseur, et générer des tables d'analyse (mode statique) ou compacter la MTSAG (mode dynamique),

- un noyau C commun à tous les analyseurs et implémentant l'algorithme GCP proprement dit. Ce noyau utilise les tables d'analyses et les fonctions C générées par MTSAG2C (mode statique) ou génère lui même ses propres tables à partir de la MTSAG compactée (mode dynamique). Le noyau est recopié sans modification dans le source de tout analyseur généré par MTSAG2C.

Le cœur du problème posé par le noyau commun à tous les analyseurs réside dans la difficulté d'obtenir de bonnes performances en temps et en espace pour la classe de MTSAG la plus large possible. Nous avons déjà pu constater que les analyseurs

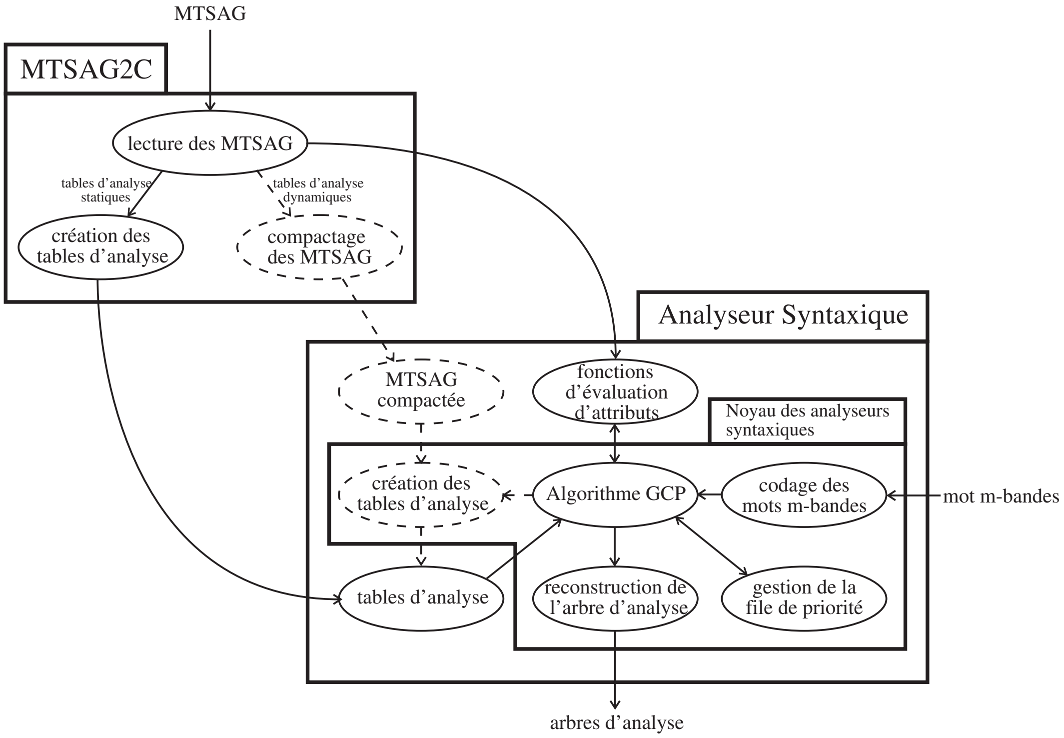


Figure 4.2: Éclaté de MTSAG2C et des analyseurs syntaxiques générés par MTSAG2C. Les éléments les plus intéressants de cet éclaté sont détaillés dans ce chapitre.

générés par MTSAG2C sont peu performants lorsque les MTSAG sont non-ambiguës (Comparaison YACC–GCP dans la section 1.7.1 p. 49) ou lorsqu'elles ne possèdent que des projections non-ambiguës (modèles d'alignement de séquences par distance d'édition section 3.1 p. 86 ou HMM section 3.8 p. 133). En revanche, les performances sont au rendez-vous dès qu'il s'agit de traiter des grammaires suffisamment ambiguës. Cette classe est assez vaste et possède assez d'applications importantes pour conserver tout leur intérêt aux MTSAG.

Les problèmes que nous évoquerons dans ce chapitre sont, soit du ressort du générateur de tables et de fonctions C, soit du ressort du noyau des analyseurs syntaxiques, et sont les suivants :

- codage des terminaux m-bandes et des index m-bandes ;
- phénomènes d'explosion combinatoire dans les états d'analyse et les ensemble de non-terminaux attendus ;
- codage et création statique ou dynamique des tables d'analyse ;
- ordonnancement des attributs physiques des symboles ;
- choix d'un algorithme de gestion de la file de priorité Q ;
- optimisation de l'allocation mémoire des états attribués et problèmes associés ;

4.1 Codages pour l'algorithme GCP m-bandes

4.1.1 Codage des états d'analyse

Il est commode lors de la présentation formelle d'un algorithme d'analyse syntaxique d'utiliser des notations faciles à appréhender, aussi bien par le lecteur de l'algorithme que par son auteur. Il en est ainsi des états d'analyse $[\Delta \rightarrowtail \alpha]$ ou $[\Delta \rightarrowtail \alpha, \vec{\imath}]$, des terminaux m-bandes ou des fonctions Expect, $\mathsf{Initial}$, Goto, Reduce. Lors de la programmation de ce même algorithme, il est en revanche plus pratique et plus efficace de remplacer ces notations et fonctions par des entiers et des tables d'analyse indexées par ces entiers. La stratégie de numérotation (dite également de codage) change d'un algorithme à l'autre, mais son principe est classique : un index est attribué à tout élément d'un ensemble fini utilisé par l'algorithme. Ce numéro sert à son tour d'index dans des tables d'analyse finies.

Les ensembles finis utilisés par l'algorithme GCP m-bandes et dont les éléments peuvent être numérotés sont les suivants :

- l'ensemble des états d'analyse $[\Delta \rightarrowtail \alpha]$,
- l'ensemble des symboles de la grammaires,
- l'ensemble 2^{V_N} des ensembles de non-terminaux.

Munis d'un codage entier des éléments des ensembles précédents, nous pouvons tabuler les fonctions utilisées dans l'algorithme :

- la fonction $\mathsf{Expect}\big([\Delta \to \alpha]\big) = \big\{D \in V_N \mid \exists B, \beta, \gamma, \exists A \in \Delta, A \to \alpha B\beta \ \wedge\ B \to^* D\gamma\big\}$ devient une table de numéros de sous-ensembles de V_N, indexée par un numéro d'état d'analyse,

- la fonction $\mathtt{Initial}(\Delta, X) = [\Delta' \rightarrowtail X]$ devient une table de numéros d'états d'analyse, indexée par un numéro de sous-ensemble de V_N et un numéro de symbole,

- la fonction $\mathsf{Goto}\big([\Delta \rightarrowtail \alpha], X\big) = [\Delta' \rightarrowtail \alpha X]$ devient une table de numéros d'états d'analyse, indexée par un numéro d'état d'analyse et un numéro de symbole,

- la fonction $\mathsf{Reduce}\big([\Delta \rightarrowtail \alpha]\big) = \big\{A \in \Delta \mid A \to \alpha\big\}$ devient une table indexée par un numéro d'état d'analyse. Plutôt que de stocker un numéro de sous-ensemble de V_N dans chaque entrée de la table, il en préférable de stocker la liste des fonctions d'évaluations d'attribut $f_{A\to\alpha}$ associées aux règles $A \to \alpha$;

- la fonction “réunion de deux sous-ensembles”, utilisée pour calculer itérativement $\Delta_{\vec{\jmath}} = \mathsf{Expect}(T_{\vec{\jmath}}) = \bigcup_{[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda] \in T_{\vec{\jmath}}} \mathsf{Expect}\big([\Delta \rightarrowtail \alpha]\big)$, devient une table de numéros de sous-ensembles de V_N, indexée par deux numéros de sous-ensembles de V_N. Nous appellerons cette table la *table des unions* et nous la noterons $T^{\cup}$.

Nous reparlerons dans la section suivante des problèmes pratiques posés par tous ces codages.

4.1.2 Codage des terminaux m-bandes

Deux autres codages se sont révélés particulièrement utiles pour écrire un algorithme GCP m-bandes aussi efficace lorsque $m = 1$ que l'algorithme GCP écrit spécifiquement pour une seule bande (Lefebvre, 1995b). Le premier codage est celui des terminaux m-bandes et le second est celui des index m-bandes.

Rappelons que les symboles de la grammaire appartiennent soit à l'ensemble V_N des non-terminaux, soit à l'ensemble $\vec{V}_T$ des terminaux. L'ensemble $\vec{V}_T$ doit être un alphabet m-bandes c'est-à-dire un produit de m alphabets Σ_k auxquels la chaîne vide a été ajoutée :

$$\vec{V}_T = \bigotimes_{1 \leqslant k \leqslant m} \big(\Sigma_k \cup \{\epsilon\}\big).$$

Le codage des états et fonctions d'analyse évoqué dans la section précédente nécessite une numérotation des éléments de $\vec{V}_T$, mais différentes considérations techniques (Lefebvre, 1996b) imposent à l'algorithme GCP de pouvoir calculer le code d'un élément de $\vec{V}_T$ à partir du code des éléments de $\Sigma_k \cup \{\epsilon\}$ et, réciproquement, de décomposer le code d'un élément de $\vec{V}_T$ en m codes d'éléments de $\Sigma_k \cup \{\epsilon\}$. La stratégie finalement adoptée, bien qu'élémentaire, s'est avérée parfaitement adaptée. Les éléments de $\Sigma_k \cup \{\epsilon\}$ sont numérotés, la chaîne vide ϵ ayant toujours le numéro 0. Le code d'un élément $a = \{\sigma_1, \ldots, \sigma_m\}$ de $\vec{V}_T$ est ensuite obtenu par simple produit :

$$\mathrm{code}(a) = \sum_{j=1}^{m} \mathrm{code}(\sigma_j) \prod_{k=1}^{j-1} \big(|\Sigma_k| + 1\big)$$

où $\mathrm{code}(a)$ (resp. $\mathrm{code}(\sigma_j)$) est le code de a (resp. de σ_j), $|\Sigma_k|$ est le nombre d'éléments de Σ_k et, par convention, $\prod_{k=1}^{0}\big(|\Sigma_k| + 1\big) = 1$.

Grâce à ce codage, le numéro d'un élément de $\vec{V}_T$ est égal au numéro d'un élément de Σ_1 lorsque $m = 1$. Ce codage des éléments de $\vec{V}_T$ est de plus compatible avec l'ordre des symboles imposé par la gestion de la file de priorité Q (voir la section 4.4).

4.1.3 Codage des index m-bandes

Un index m-bandes $\vec{\imath}$ est un vecteur de $\mathbb{Z}^m$. Un tel index est typiquement utilisé dans la notation $T_{\vec{\jmath}}$ des listes d'analyse m-bandes, ou encore la notation $[\Delta \rightarrowtail \alpha, \vec{\imath}]$ des états d'analyse m-bandes. Grâce au codage des états d'analyse $[\Delta \rightarrowtail \alpha]$, et comme $\vec{\imath} \geqslant 0$, il aurait été possible de considérer tout état $[\Delta \rightarrowtail \alpha, \vec{\imath}]$ comme un vecteur de $\mathbb{N}^{m+1}$, et d'indexer $T_{\vec{\jmath}}$ par un vecteur de $\mathbb{N}^m$. Bien que simples à concevoir, ces deux solutions présentent plusieurs inconvénients majeurs :

- la place prise en mémoire par les états d'analyse m-bandes devient très importante car le vecteur de $\mathbb{N}^{m+1}$ est représenté en pratique par $m+1$ entiers,
- l'indexation de $T_{\vec{\jmath}}$ n'est pas des plus simples,
- l'accès à tout élément dépendant d'un index m-bandes impose l'écriture d'une boucle dépendant de m. L'exécution d'une telle boucle représente une perte de temps lorsque $m = 1$, même si les compilateurs modernes peuvent détecter cette situation dans certains cas et simplifier la boucle.

Il est donc rapidement apparu nécessaire de coder les index m-bandes sous la forme d'un unique entier. Si le mot m-bandes de départ est $a_{\vec{1}} \ldots a_{\vec{n}}$, alors tous les index m-bandes $\vec{\imath}$ utilisés en cours d'analyse syntaxique vérifient $\vec{0} \leqslant \vec{\imath} \leqslant \vec{n}$. Étant données ces bornes, un premier codage vient immédiatement à l'esprit : celui obtenu grâce à un produit ressemblant à celui utilisé pour le codage des terminaux m-bandes.

$$\mathrm{code}(\vec{\imath}) = \sum_{j=1}^{m} \vec{\imath}^{\,(j)} \prod_{k=1}^{j-1} \big(\vec{n}^{\,(k)} + 1\big) \tag{4.1}$$

Nous avons utilisé la notation $\mathrm{code}(\vec{\imath})$ dans l'équation précédente pour désigner le code de $\vec{\imath}$.

Ce codage simple pose un petit problème : la relation d'ordre $\leqslant$ sur les index m-bandes n'est pas transférée au niveau du code lorsque $m > 1$. Pourtant, l'algorithme

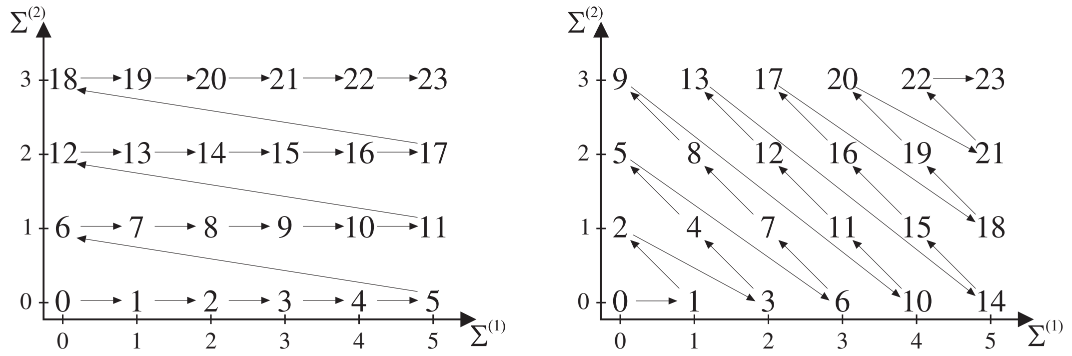


Figure 4.3: À gauche, codage des index 2-bandes par produit simple ($\vec{n}^{(1)} = 5$ et $\vec{n}^{(2)} = 3$). À droite, codage des index 2-bandes en hyperplans de somme constante.

GCP m-bandes a besoin de cette relation d'ordre pour calculer $T_{\vec{\jmath}}$ en ordre croissant d'index $\vec{\jmath}$ et pour extraire les triplets $(X, \vec{\imath}, x)$ de Q en ordre décroissant d'index $\vec{\imath}$. Ne pas transférer la relation d'ordre au niveau du code des index nécessiterait alors un décodage à chaque fois qu'une comparaison est nécessaire.

Nous avons donc remplacé le codage précédent par un codage dit *par hyperplans de somme constante*. Le principe est issu de la remarque suivante :

$$\text{si } \vec{\imath} \leqslant \vec{\jmath} \text{ alors } \sum_{k=1}^{m} \vec{\imath}^{(k)} \leqslant \sum_{k=1}^{m} \vec{\jmath}^{(k)}$$

Pour conserver la relation d'ordre au niveau du code, il suffit alors de s'assurer que si $\sum_{k=1}^{m} \vec{\imath}^{(k)} \leqslant \sum_{k=1}^{m} \vec{\jmath}^{(k)}$, alors $\text{code}(\vec{\imath}) \leqslant \text{code}(\vec{\jmath})$. Nous attribuons donc d'abord un numéro aux index $\vec{\imath}$ tels que la somme $\sum_{k=1}^{m} \vec{\imath}^{(k)}$ vaut 0 (donc $\vec{\imath} = \vec{0}$), puis à ceux tels que la somme vaut 1, etc., jusqu'à l'index $\vec{\imath} = \vec{n}$, dont la somme vaut $\sum_{k=1}^{m} \vec{n}^{(k)}$. La figure 4.3 illustre ce principe. Grâce à ce code, on a bien $\vec{\imath} \leqslant \vec{\jmath} \Rightarrow \text{code}(\vec{\imath}) \leqslant \text{code}(\vec{\jmath})$ mais la réciproque est fausse. Notons cependant que si $\text{code}(\vec{\imath}) \leqslant \text{code}(\vec{\jmath})$ et si $\vec{\imath}$ est comparable à $\vec{\jmath}$, alors $\vec{\imath} \leqslant \vec{\jmath}$. Si $\vec{\imath}$ n'est pas comparable à $\vec{\jmath}$, il n'y a aucun problème à traiter l'élément associé à $\vec{\imath}$ indépendamment de celui associé à $\vec{\jmath}$ car ils n'ont aucune influence l'un sur l'autre, que ce soit pour l'ordre de calcul des listes d'analyse ou pour l'ordre d'extraction des triplets de la file de priorité Q.

Comme il faut savoir calculer rapidement le code d'un index étant données m positions sur les m bandes, une table de transformation de codes est utilisée pour obtenir le code par hyperplans (trop long à calculer systématiquement, donc précalculé) à partir de celui de l'équation (4.1) peu coûteux à calculer systématiquement. Symétriquement, une table de décodage permet de passer du code par hyperplans à un vecteur de m positions sur les m-bandes.

Grâce au code que nous venons de décrire, les listes d'analyse $T_{\vec{\jmath}}$ peuvent en fait être implémentées sous la forme d'une table indexée par $\text{code}(\vec{\jmath})$ et les états d'analyse m-bandes $[\Delta \rightarrowtail \alpha, \vec{\imath}]$ sous la forme d'un couple d'entiers. Ces deux représentations étant les mêmes que celles utilisées par l'algorithme GCP spécialisé pour une seule bande (Lefebvre, 1996b), il a été possible de réutiliser de nombreuses portions de cet analyseur pour écrire l'analyseur m-bandes.

4.2 Création des tables d'analyse

Dans cette section, nous allons examiner plus attentivement les conséquences des codages des états d'analyse et des ensembles de non-terminaux décrits en section 4.1.1. En particulier nous allons montrer que des phénomènes d'explosion combinatoire des codages peuvent apparaître avec l'algorithme GCP.

Les différentes tables d'analyse utilisent la numérotation des états d'analyse et des ensembles de non-terminaux pour faciliter l'analyse syntaxique. Ces tables sont théoriquement calculées pendant la phase de génération de l'analyseur syntaxique de façon à décharger l'algorithme d'analyse syntaxique des calculs dont le résultat est indépendant du mot m-bandes analysé, surtout lorsque ces calculs sont coûteux.

Rappelons que l'ensemble Δ d'un état d'analyse $[\Delta \rightarrowtail \alpha]$ est réellement un ensemble de non-terminaux, au même titre que l'ensemble retournée par $\mathsf{Expect}\big([\Delta \rightarrowtail \alpha]\big)$. Si 50 non-terminaux différents partagent le même préfixe α, le nombre d'états d'analyse potentiellement utilisables par l'algorithme GCP pour ce préfixe α est de $2^{50} - 1$. En pratique, seul un très petit sous-ensemble de cet ensemble énorme est généralement utilisé car les non-terminaux tendent à être attendus par groupes. Quel est ce petit sous-ensemble ? De même, si la grammaire contient 100 non-terminaux, alors le nombre d'ensembles de non-terminaux possibles est de 2^{100}. Quels ensembles de non-terminaux parmi ces 2^{100} ensembles possibles peuvent réellement apparaître au cours d'une analyse syntaxique ? Nous allons illustrer ces deux questions par deux exemples.

Exemple 4.1 *Les ensembles de non-terminaux sont créés par la fonction* Expect *et ils sont réunis pour créer un ensemble* $\Delta_{\vec{j}} = \mathsf{Expect}(T_{\vec{j}})$ *qui sera à son tour utilisé par la fonction* $\mathsf{Initial}$. *Soit la grammaire suivante :*

$$
\begin{aligned}
S &\to a_0 A_0 \\
A_0 &\to a_1 A_1 \\
&\vdots \\
A_i &\to a_{i+1} A_{i+1} \\
&\vdots \\
A_{n-1} &\to a_n
\end{aligned}
$$

Il y a $2n+1$ *états d'analyse possibles pour cette grammaire et tous ces états apparaîtront lors de l'analyse de la chaîne* $a_0 a_1 \ldots a_n$ *(c'est la seule chaîne appartenant au langage généré par la grammaire). Les* n *sous-ensembles* $\mathsf{Expect}\big([\{A_{i-1}\} \rightarrowtail a_i]\big) = \{A_i\}$ *apparaîtront donc également lors de l'analyse. Ces* n *sous-ensembles devront donc avoir un numéro permettant d'indexer la table des unions* $T^{\cup}$. *De plus,* $T^{\cup}$ *doit permettre* *de calculer toute union nécessaire en cours d'analyse syntaxique. La méthode la plus simple et la plus sûre pour satisfaire cette contrainte est la clôture de la table par union : quels que soient* k *et* l *deux numéros d'ensembles, l'entrée* $T^{\cup}_{k,l}$ *de* $T^{\cup}$ *existe et contient un numéro de sous-ensemble à nouveau utilisable comme index dans* $T^{\cup}$ *(même si* k *et* l *ne peuvent apparaître simultanément lors du calcul d'une quelconque entrée* T_j*). L'application de ce principe à la grammaire précédente conduit à la création d'une table contenant* $(2^n)^2 = 2^{2n}$ *éléments. Or une étude rapide de la grammaire montre qu'à chaque étape de l'algorithme,* $\Delta_j = \{A_{j-1}\}$ *car* T_j *ne contient que l'état* $[\Delta \rightarrowtail a_{j-1}]$*. Aucune union ne sera donc calculée et les* 2^{2n} *éléments de* $T^{\cup}$ *sont en fait inutiles. Cependant, l'exemple suivant montre que l'étude de la grammaire ne permet pas toujours d'éviter le calcul d'un nombre exponentiel de sous-ensembles.*

Exemple 4.2 *Soit la grammaire suivante :*

$$\begin{aligned}
S &\to a_1A_1 \mid b_1B_1 \\
A_1 &\to a_2A_2 \mid b_2B_2 \mid CA_1 \\
B_1 &\to a_2A_2 \mid b_2B_2 \mid CB_1 \\
&\vdots \\
A_{n-1} &\to a_nA_n \mid b_nB_n \mid CA_{n-1} \\
B_{n-1} &\to a_nA_n \mid b_nB_n \mid CB_{n-1} \\
A_n &\to C \\
B_n &\to C \\
C &\to a_1 \mid \cdots \mid a_n \mid b_1 \mid \cdots \mid b_n
\end{aligned}$$

Au départ, $\Delta_0 = \{S\}$*. On montre aisément que si la chaîne à analyser par l'algorithme GCP est de la forme* $x_1 \ldots x_n$ *(où* $x_i = a_i$ *ou* b_i*), alors* Δ_j $(1 \leqslant j \leqslant n)$ *est de la forme* $\{C, X_1, \ldots, X_j\}$ *(où* $X_i = A_i$ *si* $x_i = a_i$ *et* $X_i = B_i$ *si* $x_i = b_i$*). Donc le nombre d'ensembles de non-terminaux à numéroter est au moins égal à* 2^{n+1} *et croît exponentiellement avec la taille de la grammaire. Comme* $\Delta_j = \{C, X_1, \ldots, X_j\}$*, la liste d'analyse* T_{j+1} *contient un état d'analyse* $[\{X_1, \ldots, X_j\} \rightarrowtail C]$*, donc le nombre d'états d'analyse possibles croît lui aussi exponentiellement avec la taille de la grammaire.*

Aucune étude de la grammaire précédente ne permettra d'éviter à la taille des tables de croître exponentiellement avec n *si l'on veut tenir compte à l'avance (i.e. statiquement) de toutes les analyses possibles.*

Bien sûr, la grammaire de l'exemple précédent peut sembler dégénérée et il est raisonnable de penser que de telles grammaires se rencontrent rarement en pratique. Effectivement, aucune des MTSAG présentées au chapitre 3 ne présente ce type de comportement. Ainsi, la grammaire nécessitant les plus grosses tables d'analyse est celle du modèle de Sankoff généralisé (p. 108) puisque 7025 états d'analyse et 1092 ensembles de non-terminaux sont répertoriés dans ces tables alors que la grammaire contient 4356 règles et 25 non-terminaux. Cependant, il serait regrettable de ne pouvoir utiliser l'algorithme GCP m-bandes avec une grammaire donnée simplement parce que la génération statique des tables d'analyse est irréaliste. Nous exposerons donc un algorithme de calcul dynamique des tables d'analyse immédiatement après l'algorithme de calcul statique.

4.2.1 Algorithme de calcul statique des tables d'analyse

Même si la tabulation statique des tables d'analyse semble parfois se heurter à un problème fondamental d'explosion combinatoire, il ne faut pas pour autant la rejeter définitivement car elle reste applicable dans de nombreux cas. L'algorithme statique que nous allons décrire n'est pas optimal car les tables créées contiennent parfois des valeurs inutiles qui ne sont utilisées au cours d'aucune analyse, mais il est cependant préférable à une numérotation systématique et aveugle qui garantit l'explosion combinatoire. De plus, les valeurs inutiles n'ont aucune incidence sur les performances de l'analyseur généré. Par contre, rien n'assure la génération d'une minorité de valeurs inutiles puisqu'il suffit d'appliquer l'algorithme à la grammaire de l'exemple 4.1 pour se rendre compte qu'il génère en fait une énorme majorité d'ensembles de non-terminaux inutiles. En pratique, cet algorithme fonctionne de manière satisfaisante.

Algorithme 4.1 (Construction des tables d'analyse)
L'algorithme suivant

- *prend en entrée une grammaire* $\vec{G}$,
- *retourne un sur-ensemble* $\mathcal{E}$ *des* $\Delta_{\vec{\jmath}}$ *réellement utiles à l'analyse syntaxique ainsi qu'un sur-ensemble* $\mathcal{S}$ *des états* $[\Delta \rightarrowtail \alpha]$ *réellement utiles à l'analyse syntaxique.*

Soit $\mathcal{D}$ *un ensemble de même type que* $\mathcal{E}$. *Au départ,* $\mathcal{E} = \{\emptyset\}$, $\mathcal{D} = \{\Delta_{\vec{0}}\}$, *où* $\Delta_{\vec{0}} = \{A \in V_N \mid \exists\beta, S \to^* A\beta\}$, *et* $\mathcal{S} = \emptyset$.

1. *Extraire un ensemble* Δ_1 *de* $\mathcal{D}$ *et l'ajouter à* $\mathcal{E}$ *;*
2. *Construction des états initiaux déduits de* Δ_1 *:*
   $\mathcal{S} := \mathcal{S} \cup \Big\{[\Delta' \rightarrowtail X] \;\Big|\; \exists X \in V, [\Delta' \rightarrowtail X] = \mathsf{Initial}(\Delta_1, X)\Big\}$
3. *Prolongation des états déjà ajoutés à* $\mathcal{S}$ *:*
   $\mathcal{S} := \mathcal{S} \cup \Big\{[\Delta' \rightarrowtail \alpha X] \;\Big|\; \exists X \in V, \exists[\Delta \rightarrowtail \alpha] \in \mathcal{S}, [\Delta' \rightarrowtail \alpha X] = \mathsf{Goto}\big([\Delta \rightarrowtail \alpha], X\big)\Big\}$
4. *Répéter l'étape 3 tant que* $\mathcal{S}$ *n'est pas stationnaire ;*
5. *Calcul des nouveaux ensembles de non-terminaux attendus :*
   $\mathcal{D} := \mathcal{D} \cup \Big\{\mathsf{Expect}\big([\Delta \rightarrowtail \alpha]\big) \;\Big|\; [\Delta \rightarrowtail \alpha] \in \mathcal{S}, \mathsf{Expect}\big([\Delta \rightarrowtail \alpha]\big) \notin \mathcal{E}\Big\}$;
6. *Clôture de* $\mathcal{D} \cup \mathcal{E}$ *:*
   $\mathcal{D} := \mathcal{D} \cup \Big\{\Delta \cup \Delta' \;\Big|\; \Delta \in \mathcal{D}, \Delta' \in \mathcal{D} \cup \mathcal{E}, \Delta \cup \Delta' \notin \mathcal{E}\Big\}$;
7. *Répéter l'étape 6 tant que* $\mathcal{D}$ *n'est pas stationnaire ;*
8. *Répéter les étapes 1 à 8 tant que* $\mathcal{D}$ *n'est pas vide.*

L'étape 6 assure que l'ensemble $\mathcal{E}$ retourné par l'algorithme est bien clos par union de ces éléments. Cet algorithme se termine toujours car $\mathcal{E}$ est borné par l'ensemble des sous-ensembles de V_N et un ensemble n'est ajouté à $\mathcal{D}$ que s'il n'appartient pas déjà à $\mathcal{E}$. Il est trivial de vérifier par récurrence sur les étapes de l'algorithme GCP que les ensembles $\mathcal{E}$ et $\mathcal{S}$ produits par l'algorithme 4.1 contiennent bien tous les ensembles de non-terminaux et tous les états nécessaires pour n'importe quelle analyse syntaxique.

Dans la plupart des cas, l'algorithme précédent retourne des ensembles $\mathcal{E}$ et $\mathcal{S}$ significativement plus petits que l'ensemble des sous-ensembles de V_N et l'ensemble des états d'analyse possibles. La figure 4.4 présente un exemple d'ensembles $\mathcal{E}$ et $\mathcal{S}$ construits à partir d'une grammaire. Le graphe joint symbolise les relations entre états d'analyse. La numérotation et les tables d'analyses déduites des résultats de la figure 4.4 sont représentées en figure 4.5.

4.2.2 Algorithme de calcul dynamique des tables d'analyse

Les étapes de l'algorithme 4.1 responsables de l'explosion combinatoire sont les étapes 6 et 7 qui réalisent la clôture de $\mathcal{E}$. Or, comme nous l'avons vu dans l'exemple 4.1, certains ensembles de non-terminaux attendus ne seront jamais combinés avec d'autres ensembles de non-terminaux attendus pour la simple raison que les états associés ne peuvent pas apparaître en même temps dans $T_{\vec{\jmath}}$. Cependant, dans certains cas (exemple 4.2), l'explosion combinatoire est inévitable et aucune amélioration de l'algorithme 4.1 ne pourra l'empêcher.

Lorsque le calcul statique des tables est irréaliste[2], il faut se contenter d'un calcul dynamique. Dans son principe, le calcul dynamique est extrêmement simple. Il suffit en effet en cours d'analyse syntaxique, une fois $\Delta_{\vec{\jmath}}$ obtenu, de mémoriser ce $\Delta_{\vec{\jmath}}$ dans $\mathcal{E}$ et d'appliquer les étapes 2 à 5 de l'algorithme 4.1. Les étapes 6 et 7 ne sont pas appliquées et, à chaque union calculée par l'algorithme GCP pour obtenir un nouveau $\Delta_{\vec{\jmath}'}$ (étape 7 dans la figure 4.1 p. 180), l'ensemble $\mathcal{E}$ et la table $T^{\cup}$ sont mis à jour.

En pratique, le calcul dynamique des tables augmente la taille du code de l'analyseur syntaxique puisqu'il doit contenir une partie du code de MTSAG2C et une description compactée de la grammaire de départ stockée sous forme de trie (cf. figure 4.2 p. 181). Le calcul dynamique ralentit aussi l'analyse même si, dans son implémentation actuelle, le ralentissement est quasiment imperceptible. En outre, les tables générées pendant une analyse syntaxique sont réutilisables lors d'une analyse syntaxique ultérieure. Le coût de calcul des tables est donc une constante amortissable dépendant uniquement de la grammaire. Nous ne l'intégrons donc jamais au coût de l'analyse syntaxique proprement dite.

Le calcul dynamique est choisi au moment de la compilation d'une MTSAG en fournissant l'option `-dynamic` à MTSAG2C ou en ajoutant une déclaration `%mode "-dynamic"` à la MTSAG (cf. annexe A p. 240).

[2] Il est difficile de choisir un critère a priori d'abandon de la génération statique des états. La génération dynamique pourra toutefois être préférée lorsque le nombre d'états générés par l'algorithme 4.1 dépasse une dizaine de fois le nombre de productions de la grammaire.

$$G = \left\{ \begin{array}{l} S \rightarrow A \mid B \mid aSu \mid gSc \\ A \rightarrow aCu \mid gCc \\ B \rightarrow aCSCu \mid gCSCc \\ C \rightarrow a \mid c \mid g \mid u \mid Ca \mid Cc \mid Cg \mid Cu \end{array} \right. \qquad \mathcal{E} = \left\{ \begin{array}{l} \{S, A, B, C\} \\ \{S, A, B\} \\ \{C\} \\ \emptyset \end{array} \right\}$$

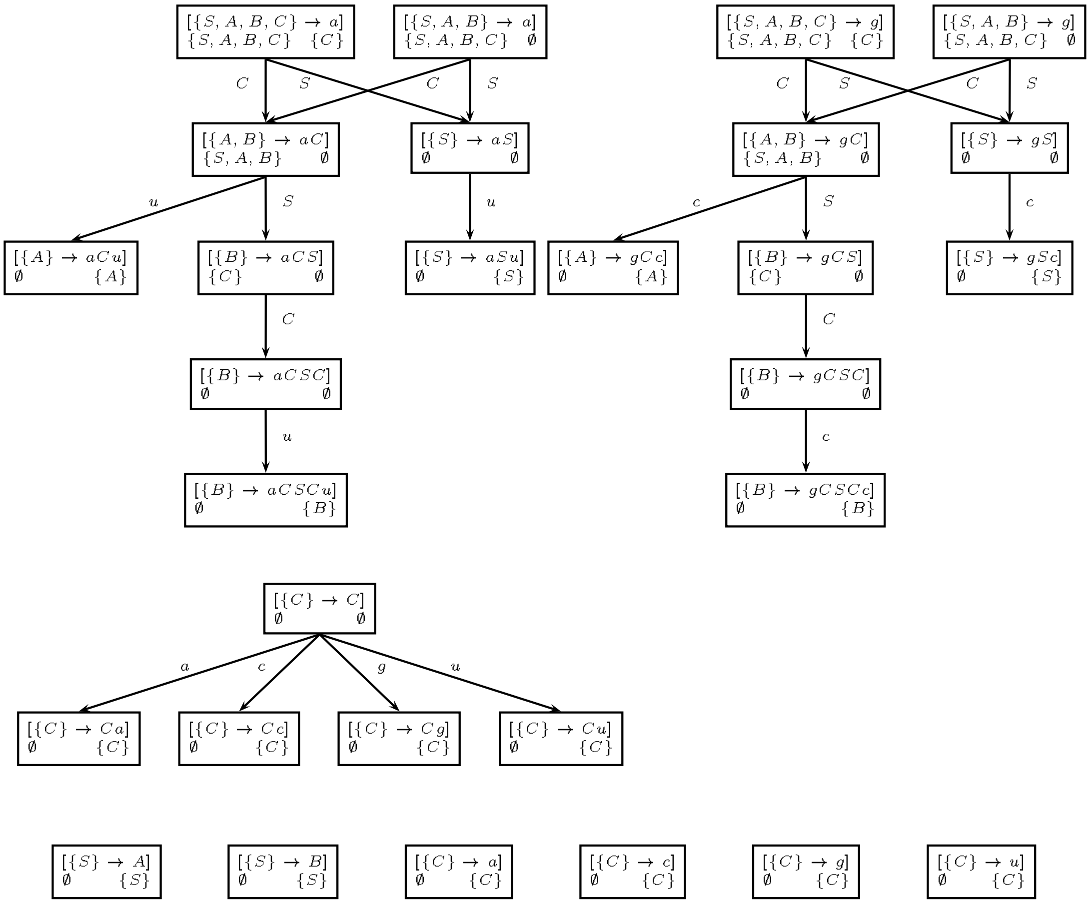
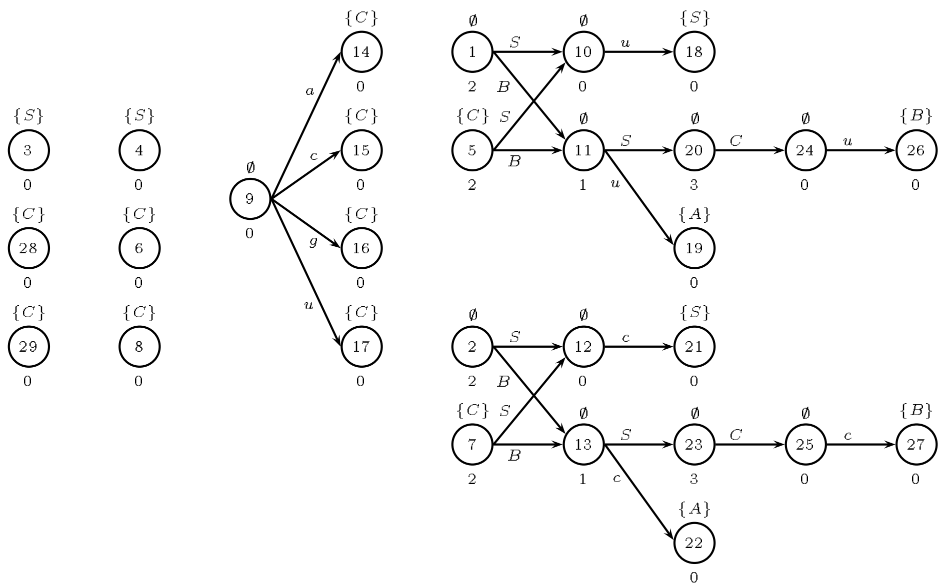


Figure 4.4: La grammaire de cette figure est une version extrêmement simplifiée d'une grammaire décrivant des structures secondaires d'ARN. Chaque rectangle du graphe dessiné contient un état $[\Delta \rightarrowtail \alpha]$ de $\mathcal{S}$ (partie haute du rectangle), $\mathsf{Expect}([\Delta \rightarrowtail \alpha])$ (en bas à gauche) et $\mathsf{Reduce}([\Delta \rightarrowtail \alpha])$ (en bas à droite). Deux états sont reliés par un arc lorsque la fonction Goto permet de passer de l'un à l'autre à l'aide du symbole étiquetant l'arc. Le nombre d'états de $\mathcal{S}$ (29) est inférieur au nombre total d'états possibles (61). De même, $\mathcal{S}$ contient seulement 4 ensembles au lieu des 16 ensembles possibles.



Initial	a	c	g	u	S	A	B	C
0 $\emptyset$	0	0	0	0	0	0	0	0
1 $\{S,A,B\}$	1	0	2	0	0	3	4	0
2 $\{S,A,B,C\}$	5	6	7	9	0	3	4	9
3 $\{C\}$	28	6	29	8	0	0	0	9

$T^\cup$	0	1	2	3
0	0	1	2	3
1	1	1	2	2
2	2	2	2	2
3	3	2	2	3

Goto	a	c	g	u	S	A	B	C	Goto	a	c	g	u	S	A	B	C
0 ϵ	0	0	0	0	0	0	0	0	15 $[\{C\} \to C\ c]$	0	0	0	0	0	0	0	0
1 $[\{S,A,B\} \to a]$	0	0	0	0	10	0	11	0	16 $[\{C\} \to C\ g]$	0	0	0	0	0	0	0	0
2 $[\{S,A,B\} \to g]$	0	0	0	0	12	0	13	0	17 $[\{C\} \to C\ u]$	0	0	0	0	0	0	0	0
3 $[\{S\} \to A]$	0	0	0	0	0	0	0	0	18 $[\{S\} \to a\ S\ u]$	0	0	0	0	0	0	0	0
4 $[\{S\} \to B]$	0	0	0	0	0	0	0	0	19 $[\{A\} \to a\ C\ u]$	0	0	0	0	0	0	0	0
5 $[\{S,A,B,C\} \to a]$	0	0	0	0	10	0	11	0	20 $[\{B\} \to a\ C\ S]$	0	0	0	0	0	0	0	24
6 $[\{C\} \to c]$	0	0	0	0	0	0	0	0	21 $[\{S\} \to g\ S\ c]$	0	0	0	0	0	0	0	0
7 $[\{S,A,B,C\} \to g]$	0	0	0	0	12	0	13	0	22 $[\{A\} \to g\ C\ c]$	0	0	0	0	0	0	0	0
8 $[\{C\} \to u]$	0	0	0	0	0	0	0	0	23 $[\{B\} \to g\ C\ S]$	0	0	0	0	0	0	0	25
9 $[\{C\} \to C]$	14	15	16	17	0	0	0	0	24 $[\{B\} \to a\ C\ S\ C]$	0	0	0	26	0	0	0	0
10 $[\{S\} \to a\ S]$	0	0	0	18	0	0	0	0	25 $[\{B\} \to g\ C\ S\ C]$	0	0	27	0	0	0	0	0
11 $[\{A,B\} \to a\ C]$	0	0	0	19	20	0	0	0	26 $[\{B\} \to a\ C\ S\ C\ u]$	0	0	0	0	0	0	0	0
12 $[\{S\} \to g\ S]$	0	0	21	0	0	0	0	0	27 $[\{B\} \to g\ C\ S\ C\ c]$	0	0	0	0	0	0	0	0
13 $[\{A,B\} \to g\ C]$	0	0	22	0	23	0	0	0	28 $[\{C\} \to a]$	0	0	0	0	0	0	0	0
14 $[\{C\} \to C\ a]$	0	0	0	0	0	0	0	0	29 $[\{C\} \to g]$	0	0	0	0	0	0	0	0

Figure 4.5: Sur cette figure, les numéros des nœuds du graphe représenté sont ceux attribués aux états de l'ensemble $\mathcal{S}$ de la figure 4.4. Les tables des fonctions Initial, Goto et $T^\cup$ correspondant à ces états sont données. La valeur de la fonction Reduce est rappelée au dessus de chaque numéro d'état, tandis que celle de la fonction Expect est rappelée au dessous.

4.3 Ordonnancement des attributs physiques des symboles

Les noms des attributs physiques utilisables dans une MTSAG sont indiqués dans l'en-tête de la MTSAG à l'aide de la déclaration suivante (cf. p. 243):

`%attribute` "*type du langage C*" *liste de symboles*

Il faut ensuite déclarer, implicitement ou explicitement, quels sont les attributs physiques utilisés par les symboles de la grammaire, afin que MTSAG2C soit à même de déterminer quels attributs physiques sont utilisés par chaque symbole.

L'implémentation la plus simple des attributs physiques consisterait à créer une unique structure réunissant tous les attributs physiques possibles. Cette structure serait alors appelée *attribut*. Cette solution est parfaite lorsque tous les symboles utilisent la totalité des attributs physiques déclarés, mais elle est dispendieuse lorsque chaque symbole utilise ses propres attributs physiques.

Une autre solution simple consiste à déclarer une structure *attribut* par symbole, chaque structure regroupant "en vrac" les attributs physiques utilisés par son symbole associé, et uniquement ces attributs physiques. Si la gestion de l'allocation des attributs dans l'analyseur syntaxique ne s'en trouve pas simplifiée (les structures ont potentiellement des tailles différentes), c'est en revanche la solution la plus économique. Elle est malheureusement inutilisable en l'état à cause du terme "en vrac" et de la possibilité offerte à l'utilisateur de déclarer des fonctions génériques d'évaluation d'attributs (cf. section A.4.2.12 p. 247). Ces fonctions doivent pouvoir être appelées avec des symboles différents, pour un unique code d'accès aux attributs physiques. Si deux règles appellent une fonction avec des non-terminaux possédant un même attribut physique `E` à des emplacements différents de leurs structures d'attributs respectives, un conflit d'accès surgira.

Il faut donc réussir à déterminer un ordre des attributs physiques tel qu'un attribut physique donné apparaît uniquement chez les symboles qui l'utilisent et toujours en même position dans les structures. Cela signifie implicitement que les attributs physiques précédant un attribut physique donné sont toujours les mêmes d'une structure à l'autre car MTSAG2C ne connaît pas la taille des types associés aux attributs physiques. Il n'est pas toujours possible de déterminer un tel ordre. Ainsi, si A utilise les attributs physiques p et q, B utilise p et C utilise q, aucune solution n'existe. Dans un tel cas, il faut ajouter l'attribut physique q à la liste des attributs physiques de B, ou p à la liste de C, ou changer le nom des attributs physiques d'un des non-terminaux.

La figure 4.6 illustre la méthode utilisée. La liste des symboles utilisant un attribut physique est d'abord déterminée pour chaque attribut physique. Puis les listes sont comparées deux à deux. Une première liste incluant une seconde indique que l'attribut physique de la première doit apparaître avant l'attribut physique de la seconde. Deux listes dont l'intersection est non vide sans qu'aucune ne soit incluse dans l'autre signalent une erreur[3]. La relation d'inclusion sur les listes permet de construire un graphe (une forêt, en fait) et le parcours des composantes connexes de ce graphe en partant des sommets révèle l'ordre des attributs physiques dans la structure de chaque symbole.

[3] La grammaire est alors rejetée. Cela ne constitue pas une restriction sur la classe des grammaires acceptées puisqu'il suffit de modifier les noms des attributs physiques impliqués dans le rejet pour supprimer l'erreur. Cette modification a pour seule conséquence négative d'augmenter l'allocation mémoire en cours d'analyse syntaxique.

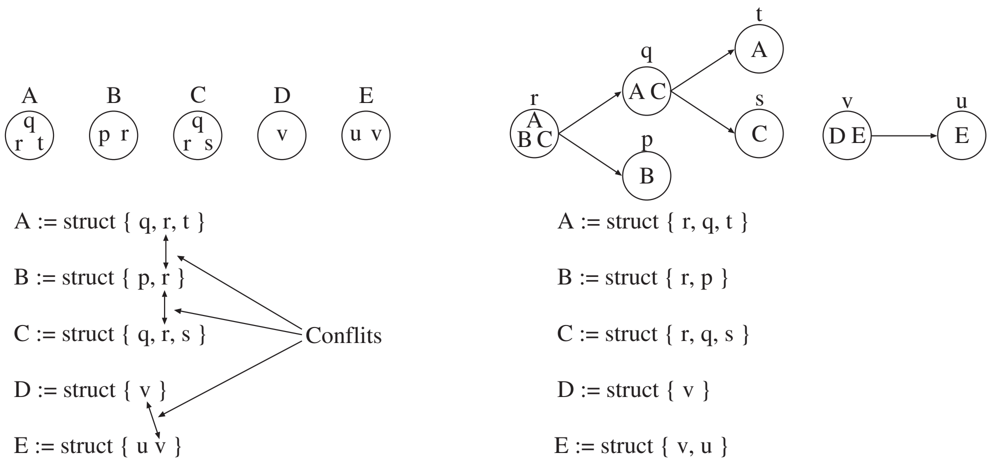


Figure 4.6: Sept attributs physiques p, q, r, s, t, u sont répartis entre cinq symboles A, B, C, D, E (cercles en haut à gauche). L'écriture directe des structures à partir des listes d'attributs génère des conflits dans les structures (à gauche). Le graphe des utilisations d'attributs (en haut à droite) permet d'obtenir l'ordre correct des attributs physiques (à droite).

4.4 La gestion de la file de priorité

La file de priorité m-bandes Q utilisée par l'algorithme GCP m-bandes est un ensemble de triplets $(X, \vec{\imath}, x)$ muni d'une relation d'ordre partiel sur les triplets :

$$(X', \vec{\imath}', x') < (X, \vec{\imath}, x) \Leftrightarrow \vec{\imath}' < \vec{\imath} \text{ ou } \vec{\imath}' = \vec{\imath} \text{ et } X' \to^{+} X$$

Il est apparu que les premières implémentations de l'algorithme GCP pouvaient passer jusqu'à 30 % de leur temps à insérer et extraire des triplets de Q, et que cette proportion dépendait essentiellement du degré d'ambiguïté de la grammaire (notion délicate à définir formellement dans le présent context) et de son nombre de non-terminaux. Avant de présenter une méthode améliorée de gestion de Q ayant permis de ramener cette proportion à 10 % dans le pire des cas, nous allons exposer les méthodes initialement utilisées.

À chaque extraction de Q, l'algorithme GCP n'extrait qu'un triplet maximal de Q, c'est-à-dire un triplet qui n'est inférieur à aucun autre triplet de Q. Il est aisé de transformer la relation d'ordre partiel sur les triplets en une relation d'ordre total en affectant un ordre arbitraire **fixe** à X' par rapport à X si $X' \not\to^{+} X$ et $X \not\to^{+} X'$. Munis de cet ordre arbitraire, nous pouvons implémenter Q de diverses manières :

1. utiliser la relation d'ordre total sur les triplets pour implémenter Q trivialement, c'est-à-dire soit sous la forme d'une liste ou d'une table non triée, soit sous la forme d'une liste ou d'une table triée,

2. utiliser la relation d'ordre total sur les triplets pour implémenter Q un peu moins trivialement, à l'aide d'une structure de tas par exemple,

3. intégrer le fait que $\vec{\imath}$ est une clé primaire de la relation $<$ et créer une première table indexée par $\vec{\imath}$ (le code de $\vec{\imath}$ en fait), chaque entrée de la table contenant un pointeur sur une file de priorité regroupant tous les triplets de même $\vec{\imath}$.

Chacune des méthodes précédentes représente un compromis entre la vitesse d'ajout d'un nouveau triplet et la vitesse d'extraction. La première méthode est bien évidemment sans intérêt puisque n'importe laquelle des deux méthodes suivantes la surpasse sans difficulté.

La deuxième méthode est d'autant plus performante que la profondeur moyenne de la file de priorité est faible, puisque le coût d'insertion d'un élément dans une file dépend directement de la taille de la file au moment de l'insertion. Les grammaires peu ou non-ambiguës, ou possédant une projection non-ambiguë avantagent cette méthode, alors que les grammaires très ambiguës la désavantagent. Dans tous les cas, la gestion de la structure de données sous-jacente introduit un surcoût non-négligeable. La structure de données offrant en pratique les meilleures performances en moyenne (compte-tenu des constantes cachées) pour cette deuxième méthode est celle d'arbre pagode (Francon *et al.*, 1978; Gonnet & Baeza-Yates, 1991), modifiée pour maintenir dans chaque nœud d'un arbre deux pointeurs sur les fils gauche et droit respectivement (figure 4.7).

La troisième méthode (figure 4.8) permet d'ajouter plus rapidement un nouveau triplet à la file de priorité que la deuxième méthode puisqu'elle décompose l'ensemble des triplets en sous-ensembles de taille bornée. Le coût d'insertion d'un triplet devient alors constant (si on ne tient pas compte du nombre de symboles de la grammaire) et le coût amorti d'extraction d'un triplet est lui aussi constant puisque l'extraction se fait en partant de $\vec{\imath} = \vec{\jmath}$ et en explorant toutes les entrées de la table jusqu'à $\vec{\imath} = \vec{0}$. Cette méthode est particulièrement intéressante lorsque la grammaire est très ambiguë (i.e. la quasi-totalité des entrées de la table indexée par $\vec{\imath}$ contiennent une file de priorité non vide lorsqu'elles sont explorées) et contient peu de non-terminaux (i.e. la constante d'insertion dans chaque file de priorité de la table est faible). Par contre, dès que la grammaire n'est plus suffisamment ambiguë, et en particulier dans le cas où une ou plusieurs de ses projections sont LR(1), l'algorithme d'extraction passe un temps significatif, une fois vidée la file d'une entrée $\vec{\imath}$, à parcourir la table dans le sens des $\vec{\imath}$ décroissant pour trouver une entrée dont la file est non vide. De même, dès que la grammaire contient beaucoup de non-terminaux, la constante d'insertion et d'extraction dans la file d'une entrée devient significative et il faut de nouveau trouver une méthode performante pour gérer la file de priorité de chaque entrée. Ce qui nous ramène aux algorithmes déjà évoqués (arbre pagode). Si la grammaire contient beaucoup de non-terminaux et est peu ambiguë (comme la grammaire pour les HMM à une bande de la figure 3.22 p. 139), alors la troisième méthode est invariablement moins performante que la deuxième.

À la lumière de la discussion précédente, il semble que la sélection d'une méthode universelle fonctionnant pour n'importe quelle grammaire soit assez délicate. Soit la grammaire est peu ambiguë, et la deuxième méthode est préférable, soit la grammaire est très ambiguë et comporte peu de non-terminaux, et la troisième méthode est préférable. Les premières implémentations de MTSAG2C imposaient à l'utilisateur de

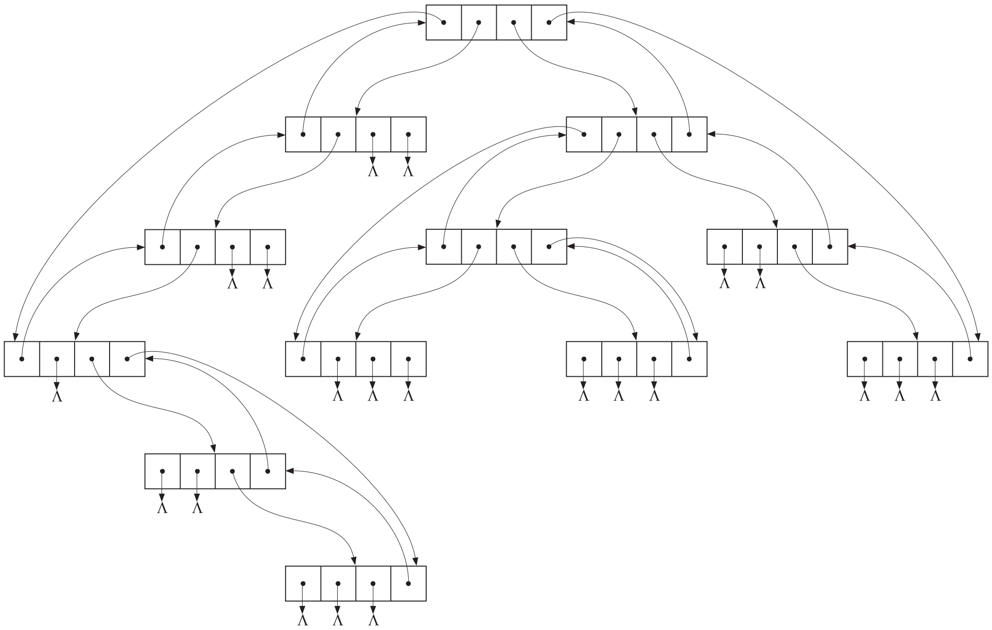


Figure 4.7: Structure d'arbre pagode modifiée pour les files de priorité de l'algorithme GCP. Chaque sommet de l'arbre contient quatre pointeurs. Le premier pointe vers le descendant le plus à gauche (pour la racine ou un descendant droit), ou le sommet parent (pour un descendant gauche). Le second pointe sur le descendant gauche. Le troisième pointe sur le descendant droit. Le quatrième pointe sur le descendant le plus à droite (pour la racine ou un descendant gauche), ou le sommet parent (pour un descendant droit).

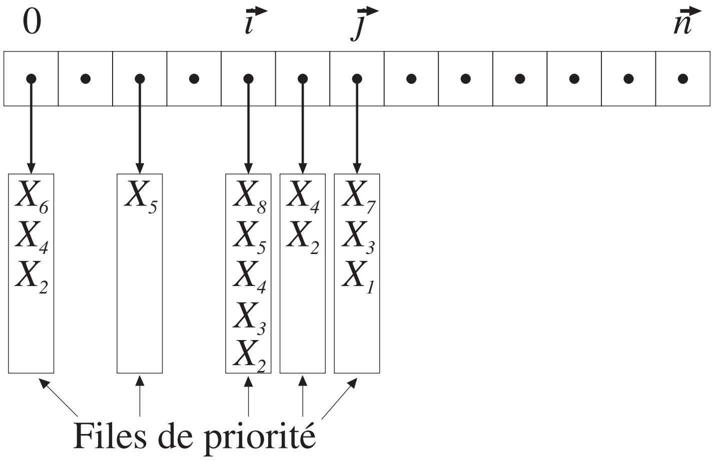


Figure 4.8: Décomposition de Q en une table de files de priorité indexée par le code $\vec{\imath}$ des triplets $(X, \vec{\imath}, x)$.

choisir lui-même sa méthode de gestion de la file de priorité. L'inconvénient principal de cette stratégie réside dans la difficulté d'un tel choix lorsque la grammaire est "moyennement grosse" et/ou "moyennement ambiguë". La solution standard consiste à exécuter un certain nombre de tests avec diverses méthodes et à retenir celle qui offre les meilleures performances. C'est une perte de temps non négligeable, surtout lorsque la grammaire peut évoluer et inverser les résultats des tests précédents. Il n'est bien sûr pas raisonnable de demander à MTSAG2C de choisir automatiquement la meilleure méthode car il risque fort de se retrouver face à un problème d'indécidabilité. Fort heureusement, il existe une solution générale au problème posé par la file de priorité Q, obtenue grâce à une nouvelle analyse des conditions ayant présidé à l'introduction de Q.

La file de priorité Q a été introduite pour éviter qu'un triplet $(X, \vec{\imath}, x)$ soit extrait, puis qu'un autre triplet $(X', \vec{\imath}', x')$ extrait ultérieurement provoque l'ajout de $(X, \vec{\imath}, x'')$. En effet, cet ajout ultérieur signifie ipso facto que l'optimisation d'attribut pour les dérivations $X \rightarrow^* u$ telles que $\langle u \rangle = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$ a été incorrecte. Or une telle anomalie ne peut se produire que s'il existe $\alpha \in V^*$ tel que $X \rightarrow^+ \alpha X'$. Et dans un tel cas, on a forcément soit $\vec{\imath} < \vec{\imath}'$ (i.e. $\alpha \neq \epsilon$), soit $\vec{\imath} = \vec{\imath}'$ (i.e. $\alpha = \epsilon$, donc $X \rightarrow^+ X'$), c'est-à-dire $(X, \vec{\imath}, x) < (X', \vec{\imath}', x')$ d'après la définition de $<$ utilisée pour Q. L'extraction des triplets maximaux de Q selon la relation $<$ est donc une première solution satisfaisante du problème.

Remarque 4.1 Soient X et X' deux non-terminaux. S'il n'existe pas $\alpha \in V^*$ tel que $X \rightarrow^+ \alpha X'$, alors $(X, \vec{\imath}, x)$ peut être extrait de Q indépendamment d'éventuels triplets $(X', \vec{\imath}', x') \in Q$, et ce même si $(X, \vec{\imath}, x) < (X', \vec{\imath}', x')$.

Cette remarque fondamentale est à l'origine de la méthode suivante d'extraction des triplets de Q.

Proposition 4.1
Soient G une grammaire non-contextuelle m-bandes non-circulaire et $\Rrightarrow$ la relation sur les symboles de G définie par :

$$X \Rrightarrow Y \iff X \neq Y \text{ et } \exists \alpha \in V^*, X \rightarrow \alpha Y$$

Cette relation définit un graphe orienté dont les sommets sont les symboles de la grammaire. Soient $C_1, \dots, C_k$ les k composantes fortement connexes de ce graphe, ordonnées de manière à vérifier la propriété suivante :

$$\forall 1 \leqslant i < j \leqslant k, \forall X \in C_i, \forall Y \in C_j, X \not\Rrightarrow^+ Y$$

Soit $<_2$ la relation d'ordre suivante sur les paires de triplets $(X, \vec{\imath}, x)$ et $(X', \vec{\imath}', x')$ telles que $X \in C_j$ et $X' \in C_{j'}$:

$$(X', \vec{\imath}', x') <_2 (X, \vec{\imath}, x) \iff j < j' \text{ ou } \big(j' = j \text{ et } (X', \vec{\imath}', x') < (X, \vec{\imath}, x)\big)$$

Alors l'extraction successive des triplets maximaux de Q pour cette relation $<_2$ garantit l'extraction correcte des triplets (cf. discussion précédent la définition 1.12 p. 32 et remarque 4.1).

Démonstration : Il faut simplement montrer que, lorsque $j < j'$, les valeurs de $\vec{\imath}$ et $\vec{\imath}'$ n'ont plus aucune importance.

Si $j < j'$, alors on a toujours $X \not\Rightarrow^+ X'$ et, d'après la remarque 4.1, $(X, \vec{\imath}, x)$ peut être librement extrait de Q avant $(X', \vec{\imath}', x')$. □

Remarque 4.2 Les k composantes fortement connexes ordonnées de la proposition précédente sont obtenues grâce à l'algorithme de Tarjan (Tarjan, 1972). La figure 4.9 donne un exemple de décomposition des symboles d'une grammaire en composantes fortement connexes. Modulo une renumérotation des symboles, il est toujours possible de ranger les symboles $X_1, \ldots X_p$ d'une grammaire de façon à ce que $\{X_1, \ldots, X_{i_1}\} \in C_1$, $\{X_{i_1+1}, \ldots, X_{i_2}\} \in C_2$, ..., $\{X_{i_{k-1}+1}, \ldots, X_l\} \in C_k$ et $i < j$ implique $X_i \not\rightarrow^+ X_j$. La gestion de la relation $<$ sur les triplets au sein de chaque composante fortement connexe s'en trouve ainsi simplifiée.

Grâce aux composantes fortement connexes (déterminées statiquement lors de la lecture de la MTSAG par MTSAG2C, avant la génération de l'analyseur syntaxique), il est possible de décomposer Q en une table de files de priorité, indexée par le numéro de composante connexe (figure 4.10). Cette décomposition est duale de celle de la figure 4.8. Dans une MTSAG typique utilisée pour un problème d'analyse de séquence, il y a approximativement autant de composantes connexes que de non-terminaux (i.e. la plupart des composantes connexes sont de taille 1) et la décomposition de Q est donc bonne. La méthode de décomposition présentée ici ne possède donc pas les inconvénients de la troisième méthode présentée au début de cette section. En effet, lorsque la grammaire est non-ambiguë, l'algorithme d'extraction ne cherche plus une sous-file de priorité non vide en un temps croissant linéairement avec la taille de la chaîne analysée, mais plutôt en un temps borné par le nombre de non-terminaux de la grammaire. En pratique, ce temps est négligeable. Par ailleurs, le temps d'extraction ne se dégrade pas significativement lorsque le nombre de non-terminaux de la grammaire devient important, comme c'est le cas avec les SCFG.

Les files de priorité de chaque entrée de la table pourraient être gérées à l'aide d'un algorithme du type arbre pagode. Mais un certain nombre d'optimisations dans l'ordre de traitement des réductions dans l'algorithme GCP (optimisations rendues possibles par l'introduction de cette méthode de gestion de Q) permettent de se contenter d'une liste doublement chaînée triée et de deux pointeurs sur le début de la liste et sur la position du dernier élément inséré. Cette dernière implémentation de Q (table de listes doublement chaînées, indexée par le numéro de composante connexe) s'est invariablement révélée nettement plus rapide que la meilleure des méthodes utilisées précédemment (cf. la discussion au début de cette section) et la proportion du temps passé à gérer Q en cours d'analyse syntaxique est passée à 10 % dans le pire des cas.

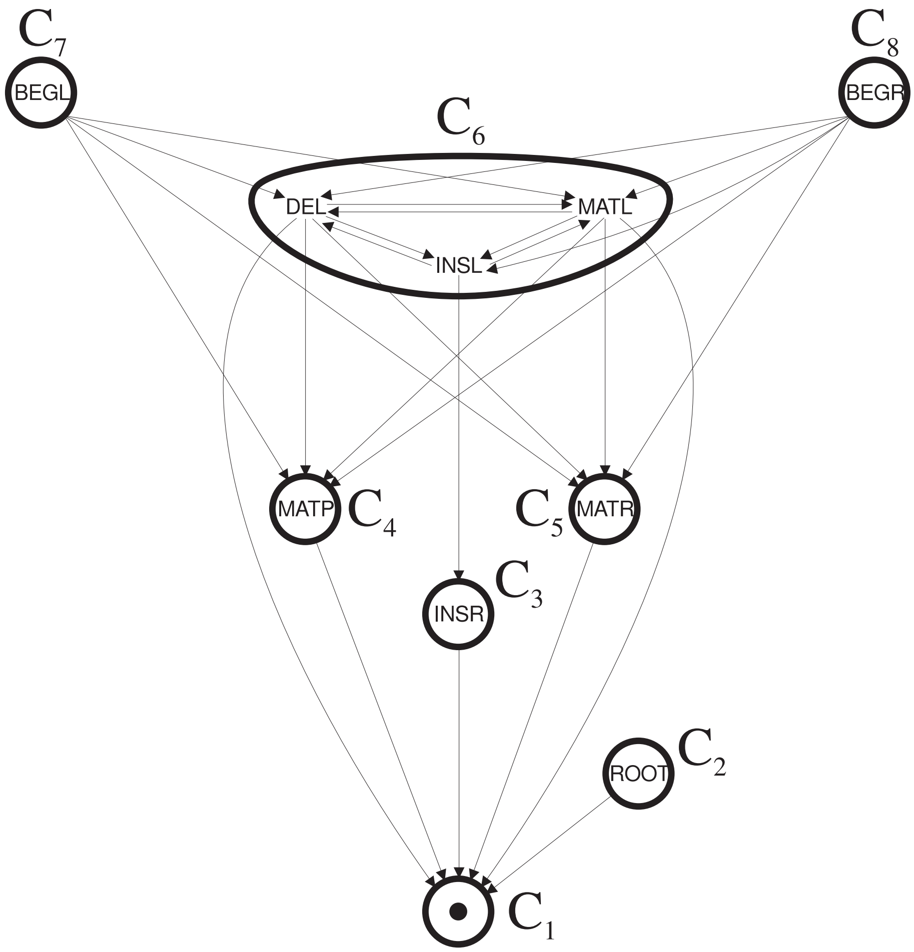


Figure 4.9: Décomposition en 8 composantes fortement connexes des symboles de la MTSAG 2-bandes des Covariance Models donnée figure 3.41 p. 163. Une seule composante fortement connexe C_1 suffit pour regrouper tous les terminaux car ils sont ajoutés à Q lors de l'initialisation pour être extraits immédiatement après. La composante fortement connexe C_6 contient 3 non-terminaux tandis que les autres contiennent au plus un non-terminal.

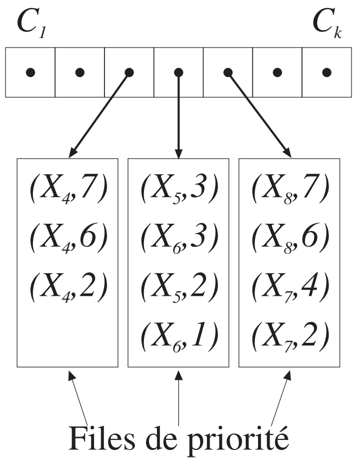


Figure 4.10: Décomposition de Q en une table de files de priorité indexée par le numéro de composante fortement connexe du symbole X des triplets $(X, \vec{\imath}, x)$ (attribut x non représenté dans la figure). L'ordre dans chaque file de priorité respecte la relation $<$ sur les triplets.

4.5 Codage des états attribués

Nous avons introduit en 4.1.1 un codage des états d'analyse $[\Delta \rightarrowtail \alpha]$. En réalité, l'algorithme GCP m-bandes manipule des états attribués de la forme $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$. Nous allons exposer le codage utilisé pour ces états attribués.

Soit $[\Delta \rightarrowtail X_1X_2 \ldots X_k, \vec{\imath}_1, x_1x_2 \ldots x_k]$ un état attribué de $T_{\vec{\jmath}}$. Dans cette section, nous noterons[4] $(\vec{\imath}_1, x_1) \centerdot (\vec{\imath}_2, x_2) \centerdot \cdots \centerdot (\vec{\imath}_k, x_k)$ la chaîne d'attributs $x_1x_2 \ldots x_k$ de cet état, pour marquer le fait que :

$$\begin{array}{llll}
x_1 & \text{est l'attribut optimal des dérivations} & X_1 \rightarrow^* u_1 & \langle u_1 \rangle = a_{\vec{\imath}_1+\vec{1}} \cdots a_{\vec{\imath}_2} \\
x_2 & \text{est l'attribut optimal des dérivations} & X_2 \rightarrow^* u_2 & \langle u_2 \rangle = a_{\vec{\imath}_2+\vec{1}} \cdots a_{\vec{\imath}_3} \\
\vdots & & \vdots & \vdots \\
x_k & \text{est l'attribut optimal des dérivations} & X_k \rightarrow^* u_k & \langle u_k \rangle = a_{\vec{\imath}_k+\vec{1}} \cdots a_{\vec{\jmath}}
\end{array}$$

L'état $[\Delta \rightarrowtail X_1X_2 \ldots X_k, \vec{\imath}_1, x_1x_2 \ldots x_k]$ pourra donc être noté

$$[\Delta \rightarrowtail X_1X_2 \ldots X_k, \vec{\imath}_1, (\vec{\imath}_1, x_1) \centerdot (\vec{\imath}_2, x_2) \centerdot \cdots \centerdot (\vec{\imath}_k, x_k)].$$

Lorsque $k = 1$, l'algorithme GCP ajoute au plus un état $\left[\Delta \rightarrowtail X_1, \vec{\imath}_1, (\vec{\imath}_1, x_1)\right]$ à $T_{\vec{\jmath}}$. Cet état est donc entièrement déterminé par la donnée du triplet $(X_1, \vec{\imath}_1, x_1)$ et de $\Delta_{\vec{\imath}_1}$.

[4] Voir également la démonstration du théorème 2.2 p. 83 pour une explication de la notation.

Par contre, lorsque $k > 1$, la décomposition de $a_{\vec{\imath}_1+\vec{1}} \dots a_{\vec{\jmath}}$ en

$$a_{\vec{\imath}_1+\vec{1}} \dots a_{\vec{\imath}_2} \quad a_{\vec{\imath}_2+\vec{1}} \dots a_{\vec{\imath}_3} \quad \dots \quad a_{\vec{\imath}_k+\vec{1}} \dots a_{\vec{\jmath}}$$

est en général multiple[5]. De nombreux états attribués ayant le même état d'analyse $[\Delta \rightarrowtail X_1 \dots X_k, \vec{\imath}_1]$, mais des chaînes d'attributs différentes, seront donc stockés dans $T_{\vec{\jmath}}$.

Remarque 4.3 La présence d'un état

$$\left[\Delta \rightarrowtail X_1 X_2 \dots X_{k-1} X_k, \vec{\imath}_1, (\vec{\imath}_1, x_1) \centerdot (\vec{\imath}_2, x_2) \centerdot \dots \centerdot (\vec{\imath}_{k-1}, x_{k-1}) \centerdot (\vec{\imath}_k, x_k)\right]$$

dans $T_{\vec{\jmath}}$ signifie obligatoirement qu'un état

$$\left[\Delta \rightarrowtail X_1 X_2 \dots X_{k-1}, \vec{\imath}_1, (\vec{\imath}_1, x_1) \centerdot (\vec{\imath}_2, x_2) \centerdot \dots \centerdot (\vec{\imath}_{k-1}, x_{k-1})\right]$$

est dans $T_{\vec{\imath}_k}$. Pour éviter de stocker des chaînes d'attributs redondantes d'un état à un autre, nous devons utiliser un codage des ensembles de chaînes d'attributs des états d'analyse de $T_{\vec{\jmath}}$.

Notation 4.1 Soit L un ensemble de chaînes d'attributs et $(\vec{\imath}, x)$ un couple formé d'un index m-bandes et d'un attribut. La notation $L \centerdot (\vec{\imath}, x)$ représente un nouvel ensemble L' de chaînes d'attributs, défini par :

$$L' = L \centerdot (\vec{\imath}, x) = \{\lambda \centerdot (\vec{\imath}, x) \mid \lambda \in L\}$$

Définition 4.1 (Ensembles L_A, L_I et L_G)
Soit $L_A([\Delta \rightarrowtail \alpha, \vec{\imath}], \vec{\jmath})$ l'ensemble des chaînes d'attributs λ des états $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ appartenant à $T_{\vec{\jmath}}$.

Soit $L_I([\Delta \rightarrowtail \alpha], \vec{\jmath})$ l'ensemble des couples $(\vec{\imath}, L_A([\Delta \rightarrowtail \alpha, \vec{\imath}], \vec{\jmath}))$ tels que l'ensemble $L_A([\Delta \rightarrowtail \alpha, \vec{\imath}], \vec{\jmath})$ est valide (i.e. est non vide) dans $T_{\vec{\jmath}}$. Cet ensemble L_I représente tous les états attribués de $T_{\vec{\jmath}}$ de même $[\Delta \rightarrowtail \alpha]$.

Soit $L_G(\vec{\jmath}, X)$ l'ensemble des couples $([\Delta \rightarrowtail \alpha], L_I([\Delta \rightarrowtail \alpha], \vec{\jmath}))$ valides de $T_{\vec{\jmath}}$ tels que $[\Delta \rightarrowtail \alpha]$ est prolongeable par le symbole X (i.e. $\mathsf{Goto}([\Delta \rightarrowtail \alpha], X) \neq \emptyset$).

En pratique, L_G sera implémentée sous la forme d'une table indexée par X dans $T_{\vec{\jmath}}$, L_A et L_I sous la forme de listes.

[5]Cette multiplicité est à l'origine de la dépendance en r, le nombre maximum de non-terminaux des parties droites des règles de dérivation, dans la complexité de l'algorithme GCP (cf. théorème 1.6 p. 47).

Grâce aux ensembles précédents, l'algorithme donné en section 2.2 p. 80 peut se réécrire de la manière indiquée dans la figure 4.11. L'intérêt de chacun des ensembles introduits précédemment est le suivant :

L_G permet de retrouver en $O(1)$ l'ensemble des états prolongeables par X utilisés à l'étape 5 de l'algorithme. Par ailleurs, il n'est alors plus nécessaire de spécifier le code $[\Delta \rightarrowtail \alpha]$ dans chaque état d'analyse attribué (gain de place). Les couples $\big([\Delta \rightarrowtail \alpha], L_I([\Delta \rightarrowtail \alpha], \vec{\jmath})\big)$ de $L_G(\vec{\jmath}, X)$ pourront être représentés sous la forme d'un entier (le code de $[\Delta \rightarrowtail \alpha]$) et d'un pointeur sur la début de la liste $L_I\big([\Delta \rightarrowtail \alpha], \vec{\jmath}\big)$;

L_I permet de ne pas spécifier l'index $\vec{\imath}$ dans chaque état attribué $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ (gain de place), et de ne pas changer de liste $T_{\vec{\imath}}$ d'états à prolonger à chaque instant dans l'étape 5 (gain de temps). Les couples $\big(\vec{\imath}, L_A([\Delta \rightarrowtail \alpha, \vec{\imath}], \vec{\jmath})\big)$ pourront être représentés sous la forme d'un entier (le code de $\vec{\imath}$) et d'un pointeur sur le début de la liste $L_A\big([\Delta \rightarrowtail \alpha, \vec{\imath}], \vec{\jmath}\big)$;

L_A tient compte de la remarque 4.3 et factorise l'ensembles des chaînes d'attributs d'un état d'analyse $[\Delta \rightarrowtail \alpha, \vec{\imath}]$ de $T_{\vec{\jmath}}$ par l'intermédiaire de l'opération $\centerdot$. En effet, l'opération de concaténation $L_A\big([\Delta \rightarrowtail \alpha, \vec{h}], \vec{\imath}\big) \centerdot (\vec{\imath}, x)$ de l'étape 5 peut être représentée sous la forme d'un triplet contenant le code de $\vec{\imath}$, l'attribut x, et un pointeur sur le début de la liste $L_A\big([\Delta \rightarrowtail \alpha, \vec{h}], \vec{\imath}\big)$.

Notons que l'ensemble $\big\{L_A\big([\Delta \rightarrowtail X, \vec{\imath}], \vec{\jmath}\big) \bigm| X \in V\big\}$ n'est pas sans rappeler l'entrée $\mathfrak{T}_{\vec{\imath},\vec{\jmath}}$ de la table utilisée pour reconstruire les arbres de dérivation (cf. section 2.2.1 p. 81). Cette remarque est utilisée dans le noyau des analyseurs syntaxiques pour économiser de la place.

4.5.1 Optimisation de l'allocation mémoire

Comme seuls $\Delta_{\vec{\jmath}}$ et les états prolongeables sont utiles à la poursuite de l'analyse syntaxique, il est inutile de conserver dans $T_{\vec{\jmath}}$ les états représentés par $L_I\big([\Delta \rightarrowtail \alpha], \vec{\jmath}\big)$ et tels que $[\Delta \rightarrowtail \alpha]$ n'est prolongeable par aucun symbole. En fait, ces états ne sont pas réellement alloués, ce qui évite d'avoir à les libérer (c'est le sens du test "Si $[\Delta \rightarrowtail \alpha X]$ est prolongeable par un symbole..." de la figure 4.11).

Si $\alpha = \beta a$ et si $[\Delta \rightarrowtail \beta]$ est prolongeable uniquement par des terminaux m-bandes, alors il est également possible de libérer certaines listes $L_I\big([\Delta \rightarrowtail \beta], \vec{\imath}\big)$ lorsque $L_I\big([\Delta \rightarrowtail \alpha], \vec{\jmath}\big)$ est libéré (où $\vec{\imath}$ vérifie $a = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$). Le même principe est applicable récursivement aux parents de $[\Delta \rightarrowtail \beta]$. Lors des tests de libération, il faut prendre garde au fait qu'un alphabet m-bandes n'induit pas un monoïde libre et qu'un état peut être prolongeable par plusieurs terminaux m-bandes différents. Avant de libérer cet état, il faut s'assurer que toutes les prolongations ont été effectuées.

Il nous faut mentionner ici le mécanisme particulier utilisé pour compenser la lenteur extrême des fonctions standard d'allocation mémoire *malloc()* et *free()*. La plupart des structures de données utilisées par l'analyseur syntaxique, et en tout cas toutes les structures allouées et libérées fréquemment, possèdent un pointeur vers un élément

Soit Q une file de priorité de triplets $(X, \vec{\imath}, x)$ et sa fonction d'extraction $\mathtt{Extract}$. Soit $\Delta_{\vec{0}} = \{A \in V_N \mid \exists\beta, S \rightarrow^* A\beta\}$. Pour $\vec{\jmath}$ allant de $\vec{0}$ (exclus) à $\vec{n}$ (inclus), en ordre croissant d'index m-bandes, exécuter les étapes suivantes :

1. Initialiser les listes L_I et L_A :
   $\forall[\Delta \rightarrowtail \alpha], \forall\vec{\imath}, L_I(([\Delta \rightarrowtail \alpha], \vec{\jmath}) := \emptyset, L_A(([\Delta \rightarrowtail \alpha, \vec{\imath}], \vec{\jmath}) := \emptyset.$
2. Initialiser la file Q: $Q := \left\{(a, \vec{\imath}, S_A(a)) \;\middle|\; \vec{\imath} < \vec{\jmath} \wedge a = a_{\vec{\imath}+\vec{1}} \ldots a_{\vec{\jmath}} \in \vec{V}_T\right\}$
3. Extraction du triplet à traiter en priorité :
   $(X, \vec{\imath}, x) := \mathtt{Extract}(Q)$
4. Création des états attribués commençant par X :
   Soit $[\Delta \rightarrowtail X] = \mathtt{Initial}(\Delta_{\vec{\imath}}, X)$.
   Pour tout $A \in \mathtt{Reduce}([\Delta \rightarrowtail X])$, calculer le triplet $(A, \vec{\imath}, x') := (A, \vec{\imath}, f_{A\rightarrow X}(x))$. Si $\mathcal{C}_A^{\mathcal{S}}(\vec{\imath}, \vec{\jmath}, x')$ est vrai alors :

   S'il existe déjà un triplet $(A, \vec{\imath}, x'')$ appartenant à Q, remplacer l'attribut x'' de ce triplet par $\mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x'', x')$.
   Si aucun triplet de ce type n'existe dans Q, ajouter $(A, \vec{\imath}, x')$ à Q.

   Si $[\Delta \rightarrowtail X]$ est prolongeable par un symbole alors :

   $L_A([\Delta \rightarrowtail X, \vec{\imath}], \vec{\jmath}) := L_A([\Delta \rightarrowtail X, \vec{\imath}], \vec{\jmath}) \cup (\vec{\imath}, x)$
   $L_I([\Delta \rightarrowtail X], \vec{\jmath}) := L_I([\Delta \rightarrowtail X], \vec{\jmath}) \cup (\vec{\imath}, L_A([\Delta \rightarrowtail X, \vec{\imath}], \vec{\jmath}))$
5. Prolongation d'états attribués par X :
   Pour $([\Delta \rightarrowtail \alpha], L_I([\Delta \rightarrowtail \alpha], \vec{\imath})) \in L_G(\vec{\imath}, X)$

   Soit $[\Delta \rightarrowtail \alpha X] = \mathtt{Goto}([\Delta \rightarrowtail \alpha], X)$.
   Pour $(\vec{h}, L_A([\Delta \rightarrowtail \alpha, \vec{h}], \vec{\imath})) \in L_I([\Delta \rightarrowtail \alpha], \vec{\imath})$

   Pour tout $A \in \mathtt{Reduce}([\Delta \rightarrowtail \alpha X])$ et pour chaque état $[\Delta \rightarrowtail \alpha X, \vec{h}, \lambda x]$ tel que $\lambda \in L_A([\Delta \rightarrowtail \alpha, \vec{h}], \vec{\imath})$, calculer le triplet $(A, \vec{h}, x') := (A, \vec{h}, f_{A\rightarrow\alpha X}(\lambda x))$. Si $\mathcal{C}_A^{\mathcal{S}}(\vec{h}, \vec{\jmath}, x')$ est vrai alors :

   S'il existe déjà un triplet $(A, \vec{h}, x'')$ appartenant à Q, remplacer l'attribut x'' de ce triplet par $\mathcal{C}_A(\vec{h}, \vec{\jmath}, x'', x')$.
   Si aucun triplet de ce type n'existe dans Q, ajouter $(A, \vec{h}, x')$ à Q.

   Si $[\Delta \rightarrowtail \alpha X]$ est prolongeable par un symbole alors :

   $L_A([\Delta \rightarrowtail \alpha X, \vec{h}], \vec{\jmath}) := L_A([\Delta \rightarrowtail \alpha X, \vec{h}], \vec{\jmath}) \cup L_A([\Delta \rightarrowtail \alpha, \vec{h}], \vec{\imath}) \centerdot (\vec{\imath}, x)$
   $L_I([\Delta \rightarrowtail \alpha X], \vec{\jmath}) := L_I([\Delta \rightarrowtail \alpha X], \vec{\jmath}) \cup (\vec{h}, L_A([\Delta \rightarrowtail \alpha X, \vec{h}], \vec{\jmath}))$
6. Répéter les étapes 3 à 6 tant que Q n'est pas vide;
7. Calculer $\Delta_{\vec{\jmath}}$ et, pour tout $X \in V$, $L_G(\vec{\jmath}, X)$.

Le mot m-bandes $a_{\vec{1}} \ldots a_{\vec{n}}$ appartient à $L(\vec{G})$ si et seulement si le triplet $(S, \vec{0}, x)$ a été extrait de Q pendant le calcul de $T_{\vec{n}}$. On dit alors que le mot a été reconnu par l'algorithme et qu'il a un attribut optimal x.

Figure 4.11: Utilisation d'ensembles L_A, L_I et L_G à la place de $T_{\vec{\jmath}}$ dans l'algorithme GCP m-bandes.

suivant. Dans une structure d'un type T donné, ce pointeur est habituellement utilisé pour gérer les listes de structures de type T. Lorsqu'une structure de type T est libérée, son pointeur vers l'élément suivant est réutilisé pour la chaîner dans une liste des structures libres de type T. L'allocation d'une structure de type T se fait prioritairement dans la liste des structures libres. Si cette liste est vide, alors la fonction *malloc()* est appelée pour allouer simultanément plusieurs centaines de structures qui sont immédiatement chaînées dans la liste de structures libres. Ainsi, le coût d'appel à *malloc()* a un impact des centaines de fois plus faible qu'un appel systématique. La fonction *free()* n'est jamais appelée dans le cas des structures allouées fréquemment comme celle des états d'analyse car tous les états libérés au rang $\vec{\jmath}$ sont en général réutilisés au rang suivant. Un mécanisme est cependant prévu pour libérer la mémoire allouée en fin d'analyse syntaxique, une fois l'arbre de dérivation construit (cf. section A.6.7 p. 264).

4.6 Conclusion

L'implémentation directe de l'algorithme GCP suivant le synoptique de la figure 4.11 et les remarques de ce chapitre n'engendrera probablement pas un analyseur syntaxique aussi rapide que ceux générés par MTSAG2C. En effet, de nombreux points importants restent encore dans l'ombre, comme l'écriture des fonctions C d'évaluation d'attributs à partir du code C des fonctions d'évaluation d'attributs donné dans une MTSAG. En particulier, il est capital de transférer un maximum de boucles serrées (tight loops) dans le code des fonctions d'évaluation d'attributs plutôt qu'en dehors, pour éviter de multiplier les appels de fonctions. Ensuite, il est important d'organiser les calculs de façon à minimiser les temps de manipulation de la file de priorité Q. Enfin, le code et les appels de fonctions doivent être organisés de façon à maximiser le degré de réutilisation des caches des données et du code, disponibles sur toutes les architecture modernes, sachant que les optimisations ne doivent pas être spécifiques à une architecture donnée et que le code C doit rester portable et lisible.

Bien qu'aucun chiffre précis ne soit disponible, l'expérience accumulée au cours du développement de l'analyseur syntaxique présenté dans cette thèse permet d'estimer qu'une implémentation directe des algorithmes donnés dans les chapitres précédents et des optimisations de ce chapitre serait probablement deux à trois fois plus lente et gourmande en mémoire que les analyseurs syntaxiques générés par MTSAG2C. Rappelons pour mémoire que les premières versions d'analyseurs utilisant l'algorithme GCP étaient plusieurs dizaines de fois plus lentes et gourmandes en mémoire que les versions actuelles.

Chapitre 5

Repères Théoriques

Introduction

Nous avons regroupé dans ce chapitre tous les résultats et remarques théoriques liés aux chapitres 1 et 2, épargnant ainsi le lecteur peu familier des concepts associés aux langages formels et à l'analyse syntaxique. Nous présumerons en revanche que le lecteur de ce chapitre possède une bonne connaissance des langages formels et des algorithmes d'analyse syntaxique des grammaires non-contextuelles, en particulier de l'algorithme de Earley.

Dans une première section, nous énoncerons les propriétés nécessaires à un algorithme d'analyse syntaxique utilisé dans le contexte de nos problèmes de recherche d'arbres d'analyse optimaux pour des grammaires S-attribuées ambiguës. À la lumière de ces propriétés, nous comparerons l'algorithme GCP à quelques autres algorithmes d'analyse syntaxique.

La deuxième section sera consacrée à l'exposition de quelques extensions possibles de l'algorithme GCP et du formalisme des MTSAG. Nous nous intéresserons en particulier à l'introduction de mécanismes de lookahead et de gestion des dérivations vides.

La troisième section contient quelques conseils et remarques relatifs à l'écriture des MTSAG.

La dernière section regroupe plusieurs résultats théoriques concernant la classe des langages générés par les grammaires non-contextuelles m-bandes.

5.1 Analyse syntaxique et algorithme GCP

L'analyse syntaxique d'une chaîne pour une grammaire G donnée, c'est-à-dire la détermination d'un arbre de dérivation de cette chaîne, n'est en général pas réalisée directement. Elle est plutôt une conséquence des étapes de calcul d'un algorithme de reconnaissance, c'est-à-dire d'un algorithme qui détermine si la chaîne appartient au langage généré par G. Une remarque fondamentale s'impose alors quant à la nature des problèmes d'analyse syntaxique traités dans cette thèse.

Remarque 5.1 Soit une MTSAG $\vec{G}$ et un mot m-bandes $a_{\vec{1}} \dots a_{\vec{n}} \in \langle \vec{V}_T^* \rangle$. Notre problème n'est pas de savoir si le mot appartient au langage $L(\vec{G})$ généré par $\vec{G}$. En général, les MTSAG utilisées dans les problèmes d'analyse de séquences sont telles que tout mot m-bandes appartient à $L(\vec{G})$. Le langage généré ne présente donc aucun intérêt, puisque c'est $\langle \vec{V}_T^* \rangle$, et la reconnaissance des mots d'un tel langage peut être effectuée en temps constant. Le problème est plutôt de savoir quel est l'arbre de dérivation possédant le meilleur attribut à la racine.

Est donc inutilisable tout algorithme de reconnaissance qui ne considère pas explicitement un codage de l'ensemble des arbres de dérivation possibles au cours de son exécution. Sont donc a priori exclus les algorithmes dérivés de l'algorithme de Valiant (Valiant, 1975; Harrison, 1978) tels que l'algorithme de Rytter (Rytter, 1995), même si leur complexité asymptotique dans le cas général ($O(n^{2+\epsilon})$) est inférieure à la complexité de l'algorithme de Earley. De toute façon, cette complexité asymptotique n'est vraiment rentable que pour de très longues chaînes.

Pour être utilisable avec nos problèmes d'analyse de séquences, un algorithme de reconnaissance doit satisfaire les conditions suivantes :

1. Il doit potentiellement considérer toutes les dérivations d'une chaîne afin de pouvoir sélectionner la meilleure ;

2. La classe de grammaires non-contextuelles acceptées doit être la plus grande possible et non pas seulement restreinte à certaines formes normales qui, bien que générant le même langage qu'une grammaire non normalisée de départ, changent la structure des dérivations et donc la nature du problème ;

3. L'algorithme doit exploiter, autant que faire se peut, les propriétés particulières de la grammaire (non-ambiguïté, linéarité, LR, etc.) afin de fournir de bonnes performances à l'analyseur m-bandes (cf. théorème 2.2 p. 83) ;

4. Il doit autoriser une implémentation rapide et économe en mémoire, même pour des chaînes analysées relativement longues[1] (> 100 lexèmes) et des grammaires à ambiguïté exponentielle de taille importante (605 règles réelles pour la grammaire de la figure 3.9 p. 104, ou plus de 4000 règles pour une généralisation du modèle de Sankoff section 3.4 p. 108).

L'algorithme GCP remplit bien entendu les quatre conditions précédentes :

1. C'est un algorithme tabulaire. Il utilise donc un codage de la forêt d'arbres de dérivation. En l'absence de contrainte de sélection, tous les arbres de dérivations sont accessibles. De plus, pour tous $\vec{\imath}, \vec{\jmath}$ et α, au plus un état d'analyse $[\Delta \rightarrowtail \alpha, \vec{\imath}]$ est ajouté à $T_{\vec{\jmath}}$ (indépendamment des chaînes d'attributs). Donc les sous-dérivations identiques ne sont pas calculées plus d'une fois ;

[1] Traditionnellement, l'analyse syntaxique des grammaires non-contextuelles a été utilisée dans le cadre de la théorie du langage naturel. Les phrases d'une langue, quelle que soit cette langue, sont significativement plus courtes que les séquences biologiques (un ou deux ordres de grandeur de différence dans les applications présentées ici).

2. Les seules restrictions imposées aux grammaires non-contextuelles sont la non-circularité et l'absence de dérivation vide $A \to^* \epsilon$. La non-circularité est nécessaire quel que soit l'algorithme utilisé sinon le nombre d'arbres de dérivation devient infini et rend impossible la recherche du meilleur arbre de dérivation[2]. Nous verrons plus loin (section 5.2.2 p. 212) comment traiter les grammaires possédant des dérivations $A \to^* \epsilon$;

3. La propriété d'utilité a priori des états d'analyse générés (cf. théorème 1.1 p. 28) est une forme de filtrage top-down qui empêche la construction de sous-dérivations inutiles au regard de la portion déjà lue de la chaîne analysée. Grâce à cette propriété, les complexités de l'algorithme GCP diminuent lorsque la grammaire vérifie de bonnes propriétés (cf. table 1.1 p. 38) ;

4. Les différents exemples donnés dans cette thèse ont montré que l'implémentation actuelle de l'algorithme GCP est tout à fait satisfaisante. Le précalcul des tables d'analyse, la non-génération des états n'appartenant pas au noyau, ainsi que la factorisation des parties droites permise par les états $[\Delta \rightarrowtail \alpha]$, jouent un rôle important dans la qualité des résultats obtenus.

L'algorithme GCP ne prétend certainement pas être imperfectible. En particulier, son utilisation avec des grammaires dont toutes les projections sont $LR(k)$ et non-récursives à droite n'engendre pas des analyseurs particulièrement performants. Il serait par exemple ridicule d'utiliser l'algorithme GCP à la place d'un automate LR (tel que ceux générés par YACC) pour analyser une chaîne suivant une grammaire $LALR(1)$ (cf. section 1.7.1 p. 49). En définitive, il n'est pas surprenant qu'un algorithme spécifique soit plus performant qu'un algorithme général lorsqu'il est appliqué à la classe de grammaires pour laquelle il est prévu. Nous allons donc uniquement comparer l'algorithme GCP à quelques autres algorithmes généraux, sans prétendre à l'exhaustivité.

Algorithme CYK La version standard de l'algorithme de Cocke–Younger–Kasami ne fonctionne que sur des grammaires sous forme normale de Chomsky. Il est aisé de modifier cette version standard pour lui permettre de traiter n'importe quelle grammaire non-contextuelle. C'est un algorithme purement bottom-up sans filtrage top-down. Ses performances sont donc nécessairement inférieures à celles de l'algorithme GCP, sauf avec les rares grammaires pour lesquelles le filtrage top-down est inutile.

Algorithme de Earley C'est un proche ancêtre de l'algorithme GCP. Il lui est cependant inférieur par définition puisque l'algorithme GCP est une forme optimisée de l'algorithme de Earley : factorisation des parties droites dans les états $[\Delta \rightarrowtail \alpha]$, non-génération des états n'appartenant pas au noyau, précalcul de tables d'analyse.

[2]Sauf si les fonctions d'évaluation d'attributs sont absorbantes le long d'un cycle, une condition bien difficile à vérifier étant donnée la généralité de ces fonctions.

Algorithme de Graham–Harrison–Ruzzo La plupart des techniques présentées dans (Graham *et al.*, 1980) ont une portée générale. En particulier, un précalcul des tables d'analyse y est suggéré et des implémentations performantes possibles sont décrites. Nous utilisons des variantes des techniques de cet article dans l'implémentation de l'algorithme GCP. Par contre, les états générés par l'algorithme décrit dans l'article sont essentiellement équivalents aux états générés par l'algorithme de Earley et ils ne bénéficient donc pas de la factorisation des parties droites et de la génération des seuls états appartenant au noyau offertes par l'algorithme GCP.

Algorithme LRE(k) Cet algorithme a été proposé par McLean et Horspool (McLean & Horspool, 1996). Il n'est d'ailleurs pas sans rappeler un analyseur LR(0) à remontée récursive (*recursive ascent LR(0) parser*) proposé par Leermakers (Leermakers, 1992). Le nom LRE(k) reflète le fait que la méthode proposée peut être considérée comme une combinaison d'analyse LR(k) et d'analyse de Earley : les états $[A \rightarrow \alpha \cdot \beta]$ générés par l'algorithme de Earley sont regroupés dans les états d'un analyseur LR(k). Grâce à ce regroupement, les tables habituellement construites pour un analyseur LR(k) peuvent être utilisées pour déterminer les actions de l'analyseur LRE(k). Les états n'appartenant pas au noyau ne sont pas générés. Si $[A \rightarrow \alpha \cdot \beta]$ avec $\alpha \neq \epsilon$ est un état généré par l'algorithme de Earley, alors les algorithmes LRE(k) et GCP contiennent tous deux une représentation codée de cet état. Notons que l'algorithme LRE(k) ne souffre pas des problèmes d'explosion combinatoire du nombre d'états de l'algorithme GCP (cf. section 4.2 p. 186). À première vue, l'algorithme GCP semble plus performant que l'algorithme LRE(k) lorsque la grammaire comporte de nombreuses règles ayant en commun des préfixes de parties droites, ce qui est le cas avec la plupart des grammaires utilisées en analyse de séquences. En revanche, l'algorithme LRE(k) devrait dépasser l'algorithme GCP lorsque la grammaire à analyser est proche d'une grammaire LR(k) ou possède peu de préfixes factorisables. Cependant, les tests ne confirment pas ces impressions préliminaires car notre implémentation de l'algorithme GCP est près de 30 fois plus rapide que l'implémentation de l'algorithme LRE(k) écrite par McLean et Horspool pour la grammaire ambiguë $S \rightarrow SS \mid a$. Les tests effectués avec la grammaire du langage C ne permettent pas de départager l'analyseur généré par MTSAG2C et l'implémentation de McLean et Horspool. Les différences constatées proviennent sans doute de la qualité inégale des implémentations car l'algorithme LRE(k) n'a, a priori, aucune raison d'être 30 fois plus lent que l'algorithme GCP pour la grammaire $S \rightarrow SS \mid a$.

Algorithme de Tomita Cet algorithme utilise comme le précédent des états LR(k) et une table d'analyse générée pour un analyseur LR(k). Contrairement aux algorithmes LR(k) habituels, l'algorithme de Tomita est capable de gérer le non-déterminisme de la table (conflits shift/reduce ou reduce/reduce) en utilisant une pile ayant une structure particulière (graph-structured stack) et en analysant en parallèle les différentes possibilités (Tomita, 1987, 1988). L'analyse en parallèle exécutée par l'algorithme de Tomita rend difficile l'appel de fonctions d'évaluation d'attributs. Ce problème a reçu une solution dont la complexité est cependant mal maîtrisée (Piastra & Bolognesi, 1991). L'algorithme de Tomita a été surtout utilisé dans le cadre des langages naturels, c'est-à-dire pour la reconnaissance de chaînes courtes selon des grammaires moins ambiguës que celles utilisées en analyse de séquences. Nous avons comparé les peformances de l'algorithme GCP à celles d'un analyseur implémentant en langage C l'algorithme de Tomita[3]. Le test fut effectué sur la grammaire ambiguë $S \rightarrow SS \mid a$ et une chaîne contenant 100 lettres a. L'analyseur implémentant l'algorithme de Tomita a effectué l'analyse en 9 secondes et 7 Mo, contre moins d'un dixième de seconde et 110 Ko pour l'algorithme GCP. La différence constatée est donc d'au moins deux ordres de grandeur. La poursuite des tests avec des chaînes de plus en plus longues a permis de confirmer la complexité en $O(n^3)$ en temps et $O(n^2)$ en espace de l'algorithme GCP sur cette grammaire, tandis que l'implémentation de l'algorithme de Tomita utilisée a exhibé des complexités de $O(n^4)$ en temps et $O(n^3)$ en espace[4].

Algorithme de Leo Cet algorithme est une extension de l'algorithme de Earley offrant une complexité d'analyse linéaire en temps lorsque la grammaire est LR(k), sans même utiliser de *lookahead* (Leo, 1991). Pour obtenir ce résultat, Leo utilise des états spéciaux permettant de supprimer le calcul de certains états de l'algorithme de Earley et de certaines dérivations. Cet algorithme n'est donc pas adapté à nos problèmes (cf. condition 1 énoncée au début de cette section).

Algorithme de Kruseman Aretz Dans (Kruseman Aretz, 1989), Kruseman Aretz explique comment générer plus efficacement les états de l'algorithme de Earley. Il propose en particulier de prolonger immédiatement les états prolongeables par des terminaux afin de détecter au plus tôt les incompatibilités avec la chaîne analysée. Cette méthode est théoriquement applicable à l'algorithme GCP. Nous avons cependant vérifié que l'implémentation de ce système de prolongation au plus tôt des états d'analyse avait en fait des conséquences néfastes sur l'efficacité de nos analyseurs : des listes d'analyse différentes doivent alors être mises à jour simultanément, ce qui interdit l'utilisation de certaines optimisations lors de l'implémentation et met à rude épreuve les caches des processeurs.

Algorithme de Voisin Cet algorithme décrit dans (Voisin, 1988) (voir aussi (Nederhof, 1993)) est proche de l'algorithme GCP. Nous précalculons cependant beaucoup plus d'informations, avec en contrepartie un risque d'explosion combinatoire du nombre d'états d'analyse s'ils sont générés statiquement, mais pas s'ils sont générés dynamiquement. En définitive, l'algorithme GCP conserve les avantages de l'algorithme de Voisin en termes de nombre d'états d'analyse générés, tout en étant plus rapide en pratique grâce aux précalculs réalisés.

[3] L'analyseur utilisé s'appelle *tom* est publiquement disponible dans ftp.cs.cmu.edu/user/ai/areas/nlp/parsing/tom.

[4] Ce qui montre que cette implémentation de l'algorithme de Tomita n'était pas optimale.

Il apparaît donc que l'algorithme GCP intègre de nombreux perfectionnements dont il n'a pas l'exclusivité. Ces perfectionnements, associés à une implémentation pointue, contribuent à faire de MTSAG2C un générateur d'analyseurs syntaxiques difficiles à concurrencer dès que les MTSAG utilisées sont ambiguës. Il serait cependant intéressant d'examiner ultérieurement de plus près les analogies entre l'algorithme GCP et l'algorithme LRE(k) afin de faire profiter ce dernier de notre implémentation performante de l'algorithme GCP.

Remarquons enfin qu'il est possible d'arguer que les comparaisons chiffrées avec un autre algorithme pourraient injustement avantager l'algorithme GCP par le fait qu'il n'accepte que les grammaires non-contextuelle propres alors que son concurrent est (peut-être) ralentit par une machinerie nécessaire au traitement des grammaires non-contextuelles non-propres. Les écarts constatés tendent cependant à montrer que cela ne peut constituer l'explication principale pour les algorithmes auxquels l'argument est applicable.

5.2 Extensions de l'algorithme GCP et des MTSAG

Nous allons d'abord décrire la méthode de lookahead actuellement implémentée dans nos analyseurs. Nous expliquerons également comment transformer l'algorithme GCP pour permettre l'utilisation de MTSAG contenant des dérivations vides $A \to \epsilon$. Nous commenterons ensuite une possible extension du formalisme des MTSAG aux grammaires L-attribuées au lieu des simples grammaires S-attribuées.

5.2.1 Introdution d'un lookahead dans l'algorithme GCP

Le lookahead est une technique standard en analyse syntaxique qui permet de lever des ambiguïtés de dérivation, et plus généralement d'accélérer les analyseurs syntaxiques en évitant certains calculs inutiles. La forme de lookahead implémentée dans MTSAG2C répond à la question suivante :

> Étant donné un état d'analyse $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ sur le point d'être ajouté à $T_{\vec{\jmath}}$, faut-il ajouter cet état étant donnée la suite $a_{\vec{\jmath}+\vec{1}} \ldots a_{\vec{n}}$ du mot m-bandes analysé ?

Pour une raison d'encombrement des tables de lookahead, nous avons choisi de limiter le lookahead à un token sur chaque bande. Rappelons que la problématique classique du lookahead est ici compliquée du fait de l'usage de plusieurs bandes. Nous allons construire une fonction $\mathsf{LookAhead}$ qui, étant donné un état d'analyse $[\Delta \rightarrowtail \alpha]$, retourne l'ensemble des symboles qu'il est raisonnable d'attendre après cet état. Cette fonction sera utilisée comme suit en cours d'analyse syntaxique :

> Soit $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ un état attribué m-bandes sur le point d'être ajouté à $T_{\vec{\jmath}}$. Si le terminal $a_{\vec{\jmath}+\vec{1}} \ldots a_{\vec{\jmath}+\vec{1}}$ n'appartient pas à $\mathsf{LookAhead}([\Delta \rightarrowtail \alpha])$, alors il est inutile d'ajouter $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ à $T_{\vec{\jmath}}$.

Avant de décrire l'algorithme permettant de calculer la fonction $\mathsf{LookAhead}$, nous devons introduire une fonction Follow, classique en analyse syntaxique (Aho & Ullman, 1972), et définir une relation d'inclusion sur les terminaux d'une grammaire m-bande.

Définition 5.1
Soit Follow *la fonction qui prend en paramètre un non-terminal* A *et retourne l'ensemble des terminaux* a *pouvant apparaître à droite de* A *dans une dérivation, c'est-à-dire l'ensemble des terminaux* a *tels que il existe* α *et* β *tels que* $S \rightarrow^* \alpha A a \beta$. *Par convention, si* $S \rightarrow^* \alpha A$, *alors* ϵ *appartient à* $\mathsf{Follow}(A)$.

L'algorithme de calcul de Follow est classique (Aho & Ullman, 1972; Aho *et al.*, 1988) mais nous l'avons accéléré grâce à des techniques empruntées à (Jourdan & Parigot, 1990).

Définition 5.2
Soit a *et* b *deux terminaux* m*-bandes. On dit que* $a \subset b$ *si quel que soit* $k \in [1, \ldots, m]$, $a^{(k)} = \epsilon$ *ou* $a^{(k)} = b^{(k)} \neq \epsilon$. *Notons que tout terminal* m*-bandes* a *vérifie* $a \subset a$.

Intuitivement, cette définition signifie qu'un terminal m-bandes a est inclus dans b dès qu'il est obtenu en remplaçant certains des tokens de b par un ϵ. La relation d'inclusion est équivalente à la relation d'égalité lorsque $m = 1$.

Algorithme 5.1 (Calcul de $\mathsf{LookAhead}$**)**
La fonction $\mathsf{LookAhead}$ *d'un état* $[\Delta \rightarrowtail \alpha]$ *est calculée comme suit :*

1. *Ajout des terminaux pouvant prolonger un non-terminal réduit :*
$$\mathsf{LookAhead}([\Delta \rightarrowtail \alpha]) := \{a \in \mathsf{Follow}(A) \mid A \in \mathsf{Reduce}([\Delta \rightarrowtail \alpha])\}$$
2. *Ajout des terminaux attendus pour prolonger l'état :*
$$\mathsf{LookAhead}([\Delta \rightarrowtail \alpha]) := \mathsf{LookAhead}([\Delta \rightarrowtail \alpha]) \cup \{a \in V_T \mid \exists \beta, \gamma, \exists A \in \Delta, A \rightarrow \alpha\beta \ \wedge \ \beta \rightarrow^* a\gamma\}$$
3. *Ajout des terminaux incluant un terminal déjà présent :*
$$\mathsf{LookAhead}([\Delta \rightarrowtail \alpha]) := \mathsf{LookAhead}([\Delta \rightarrowtail \alpha]) \cup \{a \in V_T \mid \exists b \in \mathsf{LookAhead}([\Delta \rightarrowtail \alpha]), b \subset a\}$$

Remarque 5.2 L'étape 3 de l'algorithme précédent concrétise la prise en compte du caractère non-libre du monoïde induit par $\vec{V_T}$ lorsque $m > 1$.

Remarque 5.3 La variété de lookahead définie grâce à la fonction $\mathsf{LookAhead}$ ne vérifie pas les propriétés habituelles du lookahead. En particulier, une règle $A \rightarrow \alpha$ d'un état $[\Delta \rightarrowtail \alpha]$ peut être réduite alors que l'ajout de l'état a été accepté uniquement grâce aux terminaux pouvant suivre α dans une autre règle $B \rightarrow \alpha\beta$. Le lookahead n'est donc pas optimal à cause de la factorisation apportée par l'ensemble Δ des états $[\Delta \rightarrowtail \alpha]$. Il aurait été possible de définir un lookahead amélioré, au prix d'une augmentation considérable de la taille des tables de lookahead. Le compromis actuellement utilisé s'est révélé satisfaisant.

5.2.2 Utilisation des dérivations vides dans les MTSAG

L'algorithme GCP m-bandes présenté p. 80 est limité aux MTSAG dont la grammaire non-contextuelle m-bandes sous-jacente est propre. Une grammaire propre ne contient ni terminal ou production inutile, ni dépendance circulaire, ni dérivation vide.

Il existe un algorithme d'élimination des dépendances circulaires d'une grammaire non-contextuelle normale (Aho & Ullman, 1972), facilement généralisable aux grammaires non-contextuelle m-bandes. Cet algorithme est impossible à appliquer automatiquement à une MTSAG car l'élimination des dépendances circulaires implique l'ouverture d'un cycle $A_0 \to A_2 \to \ldots \to A_k \to A_0$ à un endroit $A_i \to A_{(i+1) \bmod k}$ donné. Or la dérivation supprimée est toujours associée à une fonction d'évaluation d'attributs et il n'est pas possible de supprimer une fonction sans modifier le résultat du calcul des attributs. Les cycles détectés par MTSAG2C dans une MTSAG sont signalés comme étant des erreurs requérant une intervention manuelle de l'utilisateur.

Si l'absence de circularité dans une MTSAG est une contrainte irréductible, la restriction de l'algorithme GCP aux MTSAG ne contenant pas de dérivation vide peut s'avérer inutilement gênante dans certains cas. Habituellement, deux stratégies sont employées pour supprimer cette restriction :

- transformer la grammaire de départ pour supprimer les dérivations vides,
- modifier l'algorithme d'analyse syntaxique pour tenir compte des dérivations vides. Cette stratégie a été popularisée dans (Graham *et al.*, 1980).

La première stratégie est en fait celle qui est naturellement employée manuellement lorsque, partant d'une MTSAG dont la formulation naturelle conduit à l'écriture de dérivations vides, une MTSAG sans dérivation vide est souhaitée. La méthode utilisée pour supprimer les dérivations vides est élémentaire (Aho & Ullman, 1972) :

1. Chercher tous les non-terminaux dérivables en chaîne vide, c'est-à-dire les non-terminaux A tels que $A \to^* \epsilon$;
2. Éliminer les dérivations vides $A \to \epsilon$ de la grammaire ;
3. Soit $B \to \alpha_0 A_1 \alpha_1 \ldots A_k \alpha_k$ une règle de grammaire telle que les non-terminaux A_i $(1 \leqslant i \leqslant k)$ sont les seuls non-terminaux de la règle dérivables en chaîne vide (cf. étape 1 ci-dessus). Alors ajouter à cette règle les $2^k - 1$ règles obtenues en exprimant les différentes combinaisons de suppression ou de conservation des A_i. Par exemple, la règle $B \to \alpha_0 A_1 \alpha_1 A_2 \alpha_2$ entraîne l'ajout des trois règles $B \to \alpha_0 A_1 \alpha_1 \alpha_2$, $B \to \alpha_1 \alpha_1 A_2 \alpha_2$, $B \to \alpha_0 \alpha_1 \alpha_2$ (Cette dernière règle est ajoutée uniquement si $\alpha_0 \alpha_1 \alpha_2 \neq \epsilon$).

Cette méthode ne modifie pas fondamentalement la complexité d'analyse de la MTSAG dans le sens où les nouvelles règles introduites ont un nombre de non-terminaux strictement inférieur à la règle d'origine (cf. théorème 1.6 p. 47). Les fonctions d'évaluation d'attribut des règles vides éliminées lors de la deuxième étape sont prises en compte manuellement au moment de la suppression des non-terminaux pendant la troisième étape.

La deuxième stratégie est délicate à utiliser en l'état avec nos MTSAG car elle doit en fait être écrite en tenant compte des attributs :

> si l'état $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ a été ajouté à $T_{\vec{\jmath}}$ et si cet état est prolongeable en $[\Delta \rightarrowtail \alpha B]$ avec $B \rightarrow^* \epsilon$, alors l'état $[\Delta' \rightarrowtail \alpha B, \vec{\imath}, \lambda x]$ est immédiatement ajouté à $T_{\vec{\jmath}}$, où x est l'attribut de B.

Le problème vient bien entendu du calcul de l'attribut x de B, qui doit être l'attribut optimal de toutes les dérivations de B en ϵ. Pour simplifier le calcul de cet attribut, il est demandé à toute MTSAG contenant un non-terminal B tel que $B \rightarrow^* \epsilon$ de contenir une règle $B \rightarrow \epsilon$, et une fonction d'évaluation d'attributs $f_{B\rightarrow\epsilon}$ retournant l'attribut optimal de B. Ainsi, la grammaire

$$\begin{aligned} B &\rightarrow CD \\ C &\rightarrow \epsilon \mid c \\ D &\rightarrow \epsilon \mid d \end{aligned}$$

vérifie $B \rightarrow^* \epsilon$. Donc il faut lui ajouter la règle $B \rightarrow \epsilon$.

De manière générale, soit $\vec{G}$ une grammaire non-contextuelle m-bandes vérifiant :

$$\forall A \in V_N, A \rightarrow^* \epsilon \Rightarrow A \rightarrow \epsilon$$

Toute grammaire contenant des dérivations vides peut se mettre sous la forme précédente. Les fonctions d'évaluation d'attributs des dérivations vides $A \rightarrow \epsilon$ seront naturellement notées $f_{A\rightarrow\epsilon}$ et sont supposées retourner l'attribut optimal $\lambda = f_{A\rightarrow\epsilon}(\emptyset)$ de A. L'algorithme GCP m-bandes tenant compte des dérivations vides est reproduit en figure 5.1.

5.2.3 Extension possible des grammaires attribuées

Parmi les extensions possibles des grammaires S-attribuées multi-bandes, l'ajout d'un mécanisme d'évaluation d'attributs hérités, et non plus simplement synthétisés, est l'extension qui vient le plus naturellement à l'esprit. Nous serions alors en présence de grammaires attribuées multi-bandes. Théoriquement, il existe une grammaire S-attribuée équivalente à toute grammaire attribuée (Knuth, 1968). Mais cette grammaire S-attribuée équivalente utilise une reformulation fonctionnelle de l'évaluation des attributs hérités qui risque fort d'être inapplicable à nos grammaires ambiguës (du moins si nous souhaitons conserver de bonnes performances d'analyse).

La détermination de l'ordre d'évaluation des attributs dans une grammaire attribuée peut rapidement tourner au cauchemar et il existe d'ailleurs une littérature abondante traitant ce sujet (une bibliographie commentée est disponible dans (Deransart *et al.*, 1988)). Certaines grammaires attribuées possèdent des dépendances circulaires des attributs et le test de circularité d'une grammaire attribuée est un des premiers

Soit Q une file de priorité de triplets $(X, \vec{\imath}, x)$ et sa fonction d'extraction $\mathsf{Extract}$. Au départ, les listes d'analyse m-bandes $T_{\vec{\jmath}}$ sont vides. Soit $\Delta_{\vec{0}} = \{A \in V_N \mid \exists\beta, S \to^* A\beta\}$. Pour $\vec{\jmath}$ allant de $\vec{0}$ (inclus) à $\vec{n}$ (inclus), en ordre croissant d'index m-bandes, exécuter les étapes suivantes :

1. Initialiser la file Q: $Q := \Big\{\big(a, \vec{\imath}, S_{\mathcal{A}}(a)\big) \,\Big|\, \vec{\imath} < \vec{\jmath} \;\wedge\; a = a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}} \in \vec{V}_T\Big\}$

1'. Ajouter les non-terminaux immédiatement réductibles :
$Q := Q \cup \Big\{(A, \vec{\jmath}, x) \,\Big|\, A \to \epsilon, x = f_{A\to\epsilon}(\emptyset)\Big\}$

2. Extraction du triplet à traiter en priorité :
$(X, \vec{\imath}, x) := \mathsf{Extract}(Q)$

3. Création des états attribués commençant par X :
$T_{\vec{\jmath}} := T_{\vec{\jmath}} \cup \Big\{[\Delta \rightarrowtail X, \vec{\imath}, x] \,\Big|\, [\Delta \rightarrowtail X] = \mathsf{Initial}(\Delta_{\vec{\imath}}, X)\Big\}$

4. Prolongement d'états attribués par X :
$T_{\vec{\jmath}} := T_{\vec{\jmath}} \cup \Big\{[\Delta \rightarrowtail \alpha X, \vec{h}, \lambda x] \,\Big|\, \exists[\Delta' \rightarrowtail \alpha, \vec{h}, \lambda] \in T_{\vec{\imath}}, [\Delta \rightarrowtail \alpha X] = \mathsf{Goto}\big([\Delta' \rightarrowtail \alpha], X\big)\Big\}$

5. Pour chaque état attribué $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ ajouté à $T_{\vec{\jmath}}$ par une des étapes 3 et 4 ci-dessus, et pour chaque $A \in \mathsf{Reduce}\big([\Delta \rightarrowtail \alpha]\big)$, calculer le triplet $(A, \vec{\imath}, x) := \big(A, \vec{\imath}, f_{A\to\alpha}(\lambda)\big)$.
   Si $\mathcal{C}^{\mathcal{S}}_A(\vec{\imath}, \vec{\jmath}, x)$ est faux alors passer directement à l'étape 6.
   S'il existe déjà un triplet $(A, \vec{\imath}, x')$ appartenant à Q, remplacer l'attribut x' de ce triplet par $\mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x, x')$.
   Si aucun triplet de ce type n'existe dans Q, ajouter $(A, \vec{\imath}, x)$ à Q.

5'. Pour chaque état attribué $[\Delta \rightarrowtail \alpha, \vec{\imath}, \lambda]$ ajouté à $T_{\vec{\jmath}}$ par une des étapes 3 et 4 ci-dessus, et pour chaque A tel que $A \to \epsilon$ et $\mathsf{Goto}\big([\Delta \rightarrowtail \alpha], A\big) \neq \emptyset$, ajouter le triplet $\big(A, \vec{\jmath}, f_{A\to\epsilon}(\emptyset)\big)$ à Q s'il n'y est pas déjà présent.

6. Mettre à jour $\Delta_{\vec{\jmath}} := \mathsf{Expect}(T_{\vec{\jmath}})$.

7. Répéter les étapes 2 à 7 tant que Q n'est pas vide.

Le mot m-bandes $a_{\vec{1}} \dots a_{\vec{n}}$ appartient à $L(\vec{G})$ si et seulement si le triplet $(S, \vec{0}, x)$ a été extrait de Q pendant le calcul de $T_{\vec{n}}$. On dit alors que le mot a été reconnu par l'algorithme et qu'il a un attribut optimal x.

Figure 5.1: Algorithme GCP m-bandes tenant compte des dérivations vides. En pratique, pour éviter les tentatives d'ajout inutiles, les triplets $(A, \vec{\jmath}, \lambda)$ sont ajoutés à Q dès que nécessaire. Notons que $\vec{\jmath}$ peut être égal à $\vec{0}$.

exemples prouvés de problème pratique ayant une complexité intrinsèquement exponentielle (Jazayeri *et al.*, 1975). Pour cette raison, il est hors de question d'ajouter des attributs hérités dans toute leur généralité à nos MTSAG : la complexité d'évaluation simultanée d'attributs hérités et synthétisés génériques n'est pas compatible avec la complexité d'une recherche d'arbre de dérivation optimal pour une grammaire à ambiguïté exponentielle. D'ailleurs, la totalité des méthodes sophistiquées d'évaluation d'attributs exposées dans la littérature supposent explicitement que l'arbre de dérivation à décorer d'attributs est donné, ou bien que la grammaire non-contextuelle sous-jacente à la grammaire attribuée est non-ambiguë. Pour être utilisables avec un algorithme d'analyse tabulaire, les attributs hérités doivent être calculables à partir d'arbres de dérivation partiellement construits. Nous devons donc restreindre la classe des grammaires attribuées manipulées et, dans ce contexte, les grammaires L-attribuées apparaissent comme des candidates idéales. La définition suivante est une adaptation à nos notations de la notion de grammaire L-attribuée telle qu'elle est définie dans (Lewis *et al.*, 1974).

Définition 5.3 (Grammaire L-attribuées m-bandes)
Une grammaire L-attribuée m-bandes notée $\vec{G} = (\vec{V}_T, V_N, P, S, \mathcal{A}, S_{\mathcal{A}}, F_P, I_P)$ est une extension de la grammaire non-contextuelle m-bande $\vec{G} = (\vec{V}_T, V_N, P, S)$. Un attribut synthétisé x appartenant à un ensemble d'attributs $\mathcal{A}$ est adjoint à chaque $a \in \vec{V}_T$ par la fonction $S_{\mathcal{A}}$:

$$x = S_{\mathcal{A}}(a)$$

Un couple (x, x') d'attributs de $\mathcal{A}$ est adjoint à chaque non-terminal de V_N. Le premier attribut x de ce couple est l'attribut synthétisé associé au non-terminal. Le deuxième attribut x' est l'attribut hérité.

Soit $X_0 \to X_1 \dots X_p$ une production de $\vec{G}$.

Comme avec les grammaires S-attribuées m-bandes, l'attribut synthétisé x_0 de X_0 est calculé à partir des attributs synthétisés $x_1, \dots, x_p$ des symboles $X_1, \dots, X_P$ par une fonction $F_{X_0 \to X_1 \dots X_p}$ de F_P :

$$x_0 = F_{X_0 \to X_1 \dots X_p}(x_1 \dots x_p)$$

Soit $i \geqslant 1$. Si X_i est un non-terminal, l'attribut hérité x'_i de X_i est calculé à partir de l'attribut hérité x'_0 de X_0 et des attributs synthétisés x_k des X_k $(1 \leqslant k \leqslant i-1)$ par une fonction $I_{X_0 \to X_1 \dots X_{i-1} \bullet X_i \dots X_p}$ de I_P :

$$x'_i = I_{X_0 \to X_1 \dots X_{i-1} \bullet X_i \dots X_p}(x'_0 x_1 \dots x_{i-1}) \tag{5.1}$$

L'attribut hérité d'un non-terminal à droite d'une production de $\vec{G}$ ne dépend donc que des attributs synthétisés des symboles qui le précèdent et de l'attribut hérité de son non-terminal parent dans un arbre de dérivation. Les attributs de tous les symboles étiquetant les nœuds d'un arbre de dérivation peuvent donc être évalués grâce à un parcours de haut en bas, et de la gauche vers la droite, de l'arbre de dérivation. Les attributs hérités sont calculés lors de la descente dans l'arbre, tandis que les attributs synthétisés sont calculés lors de la remontée.

L'évaluation des attributs synthétisés et hérités d'un arbre de dérivation conjointement avec une analyse syntaxique de type bottom-up a déjà été étudiée dans le cadre des grammaires L-attribuées et de l'analyse LR des grammaires $LR(1)$ (op den Akker *et al.*, 1990). Compte tenu des résultats existants, la généralisation de l'algorithme GCP S-attribué m-bandes aux grammaires L-attribuées m-bandes ne semble pas rencontrer d'obstacle fondamental à condition d'imposer certaines restrictions aux fonctions d'évaluation d'attributs hérités. En particulier, et compte tenu de la forme des états d'analyse utilisés par l'algorithme GCP, toutes les fonctions $I_{X_0 \to X_1 \dots X_{i-1} \bullet X_i \dots X_p}$ doivent être identiques à $X_1 \dots X_{i-1}$ fixé afin que la notion de fonction $I_{[\Delta \to \alpha]}$ associée à un état d'analyse puisse être définie.

Cependant, avant d'intégrer dans MTSAG2C les nombreuses modifications nécessaires à la manipulation d'une forme restreinte de grammaires L-attribuées, nous devons nous assurer qu'il existe des applications pouvant bénéficier de l'ajout d'attributs hérités.

Une telle application pourrait être l'introduction de contraintes de sélection top-down sur les attributs-hérités, et non plus simplement bottom-up sur les attributs synthétisés comme actuellement. Les contraintes de sélection bottom-up éliminent une forêt de sous-arbres de dérivation après la construction de cette forêt. Cet élagage bottom-up empêche la prolongation de certaines règles de dérivation et permet donc de gagner ultérieurement du temps et de la mémoire. L'élagage provoqué par des contraintes de sélection top-down est sensiblement différent puisque la forêt de sous-arbres n'est même pas construite. Les contraintes top-down sont donc adaptées à l'introduction de contraintes telles que les contraintes d'alignement ou les pré-contraintes d'optimisation.

Ainsi, dans le cadre d'un modèle probabiliste d'analyse, il serait possible d'affecter à chaque non-terminal :

- un attribut synthétisé `max_proba` représentant la probabilité maximale de dérivation d'une sous-chaine $a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$ de la chaîne analysée,
- un attribut hérité `min_proba` représentant la probabilité minimale nécessaire de dérivation d'une sous-chaine $a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$.

Les fonctions de F_P resteraient inchangées par rapport à une grammaire S-attribuée classique, tandis que les fonctions $I_{X_0 \to X_1 \dots X_{i-1} \bullet X_i \dots X_p}(x'_0 x_1 \dots x_{i-1})$ de I_P seraient de la forme $x'_0/(x_1 \times \cdots \times x_{i-1})$. La contrainte d'optimisation sur les non-terminaux serait alors la contrainte usuelle de maximisation, tandis que la contrainte de sélection top-down serait le rejet de toute dérivation d'un non-terminal lorsque `min_proba` est supérieur à 1. De cette manière, lorsque la probabilité maximale d'un non-terminal B dans une dérivation $A \to BC$ est trop petite par rapport à la probabilité minimale nécessaire de A, la probabilité minimale de C devient supérieure à 1, ce qui est bien sûr impossible et justifie l'abandon de toute recherche de dérivation de C.

5.3 Écriture d'une MTSAG

Pour une MTSAG ordinaire $\vec{G}$, la complexité d'analyse est fonction du nombre maximal r de non-terminaux dans les parties droites α des règles $A \to \alpha$ de $\vec{G}$ (cf. théorèmes 1.6 et 2.2). En général, la complexité en temps est $O\big(n^{m(r+1)}\big)$ et la complexité en espace est $O\big(n^{mr}\big)$. Il est donc important de conserver r inférieur ou égal à 2, le cas $r = 1$ correspondant à une grammaire linéaire. En théorie, toute grammaire non-contextuelle vérifiant $r > 2$ peut être transformée en une grammaire équivalente (i.e. générant le même langage) sous forme normale de Chomsky, c'est-à-dire une grammaire vérifiant $r \leqslant 2$. Mais le processus de normalisation n'est pas toujours transposable aux MTSAG pour trois raisons :

- la structure définie par une grammaire est plus importante que le langage généré par cette grammaire et l'équivalence des langages générés par la normalisation n'est pas une propriété assez forte,

- il faut pouvoir transposer les fonctions d'évaluation d'attributs d'une grammaire à l'autre,

- les attributs des non-terminaux introduits au cours de la normalisation doivent pouvoir être optimisés.

Il se pourrait fort bien qu'une des trois raisons précédentes rende impossible la normalisation d'une MTSAG vérifiant $r > 2$. En pratique, nous n'avons pas trouvé de modèle d'analyse de séquence réaliste pour lequel il serait nécessaire d'écrire une MTSAG vérifiant $r > 2$ (ce qui ne veut d'ailleurs pas dire qu'un tel modèle n'existe pas). Ainsi, toutes les applications présentées dans cette thèse utilisent des MTSAG telles que $r \leqslant 2$.

Un autre paramètre influençant fortement les performances d'un analyseur généré à partir d'une MTSAG est la récursivité à droite des règles de grammaire. Si la grammaire de la figure 3.1 p. 88 avait été écrite sous la forme récursive à droite équivalente de la figure 5.2, alors les complexités d'alignement de deux séquences de tailles m et n respectivement auraient été de $O(m^2n^2)$ en temps et en espace au lieu de $O(mn)$. En règle générale, il faut toujours préférer une écriture récursive à gauche plutôt qu'une écriture récursive à droite pour un même ensemble structural. Il existe cependant des MTSAG pour lesquelles il est impossible, ou extrêmement malcommode, de transposer les fonctions d'évaluation d'attributs écrites pour des règles naturellement récursives à droite en fonctions d'évaluation d'attributs écrites pour des règles artificiellement récursives à gauche (cf section 3.10 p. 156).

Les remarques précédentes montrent que, même si les MTSAG sont plus agréables à employer que des équations de programmation dynamique lorsqu'il s'agit de décrire formellement un modèle d'analyse de séquences, elles ne sont pas dépourvues de complexité. En plus des problèmes de performance évoqués, les fonctions d'évaluation d'attributs peuvent contenir des vermines (bugs). Le générateur MTSAG2C facilite cependant le travail de vérification des fonctions en affichant un certain nombre de

```
Align : [• •]          { $$.c = s($1[1].left[0].b, $1[2].left[0].b); }
      | [• _]          { $$.c = -deletion; }
      | [_ •]          { $$.c = -insertion; }
      | [• •] Align    { $$.c = $2.c + s($1[1].left[0].b, $1[2].left[0].b); }
      | [• _] Align    { $$.c = $2.c - deletion; }
      | [_ •] Align    { $$.c = $2.c - insertion; }
      ;
%constraint Align {} { if ($$.c > $!.c) $!.c = $$.c }
```

Figure 5.2: Grammaire récursive à droite équivalente à la grammaire récursive à gauche de la figure 3.1.

messages d'avertissement ou d'erreur lorsqu'il estime que la MTSAG proposée est défectueuse (cf. section A.5 p. 254). Il est également possible d'écrire une MTSAG traduisant incorrectement le modèle de départ. Un tel problème est d'autant plus difficile à détecter que des MTSAG "visiblement correctes" ont souvent tendance à retourner des résultats cohérents dont la fausseté n'est pas immédiatement apparente. Mais ce problème est encore plus difficile à détecter avec les équations de programmation dynamique car ces dernières masquent admirablement la structure du modèle de départ ; Il est toujours malaisé de savoir si les erreurs constatées dans les résultats obtenus proviennent d'une mauvaise transcription des équations (i.e. d'une erreur dans le programme) ou des équations elles-mêmes.

5.4 Classe de langages générés par les MTCFG

Dans cette section, nous nous désintéresserons des MTSAG pour porter toute notre attention sur leur armature : les grammaires non-contextuelles m-bandes ou MTCFG. Nous remarquerons tout d'abord que ces grammaires ont déjà été étudiées sous la forme d'automates à pile non-déterministes à m-bandes (ou m-tape non deterministic pushdown automata, ou m-TPDA). Nous établirons ensuite le rapport entre les m-TPDA et les automates à pile non-déterministes à m-têtes (ou m-head non deterministic pushdown automata, ou m-HPDA). Nous énoncerons ensuite quelques résultats concernant la classe des langages générés par les m-TPDA et les m-HPDA. La totalité des résultats de cette section proviennent de (Ibarra, 1967; Harrison & Ibarra, 1968; Chrobak, 1985; Chrobak & Li, 1988). Le lecteur est invité à se reporter à ces documents pour obtenir des résultats plus détaillés que dans la présente section, ainsi que les démonstrations des théorèmes énoncés.

La prise de conscience du lien existant entre problèmes d'analyse de séquence et automates non-déterministes n'est pas nouvelle :

- motifs de PROSITE (base de motifs protéiques) ou HMM $\Rightarrow$ automates finis non-déterministes (ou non deterministic finite automata, ou NFA) ;
- structures secondaires d'ARN et grammaires non-contextuelles $\Rightarrow$ automates à pile non-déterministes (ou PDA) ;

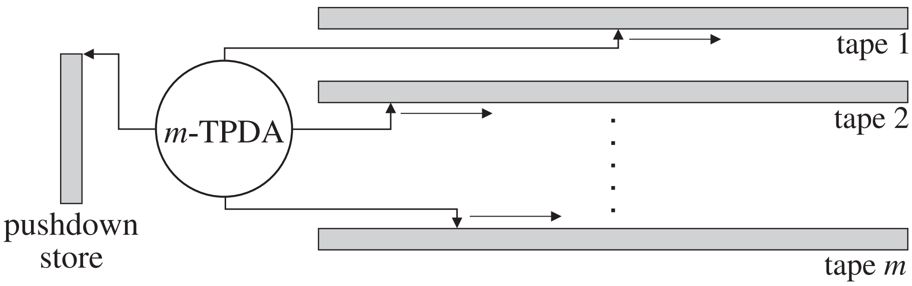


Figure 5.3: Représentation simplifiée d'un m-TPDA.

- alignement de deux séquences suivant diverses distances $\Rightarrow$ automates finis non-déterministes à m-bandes (ou m-tape non-déterministic finite automata, ou m-TNFA).

Cette thèse montre que tous les problèmes précédents, ainsi que l'alignement de deux séquences suivant une grammaire non-contextuelle, sont des instances du formalisme des MTCFG, c'est-à-dire des m-TPDA (cf. théorème 5.1) :

- un NFA est un 1-NFA, ou un 1-TPDA dont la pile est inutilisée,
- un PDA est un 1-TPDA,
- un m-TNFA est un m-TPDA dont la pile est inutilisée.

Intuitivement, un m-TPDA est une généralisation de PDA utilisant m-bandes indépendantes, chacune lue de gauche à droite par une tête de lecture, au lieu d'une unique bande lue de gauche à droite. Comme tout PDA, un m-TPDA possède un nombre fini d'états internes et une unique bande de stockage externe structurée en pile LIFO (figure 5.3). Un m-TPDA passe d'un état à l'autre, empile, ou dépile, en fonction de l'état courant, de l'information stockée au sommet de la pile, et des tokens lus par un sous-ensemble des têtes de lecture. Un mot m-bandes est reconnu par un m-TPDA lorsque l'automate est dans un état dit *final* alors que toutes les têtes de lecture se trouvent en fin de bande. L'automate est non-déterministe car plusieurs états peuvent être destination d'un même état pour des transitions identiques et car les têtes de lecture ne progressent pas nécessairement simultanément sur les bandes (i.e. le monoïde $\langle\vec{\Sigma}^*\rangle$ n'est pas libre (cf. section 2.1 p. 72 pour la notation $\langle\vec{\Sigma}^*\rangle$)). L'ensemble des mots m-bandes reconnus par un m-TPDA est le langage défini par le m-TPDA.

Un m-HPDA est un m-TPDA dont toutes les bandes sont contraintes à être identiques ou, de manière équivalente, une généralisation de PDA utilisant une seule bande et m têtes de lecture indépendantes sur cette bande.

Définition 5.4
Un sous-ensemble $\vec{L}$ de $\langle\vec{\Sigma}^*\rangle$ est un langage non-contextuel m-bandes s'il est défini par une grammaire non-contextuelle m-bandes $\vec{G}$, c'est-à-dire si $\vec{L} = L(\vec{G})$.

Théorème 5.1
Un sous-ensemble $\vec{L}$ de $\langle\vec{\Sigma}^*\rangle$ est un langage non-contextuel m-bandes si et seulement si il est défini par un m-TPDA.

Démonstration : Il s'agit essentiellement d'une généralisation de la démonstration classique d'équivalence des PDA et des CFG (cf. théorème 2.1 dans (Harrison & Ibarra, 1968)). □

Le résultat "intuitif" suivant est également vrai.

Théorème 5.2
Soit $\vec{L}$ défini par un m-TPDA. Soit $\vec{L}^{(k)} = \{\langle\vec{u}\rangle^{(k)} \mid \vec{u} \in \vec{L}\}$. Alors $\vec{L}^{(k)}$ est un langage non-contextuel "traditionnel".

Démonstration : cf. (Ibarra, 1967) et théorème 2.3 dans (Harrison & Ibarra, 1968). □

Il n'est pas difficile de constater que si $\vec{L} = L(\vec{G})$, alors $\vec{L}^{(k)} = L(\vec{G}^{(k)})$. Le théorème suivant montre que la réciproque du théorème 5.2 est fausse.

Théorème 5.3
Quel que soit $m \geqslant 2$, il existe $\vec{L} \in \langle\vec{\Sigma}^*\rangle$ non définissable par un m-TPDA tel que pour tout $k \in \{1, \ldots m\}$, $\vec{L}^{(k)}$ est un langage régulier.

Démonstration : cf. théorème 2.5 dans (Harrison & Ibarra, 1968). La démonstration s'appuie sur le langage $\vec{L} = \big(L' \times \{1\}\big) \cup \big((\Sigma^* - L') \times \{0\}\big)$, où L' est un ensemble récursivement énumérable non récursif. □

Nous disposons en revanche du théorème suivant.

Théorème 5.4
Soit $\vec{L} \in \langle\vec{\Sigma}^*\rangle$ tel que $\vec{L} = L_1 \times \cdots \times L_m$, où $L_k \in \Sigma_k^*$ ($1 \leqslant k \leqslant m$). Alors $\vec{L}$ est défini par un m-TPDA si et seulement si pour tout $k \in \{1, \ldots, m\}$, L_k est défini par un m-TPDA.

Démonstration : cf. théorème 5.8 dans (Harrison & Ibarra, 1968). □

Les langages définis par les m-TPDA ne sont pas directement comparables aux langages usuels de la hiérarchie de Chomsky du fait de la présence de plusieurs bandes. Cependant, et lorsque $\Sigma_1 = \cdots = \Sigma_m$, la projection sur une bande de l'intersection de tout langage défini par un m-TPDA avec le langage des mots m-bandes dont les projections sont identiques sur toutes les bandes[5] est définissable par un m-HPDA. Le théorème suivant n'est donc pas sans intérêt pour les m-TPDA.

[5] C'est-à-dire $L = \{\underbrace{(u, \ldots, u)}_{k \text{ fois}} \mid u \in {\Sigma_1}^*\}$.

Théorème 5.5
Les langages reconnus par les m-HPDA forment une hiérarchie propre comprise entre les langages de type 2 (context-free), pour $m = 1$, et les langages de type 1 (context-sensitive).

Les langages reconnus par les m-HPDA déterministes (ou m-HDPDA) forment une hiérarchie propre, mais il existe un langage context-free non reconnu par un m-HDPDA quel que soit m.

Démonstration : cf. (Chrobak, 1985) et (Chrobak & Li, 1988). □

Le théorème précédent signifie que tous les langages reconnus par un m-HPDA sont reconnus par un $(m+1)$-HPDA, mais que l'inverse n'est pas vrai. De plus, tous les langages reconnus par les m-HPDA sont contenus dans la classe des langages de type 1, l'inverse étant bien évidemment faux.

La concaténation des m bandes lues par un m-TPDA fournit une unique bande lue par m-têtes différentes. Il n'est pas difficile, étant donnée la définition d'un m-TPDA, de construire un m-HPDA lisant la bande concaténée et la reconnaissant si et seulement si le m-TPDA reconnaît le mot m-bandes d'origine. Le théorème précédent signifie donc que, d'une certaine manière, les m-TPDA forment une hiérarchie propre et ne sont pas plus puissants que les automates linéairement bornés. Nous avons déjà vu un exemple de MTSAG non-contextuelle dans la section 3.13, où le langage non context-free $\{a^i b^j c^i d^j \mid i > 0, j > 0\}$ était reconnu par une MTSAG, ce résultat étant utilisé pour faciliter la détermination de pseudo-nœuds dans les ARN.

Théorème 5.6 (Propriétés de clôture des m-TPDA)
Soit $\vec{R} \in \langle \vec{\Sigma}^* \rangle$ tel que $\vec{R} = R_1 \times \cdots \times R_m$, où les ensembles R_k sont réguliers ($1 \leqslant k \leqslant m$). Soit $\vec{L}$ un langage défini par un m-TPDA. Alors $\vec{L} \cap \vec{R}$ et $\vec{L} \cup \vec{R}$ sont tous deux définis par un m-TPDA.

Par ailleurs, l'ensemble des m-TPDA est clos par :

- *union,*
- *concaténation,*
- *clôture,*
- *transposition.*

Il n'est pas clos par :

- *intersection,*
- *complémentation.*

Démonstration : cf. (Ibarra, 1967) et théorèmes 5.1, 5.2 et 5.6 dans (Harrison & Ibarra, 1968). □

Théorème 5.7 (Propriétés de clôture des m-HPDA)
Soient R un ensemble régulier et L un langage défini par un m-HPDA. Alors $L \cap R$ et $L \cup R$ sont tous deux définis par un m-HPDA.

Par ailleurs, pour chaque m, l'ensemble des m-HPDA est clos par :

- *union,*
- *concaténation,*
- *clôture,*
- *transposition,*
- *transformation par machine séquentielle généralisée inverse (inverse gsm mapping).*

Il n'est pas clos par :

- *intersection,*
- *complémentation,*
- *homomorphisme si $m \geqslant 2$,*
- *transformation par machine séquentielle généralisée (gsm mapping) si $m \geqslant 2$.*

Démonstration : cf. (Ibarra, 1967), théorèmes 5.11, 5.15 et 5.16 dans (Harrison & Ibarra, 1968), et corollaire 2 dans (Chrobak & Li, 1988).

Rappelons (cf. (Salomaa, 1973)) que les langages non-contextuels sont clos par union, concaténation, clôture, transposition, homomorphisme, gsm mapping et inverse gsm mapping. Par contre, ils ne sont pas clos par intersection ou complémentation.

Théorème 5.8 (Propriétés de décidabilité pour les m-TPDA)
Soient $\vec{L}$, $\vec{L}_1$ et $\vec{L}_2$ des langages définis par des m-TPDA (ou, de manière équivalente, par des grammaires non-contextuelles m-bandes).

Problème	*Classe*
$x \in \vec{L}$?	*Décidable*
$\vec{L}_1 \subset \vec{L}_2$?	*Indécidable*
$\vec{L}_1 = \vec{L}_2$?	*Indécidable*
$\vec{L} = \emptyset$?	*Décidable*
$\vec{L}$ *infini ?*	*Décidable*
$\vec{L}$ *régulier ?*	*Indécidable*
$\vec{L}$ *non-ambigu ?*	*Indécidable*
$\sim \vec{L} = \emptyset$?	*Indécidable*
$\sim \vec{L}$ *infini ?*	*Indécidable*
$\sim \vec{L}$ *régulier ?*	*Indécidable*
$\sim \vec{L}$ *défini par un m-TPDA ?*	*Indécidable*
$\vec{L}_1 \cap \vec{L}_2$ *vide ?*	*Indécidable*
$\vec{L}_1 \cap \vec{L}_2$ *infini ?*	*Indécidable*
$\vec{L}_1 \cap \vec{L}_2$ *régulier ?*	*Indécidable*
$\vec{L}_1 \cap \vec{L}_2$ *défini par un m-TPDA ?*	*Indécidable*

Démonstration : Ces résultats généralisent les résultats connus pour $m = 1$. Seules les extensions des propriétés de décidabilité dans le cas $m = 1$ méritent en fait attention et sont démontrées dans (Ibarra, 1967) (cf. théorème 6.3 dans (Harrison & Ibarra, 1968)). □

Théorème 5.9 (Propriétés de décidabilité pour les m-HPDA)
Soient L, L_1 et L_2 des langages définis par des m-HPDA.

Problème	*Classe*
$x \in L$?	*Décidable*
$L_1 \subset L_2$?	*Indécidable*
$L_1 = L_2$?	*Indécidable*
$L = \emptyset$?	*Indécidable si $m \geqslant 2$*
L infini ?	*Indécidable si $m \geqslant 2$*
L régulier ?	*Indécidable*
L non-ambigu ?	*Indécidable*
$\sim L = \emptyset$?	*Indécidable*
$\sim L$ infini ?	*Indécidable*
$\sim L$ régulier ?	*Indécidable*
$\sim L$ défini par un m-HPDA ?	*Indécidable*
$L_1 \cap L_2$ vide ?	*Indécidable*
$L_1 \cap L_2$ infini ?	*Indécidable*
$L_1 \cap L_2$ régulier ?	*Indécidable*
$L_1 \cap L_2$ défini par un m-HPDA ?	*Indécidable*

Démonstration : L'indécidabilité de $L = \emptyset$ et L infini pour $m \geqslant 2$ est démontrée dans (Ibarra, 1967) (cf. théorème 6.7 dans (Harrison & Ibarra, 1968)). Les autres résultats généralisent les résultats connus pour $m = 1$.

Conclusion

Nous avons établi dans cette thèse que les grammaires S-attribuées multi-bandes, ou MTSAG, offrent un formalisme souple et élégant permettant de décrire de nombreux modèles d'analyse de séquence. La plupart de ces modèles étaient jusqu'à présent exposés sous la forme d'équations de programmation dynamique pas toujours faciles à écrire et à comprendre. D'autre part, nous avons introduit un algorithme d'analyse syntaxique adapté aux grammaires utilisées : l'algorithme GCP. Son implémentation dans le générateur automatique d'analyseurs syntaxiques MTSAG2C nous a permis de montrer que les MTSAG ne conduisent pas seulement à une formulation élégante des modèles mais également à une formulation efficace. L'obsolescence des équations de programmation dynamique est pour le moment évitée dans le cas des modèles élémentaires (alignement par distance d'édition, HMM). Dans tous les autres cas, les performances des analyseurs syntaxiques générés rendent caduques les implémentations manuelles des équations de programmation dynamique. Dans deux cas, la complexité asymptotique des analyseurs syntaxiques générés à partir d'une MTSAG s'est révélée meilleure que la complexité asymptotique originellement proposée. Soulignons que les MTSAG ne remettent pas en cause la pertinence des modèles, mais seulement leur implémentation sous la forme d'équations peu claires et parfois mal maîtrisées.

Fondamentalement, les MTSAG ne sont pas plus puissantes que les équations de programmation dynamique (l'algorithme GCP utilise après tout une certaine forme de programmation dynamique) ; elles sont simplement plus maniables. Grâce aux MTSAG, les structures des modèles restent sans cesse apparentes dans les règles des grammaires et leur évolution n'entraîne pas de modifications longues et incertaines des équations et des codes C. Cela fait des MTSAG et de MTSAG2C un instrument précieux pour prototyper de nouveaux modèles d'alignement et/ou de calcul de structures.

Notons cependant que les MTSAG ne sont pas une panacée pour tous les problèmes d'analyse de séquences. Des outils tels que BLAST (Altschul *et al.*, 1990) font appel à des heuristiques difficiles à inclure dans un formalisme plus général. De même, la recherche de motifs réguliers dans des séquences ou des bases de données est beaucoup mieux traitée à l'aide d'autres méthodes à base d'arbres suffixes ou de variations des algorithme de Boyer–Moore (Boyer & Moore, 1977) ou Knuth–Morris–Pratt (Knuth *et al.*, 1977). Dans le domaine de l'alignement de séquences par distance d'édition, l'emploi généralisé des techniques en espace linéaire introduites par Hirschberg (Hirschberg, 1975; Chao & Miller, 1995) permet le traitement de séquences de plusieurs centaines de milliers de bases, ce qui est pour le moment irréalisable avec notre implémentation des MTSAG. Mais il est raisonnable d'imaginer une modification de MTSAG2C détectant automatiquement les grammaires auxquelles les techniques de Hirschberg sont applicables et optimisant l'analyseur syntaxique généré en conséquence (Birney, 1996).

Perspectives

Même si les MTSAG ne sont pas exploitables avec n'importe quel problème d'analyse de séquence et sont d'un abord un peu délicat au départ, nous espérons que, dans les domaines où elles peuvent se révéler utiles (alignement de séquences, et particulièrement alignement d'ARN), elles remplaceront les sempiternelles équations de programmation dynamique et permettront la création de nouveaux modèles performants. Nous avons d'ailleurs proposé dans cette thèse une esquisse de modèle adapté à la recherche de structures secondaires consensus de familles d'ARN de plusieurs milliers de bases (section 3.12). Il est indispensable d'implémenter cette esquisse et de la tester sur des familles existantes et référencées (Nucleic Acids Research vol 24 number 1, 1996). L'utilisation des MTSAG pour la recherche de pseudo-nœuds dans les ARN doit également être approfondie car les exemples proposés en section 3.13 ne sont qu'une ébauche peu satisfaisante. Il faudrait en particulier vérifier que les MTSAG sont capables de retrouver des pseudo-nœuds répertoriés dans de grands ARN (van de Peer *et al.*, 1996). À plus long terme, la construction d'un algorithme de calcul d'une structure consensus incluant les pseudo-nœuds est envisageable.

À l'heure actuelle, les analyseurs générés par MTSAG2C n'offrent pas de mécanisme automatique efficace de recherche sur une deuxième bande de motifs définis par une première bande structurée (cf section 3.9 et 3.10). Une telle extension serait souhaitable pour faciliter la description de modèles de recherche de séquences dans des bases de données. Notons cependant que Leslie Grate a utilisé un analyseur produit par MTSAG2C à partir d'une MTSAG 2-bandes pour trouver un ARN de transfert non déjà annoté dans le génome de *Methanococcus jannaschii* et qui, apparemment, n'avait pas été découvert par d'autres méthodes de recherche d'ARN (communication personnelle, Novembre 1996).

Sur un plan purement informatique, l'algorithme GCP et son implémentation sont sans doute encore améliorables, surtout dans le cadre d'une application multi-bandes. Toutes les MTSAG présentées dans cette thèse utilisaient au plus deux bandes. Certaines de ces MTSAG sont généralisables à trois bandes ou plus. Il n'est pas raisonnable d'envisager de dépasser trois bandes étant données les complexités d'analyse syntaxique offertes par l'algorithme GCP. Pour dépasser trois bandes, il faudrait revoir complètement le processus d'analyse multi-bandes. Il serait par exemple concevable d'effectuer l'analyse syntaxique bande par bande à l'aide des grammaires projetées de la MTSAG de départ. Une "intersection" des forêts d'arbres de dérivation obtenues pourrait alors fournir la forêt associée aux dérivations de la MTSAG. Cette approche promet d'être délicate car toute forêt d'arbres de dérivation mono-bande peut être réprésentée à l'aide d'une grammaire non-contextuelle (Billot & Lang, 1989; Lang, 1992) et l'ensemble des grammaires non-contextuelles n'est pas clos par intersection.

Citons enfin les travaux de Nykänen *et al.* (Grahne *et al.*, 1994). Ils définissent un *Alignment calculus* dont ils montrent l'équivalence aux automates finis non-déterministes à plusieurs bandes dont les têtes de lecture peuvent se déplacer aussi bien vers la gauche que vers la droite (non-deterministic 2-way m-tapes finite state automata). Ces automates sont à leur tour équivalents aux machines de Turing. L'alignment calculus est donc théoriquement plus puissant que les MTSAG, même si toutes les applications proposées dans (Grahne *et al.*, 1994) restent solubles à l'aide des MTSAG. Pour le moment, aucun algorithme effectif d'évaluation d'expressions utilisant l'alignment calculus n'a été décrit, mais Nykänen *et al.* y travaillent, bien que l'obtention d'algorithmes performants ne soit pas leur objectif prioritaire (communication personnelle de Nykänen en Juin 1996).

Annexe A

Manuel de référence de MTSAG2C

A.1 Introduction

Afin de pouvoir aisément mettre en œuvre l'algorithme GCP multi-bandes sur une MTSAG donnée sans avoir à calculer manuellement les tables regroupant les états d'analyse $[\Delta \rightarrowtail \alpha]$ et les ensembles de non-terminaux attendus, nous avons créé un générateur automatique d'analyseurs syntaxiques appelé MTSAG2C. Cet outil lit une MTSAG décrite dans un fichier texte et génère le source C d'un analyseur syntaxique adapté à la grammaire lue. La syntaxe de description des MTSAG acceptées par MTSAG2C doit être :

- générale, pour permettre la description de n'importe quelle MTSAG et de n'importe quelles contraintes de sélection et d'optimisation,
- souple, pour ne pas imposer de contraintes syntaxiques inutiles (nombre d'espaces pour séparer les symboles, nombre de lignes, position des symboles, etc...) à la personne chargée d'écrire la grammaire,
- claire et naturelle, pour être facilement assimilable,
- concise, dans le contexte des problèmes d'analyse de séquences, pour éviter les répétitions inutiles de règles semblables.

C'est pourquoi nous avons choisi d'adapter aux MTSAG la syntaxe utilisée par YACC, plutôt que de créer une syntaxe entièrement nouvelle, pas nécessairement plus judicieuse que celle de YACC.

Un des avantages de la syntaxe YACC est d'utiliser le langage C comme moyen d'exprimer les règles sémantiques (ou fonctions d'évaluation d'attributs dans notre cas). Le langage C est suffisamment puissant pour permettre l'expression de n'importe quel calcul. Il dispose de plus de compilateurs performants sur la plupart des architectures existantes. La conséquence immédiate de ce choix est que l'analyseur syntaxique généré par MTSAG2C est un fichier source C contenant d'une part un noyau de fonctions d'analyse communes à tous les analyseurs générés par MTSAG2C et des tables spécifiques à chaque MTSAG, et d'autre part les différents codes inclus dans chaque MTSAG, et en particulier les codes des fonctions d'évaluation d'attributs et les codes des contraintes.

L'analyseur syntaxique généré par MTSAG2C à partir d'une MTSAG ne contient pas de description lexicale des tokens, mais simplement des constantes entières référencées dans l'analyseur comme étant les codes des tokens. L'utilisateur de MTSAG2C doit donc écrire un analyseur lexical en plus de la MTSAG. Fort heureusement, l'interface entre l'analyseur syntaxique généré par MTSAG2C et l'analyseur lexical à écrire a été conçue pour LEX, l'outil de génération automatique d'analyseurs lexicaux standard sous UNIX. L'utilisateur n'a donc qu'à écrire un fichier de description de la MTSAG et un fichier de description des expressions régulières associées aux tokens, puis à laisser le soin à deux programmes (MTSAG2C et LEX) de générer automatiquement les fichiers C nécessaires.

Le manuel de référence de MTSAG2C s'inspire de celui de BISON (Donnelly & Stallman, 1990), un proche parent de YACC. Après une brève introduction, nous donnerons deux exemples d'utilisation de MTSAG2C, puis nous détaillerons la syntaxe des grammaires lues par MTSAG2C. Finalement, les variables, fonctions et macros réalisant l'interface entre le langage C et MTSAG2C seront exposées.

A.2 Exemple de MTSAG mono-bande

A.2.1 Le fichier MTSAG2C

La grammaire `arith.cf` de la figure A.1 permet d'obtenir l'arbre de plus grande valeur d'une expression arithmétique, et d'afficher le parenthésage correspondant à cet arbre. La section vide comprise entre `%{` et `%}` est la section *déclarations C* des MTSAG. La section *déclarations MTSAG2C*, située entre `%}` et `%%` déclare :

- la grammaire est une grammaire mono-bande :

```
%nb_tapes 1
```

- il y a un seul attribut physique appelé `valeur` de type `int` :

```
%attribute "int" valeur
```

- l'attribut `valeur` du token `nombre` est calculé à partir de la suite de caractères lue sur la bande pour ce token :

```
%token nombre {extern char *yytext; sscanf(yytext,"%i",&$$.valeur);}
```

- quatre autres tokens sans attribut sont utilisés pour noter l'addition, la multiplication et les parenthèses :

```
%token '+' '*' '(' ')'
```

```
/* Fichier arith.cf :  description de la grammaire */
%{
%}

%nb_tapes        1
%attribute       "int" valeur
%token           nombre            { extern char *yytext; sscanf (yytext, "%i", &$$.valeur); }
%token           '(' ')' '+' '*'
%terminal_attribs                  { $$.valeur = $$.right[0].valeur; }
%default_nonterminal_attribs       <valeur>
%default_constraint             {} { if ($$.valeur > $!.valeur) $!.valeur = $$.valeur; }

%%
S : S '+' S             { $$.valeur = $1.valeur + $3.valeur; }
  | S '*' S             { $$.valeur = $1.valeur * $3.valeur; }
  | '(' S ')'           { $$.valeur = $2.valeur; }
  | nombre              { $$.valeur = $1.valeur; }
  ;
%%
#include "arith_lex.c"  /* le code de l'analyseur lexical est inclus ici */

void affiche_parentheses (parse_tree_t *s)
{ int i;

  if (s->token[0] == NULL)
    {
       printf ("(");
       for (i = 0; i < s->nb_subtrees; i++)
         affiche_parentheses (s->subtree[i]);
       printf (")");
    }
  else
    switch (s->token[0]->lex_code)
      {
      case '+':    printf ("+");                               break;
      case '*':    printf ("*");                               break;
      case nombre: printf ("%i", ATTRIB_valeur(s->attrib));   break;
      case '(':
      case ')':    break;
      }
}

int user_main (void)
{ FILE *f[nb_tapes];

  f[0] = stdin;       /* La chaîne à analyser est lue sur l'entrée standard */
  if (mtsagparse (f) == 0)
    { parse_tree_t *s;
      printf ("plus grande valeur : %i\n",
              ATTRIB_valeur(retrieve_attrib (NULL, START_NONTERMINAL, 0, COMPOSITE_LEN)));
      s = build_parse_tree (START_NONTERMINAL, 0, COMPOSITE_LEN);
      affiche_parentheses (s);
      printf ("\n");
    }
  else
    printf ("erreur de syntaxe\n");
  return 0;
}
```

Figure A.1: MTSAG à une bande `arith.cf` calculant l'arbre de plus grande valeur d'une expression arithmétique pour la grammaire ambiguë $S \to S + S \mid S \times S \mid (S) \mid \text{nombre}$.

```
/* Fichier arith_lex.l :  description des terminaux de l'analyseur syntaxique */
%%
"+"             { return '+'; }
"*"             { return '*'; }
"("             { return '('; }
")"             { return ')'; }
[0-9]+          { return nombre; }
<<EOF>>         { return 0; }
.               {}      /* Tout caractère inutile est supprimé */
```

Figure A.2: Description de l'analyseur lexical associé à la MTSAG de la figure A.1.

- la valeur de l'attribut physique `valeur` des terminaux est obtenue à partir de l'attribut physique des tokens :

```
%terminal_attribs { $$.valeur = $$.right[0].valeur; }
```

  Même s'il n'y a qu'une seule bande, tokens et terminaux sont deux concepts différents et l'initialisation des attributs d'un token n'est pas synonyme d'initialisation des attributs du terminal correspondant.

- l'attribut physique par défaut des non-terminaux est `valeur` :

```
%default_nonterminal_attribs <valeur>
```

- la contrainte de sélection par défaut sur les non-terminaux est vide, et la contrainte d'optimisation est une contrainte de maximisation :

```
%default_constraint  {}
                 { if ($$.valeur > $!.valeur) $!.valeur = $$.valeur; }
```

  Pour les familiers de YACC, précisons que la pseudo-variable `$!` permet d'accéder aux attributs physiques du non-terminal déjà présent dans la file de priorité.

Après la section *déclarations MTSAG2C*, la section *grammaire* (encadrée par deux `%%`) déclare la MTSAG utilisée. La syntaxe et les conventions en usage sont très proches de celles de YACC. Des pseudo-variables `$$` et `$`i suivies d'un point et d'un nom d'attribut physique sont utilisées pour accéder aux attributs physiques des symboles d'une règle.

Comme un fichier MTSAG2C ne contient pas de description des caractères composant les tokens, l'analyseur MTSAG2C utilise une fonction `yylex` chargée de décomposer la chaîne d'entrée en tokens utilisables. La section *code C supplémentaire* inclut donc le code d'un analyseur lexical automatiquement généré par `LEX` d'après la description de la figure A.2.

Nous ne décrirons pas ici la syntaxe utilisée par l'outil `LEX`. Le lecteur intéressé est invité à consulter la documentation de `LEX` (Paxon & Gong, 1993). Les 4 lignes suivant le `%%` du fichier `arith_lex.l` permettent de reconnaître les tokens `+`, `*`, `(` et `)` utilisés par la grammaire de `arith.cf`. La valeur retournée est précisément le code ASCII des tokens à reconnaître. L'expression régulière `[0-9]+` suivant ces quatre lignes est celle associée au token `nombre`. En l'occurrence, un `nombre` est une suite de chiffres. La valeur `nombre` retournée est précisément l'identificateur `nombre` utilisé dans la déclaration `%token nombre` du fichier MTSAG2C. La fin du fichier d'entrée (`<<EOF>>`) définit la fin de la chaîne analysée grâce à la valeur 0 retournée.

Nous avons utilisé l'outil `LEX` pour la génération de l'analyseur lexical parce que `LEX` est parfaitement adapté à cet usage et parce qu'un analyseur MTSAG2C est conçu pour appeler un analyseur lexical utilisant les mêmes conventions que ceux générés par `LEX`. L'analyseur lexical créé par `LEX` définit en particulier une variable `yytext` contenant la suite de caractères du token qui vient d'être reconnu. Cette variable `yytext` est utilisée dans la grammaire de la figure A.1 pour calculer l'attribut `valeur` du token `nombre`.

La fonction `main` est appelée dès le début de l'exécution, comme dans tout programme C normal. La première instruction de `main` est un appel à la fonction `mtsagparse` avec en paramètre un tableau de descripteurs de fichiers (type `FILE **`). Le $i^{\text{ème}}$ élément du tableau est un pointeur sur un descripteur du fichier contenant les caractères de la bande i avant analyse lexicale. Dans l'exemple de la figure A.1, il n'y a qu'une bande, lue sur l'entrée standard (`stdin`). Si l'analyse s'est bien passée, c'est-à-dire si `mtsagparse` a pu reconnaître une chaîne appartenant au langage avec un attribut valide, alors une valeur nulle est retournée, et l'attribut du non-terminal S tel que $S \rightarrow^* a_{\vec{1}} \ldots a_{\vec{n}}$ est affiché. La valeur de $\vec{n}$ est codée dans la macro `COMPOSITE_LEN` (cf. section A.6.3). L'attribut affiché respecte ici la contrainte de maximisation définie dans `%default_constraint`. La fonction prédéfinie `build_parse_tree` construit un arbre de dérivation associé à l'attribut trouvé, et la fonction `affiche_parentheses`, définie par l'utilisateur dans la section *code C supplémentaire*, affiche le parenthésage de l'expression grâce à l'arbre de dérivation obtenu.

A.2.2 Génération et exécution de l'analyseur

L'analyseur syntaxique est créé à partir du fichier `arith.cf` à l'aide de MTSAG2C et grâce à la commande suivante :

```
mtsag2c arith.cf -o arith.c
```

Le code de l'analyseur lexical est obtenu avec `LEX` :

```
lex -t arith_lex.l > arith_lex.c
```

Finalement, il ne reste plus qu'à compiler l'analyseur syntaxique contenu dans `arith.c` :

```
cc arith.c -o arith
```

Il est d'usage d'intégrer les trois commandes précédentes dans un fichier `Makefile` approprié. En outre, si le code C utilisant l'analyseur syntaxique généré par MTSAG2C comporte de nombreuses fonctions, il peut être avantageux de découper le programme en plusieurs fichiers compilés séparément. Les seules fonctions ou variables devant obligatoirement être définies, quelle que soit la MTSAG, sont `main` pour le point d'entrée principal et `yylex`, `yyin`, `yyrestart` pour l'analyse lexicale. Ces trois derniers identificateurs sont automatiquement définis si `LEX` est utilisé pour générer l'analyseur lexical.

Nous pouvons maintenant fournir une trace d'exécution typique du programme `arith` obtenu après compilation :

```
> arith
8*9+5*4
^D
plus grande valeur : 448
((8)*(((9)+(5))*(4)))
> arith
8*(2*3+4*5*6+3)+5*3
^D
plus grande valeur : 15240
((8)*(((((2)*(((3)+(4))*((5)*((6)+(3))))))+(5))*(3)))
>
```

Si l'expression de plus petite valeur est en fait cherchée, alors la contrainte par défaut devient :

```
%default_constraint
  {} { if ($$.valeur < $!.valeur)  $!.valeur = $$.valeur; }
```

Et, à partir des mêmes expressions que dans la trace précédente, les résultats obtenus sont :

```
> arith
8*9+5*4
^D
plus petite valeur : 92
(((8)*(9))+((5)*(4)))
> arith
8*(2*3+4*5*6+3)+5*3
^D
plus petite valeur : 1047
(((8)*((((2)*(3))+(((4)*((5)*(6)))+(3)))))+((5)*(3)))
>
```

Lorsque les nombres utilisés sont supérieurs à 1 (ce qui est le cas ici), l'arbre de plus petite valeur correspond aux priorités usuelles des opérateurs $+$ et $\times$, tandis que l'arbre de plus grande valeur correspond aux priorités inversées.

A.3 Exemple de MTSAG multi-bandes

A.3.1 Le fichier MTSAG2C

La grammaire `frame.cf` de la figure A.3 permet d'aligner deux séquences différentes. Le modèle d'alignement retenu est celui de pénalités dépendant du décalage du cadre ouvert de lecture (Searls & Murphy, 1995). Par rapport à la grammaire précédente (figure A.1), cette grammaire comporte les différences suivantes :

```
%{ %}
%nb_tapes 2
%empty      _        %token                        A U C G    %abbreviation      . A U C G
%attribute  "int" v  %default_nonterminal_attribs < v >      %terminal_attribs  {}
%default_constraint  {} { if ($$.v < $!.v) $!.v = $$.v; }

%%
frame0          : frame0 [. .]  { if ($2[1].left[0].lex_code != $2[2].left[0].lex_code)
                                    $$.v = $1.v + 2; else $$.v = $1.v; }
                | frame1 [_ .]  { $$.v = $1.v + 1; }
                | frame2 [. _]  { $$.v = $1.v + 1; }
                | [. .]         { if ($1[1].left[0].lex_code != $1[2].left[0].lex_code)
                                    $$.v = 2; else $$.v = 0; } ;
frame1          : frame0 [. _]  { $$.v = $1.v + 3; }
                | frame1 [. .]  { if ($2[1].left[0].lex_code != $2[2].left[0].lex_code)
                                    $$.v = $1.v + 3; else $$.v = $1.v + 1; }
                | frame2 [_ .]  { $$.v = $1.v + 3; } ;
frame2          : frame0 [_ .]  { $$.v = $1.v + 3; }
                | frame1 [. _]  { $$.v = $1.v + 3; }
                | frame2 [. .]  { if ($2[1].left[0].lex_code != $2[2].left[0].lex_code)
                                    $$.v = $1.v + 3; else $$.v = $1.v + 1; } ;
%%
#include "frame_lex.c"

void affiche_alignement (parse_tree_t *s, int tape)
{
  int i;

  if IS_TERMINAL (s->symbol)
    printf ("%s", s->token[tape]->syntax ? s->token[tape]->syntax : "-");
  else
    for (i = 0; i < s->nb_subtrees; i++)
      affiche_alignement (s->subtree[i], tape);
}

int main (int argc, char **argv)
{ FILE *f[nb_tapes];    int i;

  if (argc == 1 + nb_tapes)
    { for (i = 0; i < nb_tapes; i++)
        f[i] = fopen (argv[i+1], "r");
      if (mtsagparse (f) == 0)
        { parse_tree_t *s = build_parse_tree (START_NONTERMINAL, 0, COMPOSITE_LEN);
          for (i = 0; i < nb_tapes; i++)
            { affiche_alignement (s, i);
              printf ("\n");
            }
        }
    }
}
```

Figure A.3: MTSAG 2-bandes `frame.cf` permettant d'aligner deux séquences. Cette MTSAG est une transcription de la MTSAG donnée figure 3.6 p.92.

```
%{
%}
delim           [ \t]
ws              {delim}+

%%
"A"|"a"         { return A; }
"C"|"c"         { return C; }
"G"|"g"         { return G; }
"U"|"u"         { return U; }
<<EOF>>         { return 0; }
.               { }
%%
```

Figure A.4: Analyseur lexical pour la MTSAG de la figure A.3.

- la grammaire a deux bandes au lieu d'une : `%nb_tapes 2`,
- l'attribut physique `valeur` a été remplacé par `v`,
- il faut deux tokens entre crochets pour faire un terminal,
- le token vide est utilisé pour introduire des gaps dans les alignements,
- les tokens et les terminaux n'ont pas d'attribut,
- un token spécial "`.`", appelé token abrégé, est défini pour représenter les tokens `A`, `C`, `G` ou `U` dans les règles de la grammaire.

Dans les fonctions d'évaluation d'attributs de la grammaire, l'attribut physique `left` est utilisé à plusieurs reprises, sans avoir été déclaré grâce à `%attribute` et sans avoir été initialisé. En fait, il existe trois attributs physiques toujours définis pour tous les terminaux et non-terminaux de la grammaire (mais pas pour les tokens). Ces trois attributs sont `left`, `right` et `len` (section A.4.5). Dans la fonction d'évaluation de la règle `frame0 : frame0 [. .]`, le test `$2[1].left[0].lex_code != $2[2].left[0].lex_code` vérifie en fait que les deux tokens du terminal `[. .]` sont différents (cf. section A.4.4 pour une autre description possible).

Le principe de la fonction d'affichage des alignements (`affiche_alignement`) est très proche du principe de la fonction d'affichage des parenthèses de la figure A.1 : il s'agit simplement d'une descente en profondeur d'abord de l'arbre de dérivation, avec affichage des feuilles de l'arbre sur une bande donnée. La fonction `main`, une fois l'analyse effectuée, affiche d'abord la projection des feuilles sur la première bande, puis la projection des feuilles sur la deuxième bande.

Comme les tokens de la MTSAG correspondent simplement aux symboles A, C, G, U, l'analyseur lexical reconnaissant ces tokens est très simple et est donné figure A.4.

A.3.2 Génération et exécution de l'analyseur

Les commandes permettant de créer un programme d'alignement `frame` à partir de la grammaire précédente sont :

```
mtsag2c frame.cf -o frame.c
lex -t frame_lex.l > frame_lex.c
cc frame.c -o frame
```

Étant données deux séquences, le résultat de l'exécution du programme `frame` est le suivant :

```
> frame
GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGAGGAUGCCCUGGGUUGGAAUCCCAGUGGGUCCA
^D
GGGCGAAUAGUGUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
^D

GGGCCCAUAGCUCAGUGGUAGAGUGCCUCCUUUGCAAGGA---GGAUGCCCUGGGUUGGAAUCCCAGUGGGU---CCA
GGGCGAAUAG-U--GUCAGCGGGAGCACACCAGACUUGCAAUCUGGUAGGGAGGGUUCGAGUCCCUCUUUGUCCACCA
```

Nous pouvons constater sur cet exemple que le caractère utilisé par défaut pour le token vide dans les alignements est `-`, ce qui correspond à la notation usuelle des gaps en biologie.

A.4 Syntaxe d'un fichier MTSAG2C

MTSAG2C lit un fichier décrivant une MTSAG et ses contraintes de sélection et d'optimisation, et produit un fichier en langage C contenant un analyseur syntaxique pour la MTSAG décrite. Un fichier MTSAG2C possède une syntaxe spécifique dont deux exemples ont été donnés dans les sections A.2 et A.3. Dans cette section, nous allons décrire en détail cette syntaxe.

A.4.1 Structure générale d'un fichier MTSAG2C

Un fichier MTSAG2C comporte toujours quatre sections séparées par des délimiteurs spéciaux :

`%{`

déclarations C

`%}`

déclarations MTSAG2C

`%%`

grammaire

`%%`

Code C supplémentaire

Les commentaires sont autorisés n'importe où et sont délimités par '/* ... */'.

La section *déclarations C* contient n'importe quel code C, et en particulier les définitions de macros, fonctions et variables qui sont utilisées dans les portions de code C des deux sections suivantes. En fait, n'importe quel code C peut apparaître dans cette section, y compris des instructions au préprocesseur pour inclure des en-têtes standard. Les déclarations sont copiées sans changement au début du code C généré par MTSAG2C. À la différence de YACC, les délimiteurs '`%{`' et '`%}`' doivent être présents même si la section ne contient pas de code.

La section *déclarations MTSAG2C*, décrite en A.4.2, contient toutes les déclarations destinées à MTSAG2C et distinctes des règles de grammaire et des fonctions d'évaluation d'attributs. Cela inclut les déclarations de tokens, d'attributs, de tokens abrégés, de fonctions génériques et de contraintes.

La section *grammaire*, décrite en A.4.3, est la section principale, comprise entre deux '`%%`'. Elle contient toutes les règles de la MTSAG ainsi que les fonctions d'évaluation d'attributs associées à ces règles. Des contraintes peuvent aussi apparaître dans cette section si l'utilisateur le désire.

La section *code C supplémentaire* est copiée sans changement à la fin du code C généré par MTSAG2C. C'est l'endroit idéal pour définir les variables et fonctions déclarées dans la section *déclarations C*. La définition de l'analyseur lexical doit obligatoirement s'y trouver, puisque ce dernier retourne les tokens à l'aide de macros définies uniquement dans le fichier généré par MTSAG2C. Généralement, la définition de la fonction `main` est également effectuée dans cette section. Dans le cas d'un programme utilisant une MTSAG pour quelques calculs simples, tout le reste du code C peut aller dans cette section. Si le programme doit être plus important, il est bien sûr possible d'utiliser tous les mécanismes prévus par le langage C pour le découpage des programmes en plusieurs fichiers.

A.4.2 Structure de la section *déclarations MTSAG2C*

La section *déclarations MTSAG2C* peut contenir les éléments suivants, sans que l'ordre indiqué soit obligatoire (à quelques exceptions près qui seront énumérées plus loin) :

- les options de compilation (optionnel)

  `%mode` "*chaîne de caractères*"

- le nombre de bandes de la grammaire (optionnel)

  `%nb_tapes` *nombre*

- la déclaration du token vide (optionnel)

  `%empty` *symbole*

- la définition des tokens non vides (obligatoire)

`%token` *liste de tokens* *code C optionnel de calcul des attributs des tokens*

- la déclaration des tokens abrégés (optionnel)

  `%abbreviation` *token abrégé* *liste de symboles*
  ⋮
  token abrégé *liste de symboles*

- la définition des attributs physiques (optionnel)

  `%attribute` "*type du langage C*" *liste de symboles*

- la définition de la fonction de calcul des attributs des terminaux (obligatoire)

  `%terminal_attribs {` *code C de calcul des attributs physiques des terminaux* `}`

- la déclaration des attributs physiques par défaut des non-terminaux (optionnel)

  `%default_nonterminal_attribs` < *liste d'attributs physiques* >

- la déclaration des attributs physiques de certains non-terminaux (optionnel)

  `%nonterminal_attribs` < *liste d'attributs physiques* > *liste de non-terminaux*

- les contraintes de sélection et d'optimisation par défaut des non-terminaux (optionnel)

  `%default_constraint {` *code C de la contrainte de sélection* `}`
  `{` *code C de la contrainte d'optimisation* `}`

- les contraintes de sélection et d'optimisation de certains non-terminaux (optionnel)

  `%constraint` *liste de non-terminaux* `{` *code C de la contrainte de sélection* `}`
  `{` *code C de la contrainte d'optimisation* `}`

- les définitions des fonctions génériques d'évaluation d'attributs (optionnel)

  `%rule_model` *partie droite spéciale de règle* `%generic` *symbole*
  < *liste de* "*types du langage C*" >
  `{` *code C de la fonction d'évaluation d'attributs* `}`

Les symboles utilisés dans une MTSAG (attributs, terminaux, non-terminaux, etc.) doivent commencer par une lettre ou un “underscore” (symbole `_`), et être suivis de zéro ou plusieurs lettres, chiffres ou underscores. Donc, les symboles d'une MTSAG sont aussi des symboles valides du langage C. Nous allons détailler ci-dessous chacune des déclarations précédentes.

A.4.2.1 Options de compilation

Syntaxe :

`%mode` "*chaîne de caractères*"

La chaîne de caractères peut contenir n'importe quel caractère à l'exception du guillemet `"`. Les caractères sont interprétés tels quels, c'est-à-dire qu'il n'y a pas d'interprétation de codes de contrôle spéciaux comme `\t` dans les chaînes du langage C.

Cette déclaration est *optionnelle* et sert uniquement à spécifier des options de compilation à MTSAG2C. Cette déclaration est principalement utilisée avec l'option `-dynamic` spécifiée lorsque la génération des états doit se faire dynamiquement (cf. section 4.2.2 p.189) :

```
%mode "-dynamic"
```

La liste des options de MTSAG2C est donnée dans la section A.7. Une option donnée dans la grammaire a précédence sur une option de la ligne de commande.

A.4.2.2 Nombre de bandes

Syntaxe :

`%nb_tapes` *nombre*

Le nombre de bandes doit être compris entre 1 et 9. Les valeurs le plus souvent utilisées sont 1 (pour les SAG) et 2 (pour les MTSAG à deux bandes). Les analyseurs syntaxiques générés sont d'autant plus lents et gourmands en mémoire que le nombre de bandes est grand (cf. théorème 2.2 p.83)

Cette déclaration est *optionnelle*. En son absence, le nombre de bandes est égal à 1.

Exemple A.1

```
%nb_tapes 2
```

déclare une MTSAG 2-bandes.

A.4.2.3 Token vide

Syntaxe :

`%empty` *symbole*

Le *symbole* de la déclaration `%empty` pourra être utilisé dans la suite de la MTSAG pour représenter le token vide (noté ϵ lors de la présentation théorique, ou - dans les alignements). Le token vide est inutile avec les SAG, puisque les règles vides sont interdites. En revanche, il est indispensable pour créer des MTSAG utiles car une MTSAG dont aucune règle ne contient de terminal utilisant le token vide est équivalente à une SAG (cf. remarque 2.5 p.84).

La déclaration du token vide est *optionnelle* et un token vide par défaut est déclaré par MTSAG2C en cas d'absence de `%empty`. Si la déclaration du token vide est présente, alors elle doit être unique et précéder toute autre définition de token.

Exemple A.2

```
%empty _
```

Le caractère `_` est souvent utilisé pour déclarer le token vide car c'est le caractère autorisé dans les tokens se rapprochant le plus du caractère - habituellement rencontré dans les alignements.

A.4.2.4 Tokens non vides

Syntaxe :

`%token` *liste de tokens* *code C optionnel de calcul des attributs des tokens*

Les tokens de MTSAG2C sont de deux types :

- Les symboles, comme `nombre` dans la figure A.1 ou `U` dans la figure A.3. Ces tokens sont déclarés dans l'analyseur syntaxique généré par MTSAG2C sous la forme d'une macro dont le nom est celui du token et dont la valeur est un entier strictement positif. L'analyseur lexical doit utiliser cette macro pour retourner la valeur correspondante lorsqu'il a reconnu le token dans la chaîne de caractères à analyser.

- Les caractères explicites, comme `'('` ou `'+'` dans la figure A.1. Ces *tokens spéciaux* sont formés d'un seul caractère entre deux guillemets simples. Ce caractère doit être différent du guillemet simple `'`, du guillemet double `"` ou du retour chariot. Le caractère entre guillements est interprété tel quel, c'est-à-dire qu'il n'y a pas d'interprétation de code de contrôle spécial comme `\t` dans les caractères du langage C.

  L'analyseur lexical doit retourner le code ASCII du caractère lorsqu'il a reconnu la suite de caractères correspondant au token. Ce code ASCII est facilement obtenu en langage C en utilisant la même syntaxe que lors de la déclaration du token spécial, c'est-à-dire en encadrant le caractère par des guillemets simples.

  En général, il est préférable, car plus clair, de retourner l'entier `'+'` (i.e. l'entier égal au code ASCII de `+`) lorsque la chaîne `"+"` a été reconnue.

  Les caractères explicites ont été introduits pour maintenir une compatibilité syntaxique avec YACC et éviter à l'utilisateur de fournir des symboles dont le nom représente un unique caractère : il est plus simple d'écrire `'+'` ou `'('` dans une MTSAG que `PLUS` ou `PAR_OPEN`.

Le *code C optionnel de calcul des attributs des tokens* permet, comme son nom l'indique, de calculer les attributs physiques de tous les tokens de la liste précédant ce code.

Exemple A.3 *Le code suivant donne la valeur 0 à l'attribut physique* `valeur` *des quatre tokens* `'('`, `')'`, `'+'` *et* `'*'`

```
%token '(' ')' '+' '*'   { $$.valeur = 0; }
```

Si un attribut physique est utilisé pour un token (`valeur` dans l'exemple précédent), alors il doit avoir été déclaré auparavant à l'aide de `%attribute`. La syntaxe d'accès aux attributs physiques des tokens dans le code d'initialisation est la même que la syntaxe d'accès aux attributs physiques des non-terminaux réduits dans les règles de grammaire, à ceci près que les attributs physiques spéciaux `left`, `right` et `len` sont interdits (section A.4.5).

A.4.2.5 Tokens abrégés

Syntaxe :

`%abbreviation` *token abrégé liste de symboles*
⋮
token abrégé liste de symboles

Les tokens abrégés sont particulièrement utiles dans les MTSAG pour déclarer des règles de grammaire valables pour une série de tokens ou de multiplets de tokens différents.

Exemple A.4 *Soit la définition suivante du token abrégé* • *:*

```
%abbreviation • A U C G
```

Toute occurrence du token abrégé • dans une règle de MTSAG entraînera le remplacement de cette règle par quatre règles de grammaire contenant chacune, à la place du •, un des quatre tokens de la définition de ce •.

Exemple A.5 *Soit la définition suivante de la paire de tokens abrégés* `(` *et* `)` *:*

```
%abbreviation ( A U G C    ) U A C G
```

Toute occurrence sur une même bande de la paire de tokens abrégés précédente dans une règle de MTSAG entraînera le remplacement de cette règle par quatre règles de grammaire contenant chacune, à la place de la paire ( ), une des quatre paires de tokens de la définition. Ainsi, la règle `DoubleStrand : ( DoubleStrand )` *sera remplacée par*

```
DoubleStrand : A DoubleStrand U
DoubleStrand : U DoubleStrand A
DoubleStrand : G DoubleStrand C
DoubleStrand : C DoubleStrand G
```

La notion précédente peut être étendue à des abréviations d'ordre supérieur, comme dans la déclaration suivante :

```
%abbreviation ( A U   .  U U  ) U A
```

L'ordre maximal a été fixé à 10.

Un token abrégé est formé d'un point (ou d'une parenthèse ouvrante, ou d'une parenthèse fermante), suivi de zéro ou plusieurs lettres, chiffres, underscores, points, parenthèses ouvrantes, parenthèses fermantes[1]. La longueur totale d'un nom de token abrégé ne peut dépasser cinq caractères.

Tous les tokens utilisés à droite d'un token abrégé doivent avoir été déjà définis. Il est possible d'utiliser le token vide. Il doit y avoir le même nombre de tokens à droite de chaque token abrégé d'une déclaration `%abbreviation`. Toute déclaration `%abbreviation` définit une *abreviation*. Le $i^{\text{ème}}$ token abrégé d'une abréviation doit être différent du $i^{\text{ème}}$ token abrégé d'une autre abréviation, sauf si tous les tokens abrégés des deux abréviations sont identiques. Cela revient à dire qu'il est possible d'écrire :

```
%abbreviation ( A ) U
%abbreviation ( U ) A
%abbreviation ( G ) C
%abbreviation ( C ) G
```

à la place de `%abbreviation ( A U G C   ) U A C G`, mais qu'il est impossible de leur ajouter les abréviations suivantes :

```
%abbreviation . A ) A
%abbreviation ( A . A
%abbreviation ( A ) A . A
```

A.4.2.6 Attributs physiques

Syntaxe :

`%attribute "`*type du langage C*`"` *liste de symboles*

La *liste de symboles* représente une liste de noms d'attributs physiques de symboles de la MTSAG séparés par un ou plusieurs espaces. Tous les attributs physiques utilisés dans une MTSAG doivent avoir été déclarés au préalable. Le *type du langage C* pourra être un type prédéfini comme `int` ou `double`, mais aussi un nom de structure déclaré à l'aide d'un `typedef` dans la section *declarations C*. Pour qu'une déclaration d'attribut soit valide, il faut que la déclaration :

[1] l'expression régulière utilisée pour décrire les tokens abrégés est `[.()][A-Za-z0-9_.()]*`

```
%attribute "type" symbole
```

puisse être transformée en une définition C valide

```
type symbole;
```

Il est interdit de déclarer deux attributs physiques ayant le même nom ou de déclarer un attribut physique ayant le nom d'un des attributs physiques spéciaux `left`, `right` ou `len`.

MTSAG2C n'associe pas systématiquement tous les attributs physiques à tous les symboles de la grammaire (cf. section 4.3 p.192). Un attribut physique est associé à un symbole seulement si ce dernier l'utilise. Cela permet de gagner de la place en mémoire lors de l'exécution de l'analyseur généré par MTSAG2C. Les déclarations suivantes ont pour but de définir plus précisément quels attributs physiques sont utilisés par quels symboles.

Exemple A.6

```
%attribute "double" Energy
%attribut "char *" Pointer1 Pointer2
```

A.4.2.7 Attributs physiques des terminaux

Syntaxe :

```
%terminal_attribs { code C de calcul des attributs physiques des terminaux }
```

Cette définition **obligatoire** permet à la fois de définir les attributs physiques des terminaux (tous les terminaux ont les mêmes attributs physiques), et le code C utilisé pour calculer ces attributs physiques. En fait, si `$$.`*attribut* apparaît dans le code C, alors MTSAG2C considère que *attribut* fait partie des attributs physiques des terminaux, même si sa valeur n'est pas changée (il n'y a donc pas de vérification de cohérence du code C). Ce comportement est justifiable par le fait que l'attribut physique d'un terminal ne peut pas dépendre d'un attribut physique d'un autre terminal ou non-terminal, puisque le calcul des attributs dans un arbre de dérivation se fait de bas en haut. Notons que les attributs physiques spéciaux `left`, `right` et `len` sont autorisés, ainsi que la référence aux attributs physiques des tokens car ces derniers sont calculés une fois pour toutes au moment de l'analyse lexicale (cf. section A.6.1).

Exemple A.7

```
%terminal_attrib {}
```

La déclaration précédente est celle d'un code d'initialisation vide. Dans ce cas, les terminaux n'ont pas d'attribut physique.

Dans le cas d'une MTSAG mono-bande, les tokens restent différents des terminaux et l'initialisation des attributs physiques des tokens ne dispense pas de l'initialisation des attributs physiques des terminaux (figure A.1).

A.4.2.8 Attributs physiques par défaut des non-terminaux

Syntaxe :

`%default_nonterminal_attribs <` *liste d'attributs physiques* `>`

Les attributs physiques des non-terminaux ne sont pas déterminés implicitement comme pour les terminaux car une détermination implicite serait incompatible avec une détection des incohérences dans les fonctions d'évaluation d'attributs (cf. section A.5). Il faut donc déclarer explicitement quels sont les attributs physiques des non-terminaux. Lorsque la plupart des non-terminaux ont les mêmes attributs physiques (figure A.3), il est suffisant de fournir la liste de ces attributs physiques dans une déclaration `%default_nonterminal_attribs`.

Remarque A.1 Cette déclaration est *optionnelle*. Cependant, tous les non-terminaux doivent avoir des attributs physiques. Donc, s'il existe des non-terminaux de la MT-SAG dont les attributs physiques n'ont pas été explicitement déclarés à l'aide de `%non_terminal_attrib`, il faut ajouter une déclaration `%nonterminal_attribs` pour ces non-terminaux ou inclure une déclaration `%default_nonterminal_attribs`.

Exemple A.8

```
%default_nonterminal_attribs < Energy SumEnergy >
```

A.4.2.9 Attributs physiques de certains non-terminaux

Syntaxe :

`%nonterminal_attribs <` *liste d'attributs physiques* `>` *liste de non-terminaux*

Cette déclaration est le pendant de la déclaration précédente (cf. remarque A.1).

Exemple A.9

```
%nonterminal_attribs < ExtraEnergy > InternalLoop
```

A.4.2.10 Contraintes de sélection et d'optimisation des non-terminaux

Syntaxe :

`%default_constraint {` *code C de la contrainte de sélection* `}`
`{` *code C de la contrainte d'optimisation* `}`

Les contraintes de sélection et d'optimisation des non-terminaux peuvent être définies grâce à `%default_constraint` ou à `%constraint`. Tous les non-terminaux doivent avoir une contrainte de sélection et une contrainte d'optimisation. Chacune de ces contraintes peut être vide, mais cela doit apparaître explicitement. Une contrainte vide est une contrainte dont le code C ne fait rien. Il est préférable de réduire une telle contrainte à la chaîne vide entre accolades (i.e. aucun espace ne doit apparaître entre le { et le }) pour permettre à MTSAG2C de le détecter et d'optimiser le code généré en conséquence. Dans les figures A.1 et A.3, les contraintes de sélection sont vides.

Remarque A.2 Si les contraintes par défaut ne sont pas implicitement définies par `%default_constraint`, tous les non-terminaux doivent alors avoir des contraintes explicitement déclarées grâce à `%constraint`.

Dans la contrainte de sélection, la pseudo-variable `$$` est utilisée pour accéder aux attributs physiques du non-terminal qui vient d'être réduit (cf. étape 5 de l'algorithme GCP p.80). Si la contrainte de sélection retourne la valeur 0, alors le non-terminal n'est pas sélectionné. Si la contrainte retourne une valeur non nulle ou se termine sans retourner de valeur (c'est le cas lorsqu'elle est vide), le non-terminal et ses attributs physiques sont considérés plus avant. Si le non-terminal n'est pas déjà présent dans la file de priorité, il y est alors ajouté ainsi que ses attributs physiques (la contrainte de sélection peut donc changer les attributs physiques d'un non-terminal avant son ajout à la file de priorité ou avant l'exécution de la contrainte d'optimisation). Si le non-terminal est déjà présent dans la file de priorité, la contrainte d'optimisation est exécutée.

Dans la contrainte d'optimisation, la pseudo-variable `$$` permet d'accéder aux attributs physiques du non-terminal qui vient de passer par la contrainte de sélection. La pseudo-variable `$!` permet d'accéder aux attributs physiques du non-terminal déjà présent dans la file de priorité (cf. sections A.4.5 et A.5). Lorsque la contrainte d'optimisation d'un non-terminal est vide, seule la première dérivation rencontrée en cours d'analyse est retenue.

Exemple A.10

```
%default_constraint
{
  if (labs ($$[1].len - $$[2].len) < ALIGN_CONSTRAINT)
  return 0;
}
{
  if ($$.E < $!.E) {
    $!.E = $$.E;
    $!.extra_attrib = $$.extra_attrib;
  }
}
```

La contrainte de sélection par défaut est une contrainte d'alignement (cf. section 3.2 p. 93). La contrainte d'optimisation par défaut minimise l'attribut `E` *des non-terminaux en tenant compte de l'attribut supplémentaire* `extra_attrib` *des non-terminaux.*

A.4.2.11 Contraintes de sélection et d'optimisation de certains non-terminaux

Syntaxe :

`%constraint` *liste de non-terminaux* `{` *code C de la contrainte de sélection* `}`
`{` *code C de la contrainte d'optimisation* `}`

Cette déclaration est le pendant de la déclaration précédente. Elle est utilisée avec les non-terminaux qui ne peuvent utiliser les contraintes par défaut, soit parce qu'ils n'ont pas des attributs par défaut, soit parce qu'ils vérifient d'autres contraintes d'optimisation ou de sélection que celles par défaut.

Exemple A.11 *Il aurait été possible de redéfinir la contrainte par défaut de la figure A.3 de la manière suivante :*

```
%constraint frame0 frame1 frame2 {} { if ($$.v < $!.v) $!.v = $$.v; }
```

Pour des raisons de compatibilité avec les version antérieures de MTSAG2C, il est possible de définir des contraintes dans la section *grammaire*.

A.4.2.12 Fonctions génériques d'évaluation d'attributs

Syntaxe :

```
%rule_model partie droite spéciale de règle
%generic    nom de fonction  < liste de "types du langage C" >
            { code C de la fonction d'évaluation d'attributs }
```

Lorsqu'une MTSAG contient plusieurs règles dont la séquence des types des symboles (terminaux ou non-terminaux) des parties droites et les fonctions d'évaluation d'attributs sont identiques à quelques constantes prêt, alors il est possible d'utiliser les fonctions génériques pour remplacer les différentes fonctions d'évaluation d'attributs par une unique fonction générique. Cela permet d'une part d'éviter de réécrire plusieurs fois la même fonction (cf. figures 1.13 p.68, 3.22 p.139, 3.26 p.142, 3.41 p.163) et d'autre part de diminuer la taille de l'analyseur généré par MTSAG2C. Nous allons illustrer notre propos par un exemple. Supposons que nous souhaitions calculer le nombre d'arbres de dérivation d'une expression arithmétique simple (cf. figure A.1 et section A.2). Sans fonction générique, la grammaire ressemble à :

```
%{ %}
%attribute "int" valeur
%token          nombre    {}
%default_constraint       {} { $!.valeur += $$.valeur; }
%%
S : S '+' S               { $$.valeur = $1.valeur * $3.valeur; }
  | S '*' S               { $$.valeur = $1.valeur * $3.valeur; }
  | '(' S ')'             { $$.valeur = $2.valeur; }
  | nombre                { $$.valeur = 1; }
  ;
%%
```

Notons la forme particulière de la contrainte d'optimisation permettant effectivement de calculer le nombre d'arbres de dérivation. Une telle contrainte avait déjà été utilisée pour sommer des probabilités de séquences (figures 3.22 p.139 et 3.26 p.142). Dans la MTSAG que nous venons de donner, deux fonctions d'évaluation d'attributs identiques sont utilisées avec deux règles très similaires dont le format type est :

Non-terminal : Non-terminal Terminal Non-terminal

```
{$$.valeur = $1.valeur * $3.valeur; }
```

L'introduction d'une fonction générique permet de simplifier l'écriture de ces règles quasi identiques :

```
%{ %}
%attribute "int" valeur
%token           nombre    {}
%default_constraint     {} { $!.valeur += $$.valeur; }

%rule_model N T N
%generic multiplie < >  { $$.valeur = $1.valeur * $3.valeur; }

%%

S : S '+' S             % multiplie <>
  | S '*' S             % multiple <>
  | '(' S ')'           { $$.valeur = $2.valeur; }
  | nombre              { $$.valeur = 1; }
  ;
%%
```

Comme nous pouvons le constater sur cet exemple, une fonction générique, appelée ici `multiplie`, doit d'abord être définie dans la section *déclarations MTSAG2C*, puis utilisée dans la section *grammaire*. Lors de l'utilisation d'une fonction générique la déclaration `%generic` précédant le nom de la fonction a été remplacée par un simple `%`.

Pour définir une fonction générique, il faut commencer par donner un modèle de la règle qui utilisera cette fonction. Les symboles spéciaux N et T, signifiant respectivement "n'importe quel non-terminal" et "n'importe quel token" (cf. section A.4.4 pour l'écriture des terminaux à partir des tokens) sont utilisés à cet effet. Si le modèle de la règle est vide (i.e. `%rule_model` est immédiatement suivi par `%generic`), alors n'importe quelle règle peut utiliser la fonction générique, mais l'accès aux attributs physiques des symboles de la partie droite devient impossible puisque la taille de la partie droite n'est pas connue au moment de l'écriture de la fonction générique.

Une fois le modèle de règle déclaré, la fonction générique doit être nommée. La définition de deux fonctions génériques ayant le même nom est signalée comme étant une erreur.

Ensuite, le type des paramètres de la fonction générique est indiqué entre `< >` (aucun paramètre dans l'exemple précédent). Ce type doit être un type C valide obéissant aux mêmes règles que le type utilisé pour définir les attributs.

Exemple A.12 *La fonction générique suivante, appelée* `foo`*, prend deux paramètres, le premier étant de type* `double` *et le deuxième de type* `int`.

```
%rule_model N N %generic foo < "double" "int" >
                {
                  if ($#[2] == 1)
```

```
            $$.valeur = $1.valeur + $2.valeur + $#[1];
          else
            $$.valeur = $1.valeur * $2.valeur * $#[1];
        }
```

Le code C de la fonction d'évaluation d'attributs s'écrit comme si la fonction générique était une fonction d'évaluation d'attributs normale. Le code dispose en plus d'une pseudo-variable spéciale `$#` à utiliser avec les paramètres de la fonction générique (cf. section A.4.5). Dans l'exemple précédent, `$#[1]` est la valeur du premier paramètre et `$#[2]` est la valeur du deuxième paramètre. Les paramètres ont une valeur définie au moment de l'utilisation de la fonction générique.

```
NonTerminal_A : NonTerminal_B NonTerminal_C   % foo <"0.34563456" "1">
              | NonTerminal_D NonTerminal_E   % foo <"1.789789789" "0">
              ;
```

L'exemple précédent utilise deux types simples du langage C. L'initialisation de structures ou de tableaux à l'aide de paramètres de fonctions génériques est un peu plus délicate et peut se faire par l'intermédiaire de pointeurs et de fonctions d'allocation (cf. code du fichier `hmm.cf` dans la distribution standard, ou figure 3.22 p.139 dans le présent document).

Il doit y avoir autant de paramètres lors de la définition d'une fonction générique que lors de son utilisation.

A.4.3 Structure de la section *grammaire*

Une règle de grammaire a la forme générale suivante :

non-terminal : *partie droite*
code C de la fonction d'évaluation d'attributs
;

La partie droite, ou rhs (pour right-hand side), est en fait une liste de terminaux et de non-terminaux. Cette liste ne peut pas être vide (cf. section A.4.4 pour la syntaxe des terminaux).

Lorsque plusieurs règles de grammaire partagent le même non-terminal à gauche, il est possible de les regrouper grâce au caractère '`|`' :

non-terminal : *partie droite*
code C de la fonction d'évaluation d'attributs
| *partie droite*
code C de la fonction d'évaluation d'attributs
⋮
| *partie droite*
code C de la fonction d'évaluation d'attributs
;

Lorsque plusieurs règles de grammaire partagent le même non-terminal à gauche, et les mêmes types de symboles aux mêmes positions dans la partie droite (comme $A \to BaCb$ et $A \to CbDe$), et les mêmes fonctions d'évaluation d'attributs, il est possible de factoriser le code C des règles grâce à la syntaxe suivante :

non-terminal : *partie droite* | ... | *partie droite*
code C de la fonction d'évaluation d'attributs
| *partie droite* | ... | *partie droite*
code C de la fonction d'évaluation d'attributs
⋮
| *partie droite* | ... | *partie droite*
code C de la fonction d'évaluation d'attributs
;

Cette factorisation du code C est donc une forme instantanée de fonction générique (cf. section A.4.2.12). Avec ces notation, l'exemple de la section A.4.2.12 peut aussi s'écrire :

```
%{ %}
%attribute "int" valeur
%token          nombre    { sscanf (yytext, "%i", &$$.valeur); }
%default_constraint    {} { $!.valeur += $$.valeur; }
%%
S : S '+' S
  | S '*' S             { $$.valeur = $1.valeur * $3.valeur; }
  | '(' S ')'           { $$.valeur = $2.valeur; }
  | nombre              { $$.valeur = $1.valeur; }
  ;
%%
```

Le code C des fonctions d'évaluation peut prendre deux formes. Soit il est écrit entre deux accolades, comme n'importe quel bloc C, soit il fait appel à une fonction générique (section A.4.2.12). Pour accéder aux attributs du non-terminal à gauche et des symboles de la partie droite, il faut utiliser les pseudo-variables (section A.4.5).

Les fonctions d'évaluation d'attributs des non-terminaux doivent calculer une valeur pour tous les attributs physiques du non-terminal à gauche, à l'exception des attributs physiques spéciaux prédéfinis `left`, `right` et `len` (cf. section A.5 pour une explication des vérifications effectuées par MTSAG2C)

A.4.4 Syntaxe des terminaux

Chaque occurrence d'un terminal m-bandes dans la MTSAG doit indiquer quels sont les tokens qui le composent. Si $a_1, \ldots, a_m$ sont m tokens, alors on note $[a_1 \ldots a_m]$ le terminal m-bandes possédant le token a_i sur la bande i. Les tokens autorisés sont les tokens déclarés à l'aide de `%token`, le token vide, et les tokens abrégés.

Il est impossible de déclarer explicitement un terminal m-bande ayant le token vide sur toutes les bandes. En revanche, un tel terminal peut-être généré implicitement lorsque la définition du terminal contient uniquement des tokens vides et des tokens abrégés contenant le token vide. Dans un tel cas, un avertissement est affiché lors de l'expansion des tokens abrégés pour signaler que la combinaison de tokens abrégés conduisant à un terminal vide provoque le rejet de la règle de grammaire associée.

Un token abrégé peut être suivi par un caractère = puis un entier k compris entre 1 et le nombre de bandes. Cela signifie alors que le token abrégé généré doit être égal au token de la bande k. Ainsi, le terminal `[. .=1]` contient uniquement des paires de tokens abrégés identiques.

De la même manière, Un token abrégé peut être suivi par un caractère `!` puis un entier k compris entre 1 et le nombre de bandes. Cela signifie alors que le token abrégé généré doit être différent du token de la bande k. Ainsi, le terminal `[. .!1]` contient uniquement des paires de tokens abrégés différents.

Grâce aux deux constructions précédentes, les premières règles de l'exemple de la figure A.3 peuvent se réécrire :

```
frame0          : frame0 [. .=1]  { $$.v = $1.v; }
                | frame0 [. .!1]  { $$.v = $1.v + 2; }
                | frame1 [_ .]    { $$.v = $1.v + 1; }
                | frame2 [. _]    { $$.v = $1.v + 1; }
```

Lorsque le nombre de bandes est égal à 1, il est possible d'abréger la syntaxe d'un terminal m-bandes en omettant les crochets et en écrivant simplement le token.

```
S : S '+' S
  | S '*' S              { $$.valeur = $1.valeur * $3.valeur; }
  | '(' S ')'            { $$.valeur = $2.valeur; }
  | nombre               { $$.valeur = $1.valeur; }
  ;
```

est équivalent à

```
S : S ['+'] S
  | S ['*'] S            { $$.valeur = $1.valeur * $3.valeur; }
  | ['('] S [')']        { $$.valeur = $2.valeur; }
  | nombre               { $$.valeur = $1.valeur; }
  ;
```

Il est possible de faire suivre chaque token d'un **caractère indicatif** qui sera utilisé lors de la construction des arbres de dérivation pour remplir un champ spécial de la structure `parse_tree_t` associée aux sommets des arbres (cf. p.260 pour la définition de `parse_tree_t`). La syntaxe utilisée est `/"`c`"`, où c est un caractère obéissant aux mêmes contraintes que les caractères utilisés par les tokens spéciaux (section A.4.2.4).

Le caractère indicatif c est typiquement utilisé pour indiquer aux procédures d'affichage des arbres de dérivation quels caractères doivent être considérés appariés. Par exemple :

```
S :  A/"("   S   U/")"     { ... }
  |  U/"("   S   A/")"     { ... }
  |  G/"("   S   C/")"     { ... }
  |  C/"("   S   G/")"     { ... } ;
```

Dans chacune des quatre règles précédentes, on indique clairement que les deux terminaux forment une paire. Ces indications ne modifient *ni l'analyse syntaxique, ni la construction des arbres de dérivation.* Elles sont uniquement une aide visuelle pour le créateur de la MTSAG, et elles facilitent l'écriture de procédures d'affichage des arbres de dérivation tenant compte des appariements (MTSAG2C est après tout conçu dans le contexte des structures biologiques et des appariements).

Le *caractère indicatif* par défaut d'un token non abrégé est le point. Le *caractère indicatif* par défaut d'un token abrégé est le premier caractère de ce token abrégé (point ou parenthèse).

Le fait que les caractères indicatifs soient associés aux tokens signifie que lorsque la MTSAG comporte plusieurs bandes, il faut spécifier les caractères directement après chaque token d'un terminal, comme dans les règles suivantes :

```
S       : ['('/"(" ./"("] S [')'/")" ./")"]
        | ['('/"(" _    ] S [')'/")" .    ]
        | ['('/"(" .    ] S [')'/")" _    ]
                                            { ... } ;
```

Il est possible d'ajouter un caractère indicatif au token `T` utilisé dans une partie droite de définition de règle générique (cf. section A.4.2.12). Ce caractère indicatif sera ajouté au token correspondant d'une règle de grammaire lors de l'utilisation de la règle générique, sauf si un caractère indicatif a déjà été explicitement ajouté à ce token dans la définition de la règle.

A.4.5 Syntaxe des pseudo-variables

Des pseudo-variables spéciales ont été créées pour accéder aux attributs physiques des symboles de la grammaire, aux attributs physiques des tokens ou aux paramètres des fonctions génériques. Ces pseudo-variables sont directement inspirées des pseudo-variables de YACC. Elles commencent toutes par le caractère `$`, normalement interdit dans les noms de variables du langage C. Le type d'information retourné par une pseudo-variable dépend directement du caractère suivant ce premier `$` :

- Si c'est un `$`, alors on accède aux attributs physiques du symbole dont on cherche à calculer les attributs. Il peut s'agir du non-terminal réduit dans une fonction d'évaluation d'attributs, dans une contrainte de sélection ou dans une contrainte d'optimisation. Il peut également s'agir d'un token dans le code d'initialisation des attributs d'un token, ou d'un terminal dans le code d'initialisation des attributs des terminaux.

- Si c'est un entier i, alors on accède aux attributs physiques du $i^{\text{ème}}$ symbole de la partie droite d'une règle. Une pseudo-variable `$`i ne peut donc apparaître que dans le code d'une fonction d'évaluation d'attributs. Le premier symbole d'une partie droite de règle porte toujours le numéro 1.

- Si c'est un `!`, alors on accède aux attributs physiques d'un non-terminal déjà présent dans la file de priorité. Une pseudo-variable `$!` ne peut donc apparaître que dans le code d'une contrainte d'optimisation.

- Si c'est un `#`, alors on accède aux paramètres d'une fonction générique. Une pseudo-variable `$#` ne peut donc apparaître que dans le code d'une fonction générique.

Nous noterons ci-après `$?` les pseudo-variables `$$`, `$!`, `$`i, et $a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$ la portion de la chaîne d'entrée m-bandes dérivable du symbole X associé à `$?` au moment où cette pseudo-variable est utilisée.

Nous distinguerons également deux types d'attributs physiques. Les attributs physiques déclarés par l'utilisateur à l'aide de `%attribute` que nous noterons *attrib*, et les attributs physiques spéciaux `left`, `right` et `len` que nous noterons *special*.

Les utilisations possibles des pseudo-variables ont les syntaxes suivantes :

`$?.`*attrib* retourne la valeur de l'attribut physique *attrib* associé à X.

Si X est un non-terminal, il faut que l'attribut ait été déclaré dans la liste des attributs du non-terminal à l'aide d'une déclaration `%nonterminal_attribs` ou `%default_nonterminal_attribs`.

Si X est un terminal et si `$?` se trouve dans le code C de `%terminal_attribs`, alors `$?` est en réalité `$$`, et l'attribut physique est ajouté à la liste des attributs du terminal par MTSAG2C, en même temps qu'il est utilisé dans le code.

Si X est un terminal et ne satisfait pas la condition précédente, alors la pseudo-variable `$?` est en réalité `$`i et elle se trouve dans une fonction d'évaluation d'attributs, éventuellement générique. Pour pouvoir utiliser *attrib* avec X, il faut donc que l'attribut ait été initialisé auparavant dans le code C de `%terminal_attribs`.

Si X est un token alors la pseudo-variable `$?` est en réalité `$$` et elle se trouve dans le code C d'une déclaration `%token` pour initialiser et déclarer l'attribut physique *attrib* des tokens de la déclaration.

`$?.`*special* retourne la valeur de l'attribut physique *special* associé à X, qui ne peut donc pas être un token.

Si *special* est `left`, alors le code associé à $\vec{\imath}$ (cf. section 4.1.3 p.184) est retourné après avoir été incrémenté de 1, l'incrément n'ayant de sens que si le nombre de bandes est égale à 1.

Si *special* est `right`, alors le code associé à $\vec{\jmath}$ est retourné.

Si *special* est `len`, alors $\vec{\jmath} - \vec{\imath}$ est retourné lorsque le nombre de bandes est égal à 1. Une erreur de syntaxe est signalée sinon.

`$?[`k`].`*special* retourne la valeur de l'attribut physique *special* après projection sur la bande k. L'entier k doit donc être compris entre 1 et le nombre total de bandes. Cette syntaxe est équivalente à la précédente lorsque le nombre de bandes est égal à 1.

Si *special* est `left`, alors $\vec{\imath}^{(k)} + 1$ est retourné.

Si *special* est `right`, alors $\vec{\jmath}^{(k)}$ est retourné

Si *special* est `len`, alors $\vec{\jmath}^{(k)} - \vec{\imath}^{(k)}$ est retourné.

Notons que lorsque `len` vaut 0, alors $\vec{\imath}^{(k)} = \vec{\jmath}^{(k)}$. Donc la valeur retournée par `left` est supérieure à la valeur retournée par `right`.

`$?[`k`].`*special*`[`l`]` retourne le $l^{\text{ème}}$ token (de type `tape_token_t`, défini p.262) situé sur la bande k à partir de la position `$?[`k`].`*special*. L'attribut spécial ne peut donc pas être `len`, mais uniquement `left` ou `right`. L'entier k doit bien sûr être compris entre 1 et le nombre total de bandes. Si le nombre total de bandes est égal à 1, alors on peut omettre `[`k`]`, puisque k vaut nécessairement 1. L'entier l doit avoir une valeur choisie de manière à rendre l'opération possible (i.e. il ne faut pas sortir des bandes). Notons que la valeur de l n'est pas vérifiée. Le token retourné correspond à la position $\vec{\imath}^{(k)} + l + 1$ lorsque *special* est left, ou à la position $\vec{\jmath}^{(k)} + l$ lorsque *special* est right. Ainsi, lorsque $l = 0$, le token situé exactement à la position `$?[`k`].`*special* de la bande k est retourné.

`$#[`k`]` retourne la valeur du $k^{\text{ème}}$ paramètre d'une fonction générique. L'entier k doit être compris entre 1 et le nombre de paramètres de la fonction générique.

A.5 Cohérence des fichiers MTSAG2C

Quand une MTSAG contient un grand nombre de règles et d'attributs physiques différents, il est fréquent d'oublier d'initialiser des attributs physiques dans les fonctions d'évaluation, ou d'oublier de recopier des attributs dans les contraintes d'optimisation. MTSAG2C vérifie donc en fin de lecture d'une MTSAG que les déclarations des attributs physiques des terminaux et des non-terminaux sont compatibles avec l'usage qui en est fait. Certaines incompatibilités sont fatales et empêchent la génération d'un code C à partir d'une MTSAG, alors que d'autres sont simplement signalées par MTSAG2C afin que l'utilisateur puisse être averti d'un éventuel problème.

Les vérifications effectuées par MTSAG2C se résument essentiellement à la recherche de toutes les occurrences de pseudo-variables dans les différentes portions de code C d'une MTSAG, puis à l'analyse de ces occurrences. MTSAG2C n'analyse pas syntaxiquement le code C, et ne fait donc pas de différence entre le code C de fonction d'évaluation suivant :

```
{ $$.attrib = 0; }
```

et cet autre code C, tout aussi valide, mais parfaitement inutile :

```
{ $$.attrib == 0; }
```

Dans les deux cas, MTSAG2C suppose que l'utilisateur a donné une valeur à l'attribut physique `attrib` du non-terminal associé à `$$`. En effet, la pseudo-variable `$$.attrib` ne contient jamais de valeur cohérente en début de fonction d'évaluation, et une première utilisation ne peut-être qu'une initialisation.

Dans la discussion suivante, la notion d'attribut physique se rapporte uniquement aux attributs physiques déclarés par l'utilisateur et non aux attributs physiques spéciaux `left`, `right`, `len`. MTSAG2C effectue les vérifications suivantes :

- Tous les non-terminaux doivent avoir une liste d'attributs physiques. Chaque non-terminal doit donc apparaître dans une déclaration `%nonterminal_attribs`, ou une déclaration `%default_nonterminal_attribs` doit avoir été faite.

- Tous les terminaux doivent avoir une liste d'attributs physiques et ces attributs physiques doivent être initialisés. En fait, MTSAG2C connaît la liste des attributs physiques d'un terminal en énumérant les attributs *attrib* utilisés sous la forme d'une pseudo-variable `$$.`*attrib* dans une déclaration `%terminal_attribs`. Ces attributs physiques sont alors supposés correctement initialisés, puisque c'est le but de la déclaration `%terminal_attribs`.

- Lorsque un attribut physique `$`*i*`.`*attrib* est utilisé dans la fonction d'évaluation d'une règle, alors *attrib* doit être un attribut physique déclaré du $i^{\text{ème}}$ symbole de la règle. Ce principe reste valable lorsque la fonction d'évaluation est une fonction générique puisque la vérification est faite pour chaque instance de la fonction générique.

- Les contraintes de sélection et d'optimisation d'un non-terminal ne peuvent utiliser que des attributs déclarés du non-terminal. Autrement dit, l'attribut physique *attrib* de `$$.`*attrib* dans une contrainte de sélection ou d'optimisation, ou celui de `$!.`*attrib* dans une contrainte d'optimisation, doit être un attribut de la liste d'attributs physiques du non-terminal associé à `$$` et `$!`.

- MTSAG2C affiche un message d'avertissement (mais pas une erreur) lorsque un attribut physique *attrib* d'un non-terminal n'est pas utilisé dans la contrainte d'optimisation du non-terminal, que ce soit sous la forme `$$.`*attrib* ou sous la forme `$!.`*attrib*. En particulier, lorsqu'on cherche à optimiser un ou plusieurs attributs d'un non-terminal, les attributs non concernés par le critère d'optimisation doivent également être recopiés. C'est le cas dans la contrainte d'optimisation suivante d'un non-terminal `S` possédant les attributs physiques `Energy` et `Type` :

```
%constraint S
{}
{
  if ($$.Energy < $!.Energy)
    {
      $!.Energy = $$.Energy;
      $!.Type = $$.Type;
    }
}
```

L'attribut `Type` associé à `S` dépend de la dérivation, et donc de l'énergie `Energy` optimisée. Il faut le recopier en même temps que `Energy` pour en garder la trace dans la dérivation optimisée. Lorsque certains attributs physiques ne sont pas recopiés lors de l'optimisation, il est impossible de reconstruire l'arbre de dérivation associé à un attribut donné de non-terminal, puisque l'attribut n'est plus indépendant du processus d'analyse syntaxique.

Il existe une exception à cette règle lorsque la contrainte d'optimisation est vide, mais un message d'avertissement est malgré tout affiché pour signaler une contrainte d'optimisation vide lorsqu'un non-terminal possède des attributs physiques. Rappelons qu'une contrainte d'optimisation vide signifie que, pour $a_{\vec{i}} \ldots a_{\vec{j}}$ donné, seule la première dérivation $X \to^* a_{\vec{i}} \ldots a_{\vec{j}}$ d'un non-terminal X trouvée en cours d'analyse sera considérée.

Lorsqu'on souhaite réellement ne pas recopier un attribut physique dans une contrainte et ne pas recevoir de message d'avertissement, il faut utiliser l'attribut physique dans une instruction C vide car MTSAG2C ne sait pas faire la différence entre une instruction C utile et une instruction C inutile. Ainsi, il est possible de réécrire la contrainte précédente de façon à ne pas recevoir de message d'avertissement même si l'attribut physique `Type` n'est pas recopié.

```
%constraint S
{}
{
  if ($$.Energy < $!.Energy)
    {
      $!.Energy = $$.Energy;
      $!.Type;
      $$.Type;
    }
}
```

Notons que `$!.Type` et `$$.Type` doivent tous deux être présents pour supprimer le message d'avertissement.

- Tous les tokens ont la même liste d'attributs physiques. Cette liste est déterminée par MTSAG2C en faisant l'union de tous les attributs physiques utilisés dans les codes C des déclarations `%token`. Pour obtenir la valeur de l'attribut physique *attrib* d'un token, il faut utiliser la pseudo-variable `$?[`*k*`].`*special*`[`*l*`].`*attrib*. Si *attrib* n'appartient pas à la liste des attributs physiques des tokens, une erreur sera signalée lors de la compilation du code C généré par MTSAG2C, et non par MTSAG2C directement.

  Mais *attrib* peut appartenir à la liste des attributs physiques des tokens sans que l'expression `$?[`*k*`].`*special*`[`*l*`].`*attrib* représente un attribut initialisé, puisque *special*`[`*l*`]` peut correspondre à la position d'un token sur la bande k n'ayant pas initialisé *attrib*.

Les vérifications précédentes permettent normalement de signaler une grande partie des incohérences d'écriture des MTSAG. Mais il reste possible d'utiliser une pseudo-variable dans une instruction C inutile et de tromper ainsi MTSAG2C, volontairement ou involontairement.

A.6 Interface entre le langage C et l'analyseur syntaxique

Comme nous l'avons vu dans les exemples de la figure A.1 p.231 et de la figure A.3 p.235, tout analyseur généré par MTSAG2C possède une interface avec le langage C permettant à l'utilisateur de définir aussi bien les bandes à analyser que les opérations à réaliser une fois les arbres de dérivation obtenus.

A.6.1 Analyse lexicale

Comme toujours, l'analyse syntaxique doit être précédée d'une phase d'analyse lexicale. En fait, l'appel au code d'analyse lexicale est à la responsabilité de la fonction d'analyse syntaxique une fois cette dernière appelée par l'utilisateur. L'utilisateur est responsable de la définition d'un analyseur lexical ayant une interface compatible avec celle attendue par l'analyseur syntaxique. Pour faciliter l'écriture des analyseurs lexicaux, cette interface attendue est celle définie par `LEX`.

Il suffit donc d'écrire une description des tokens du type de celle de la figure A.2, puis de fournir cette description à `LEX` pour générer automatiquement un code C prêt à être inclus dans la section *code C supplémentaire* de l'analyseur syntaxique. L'analyseur syntaxique utilise seulement les variables `yyin`, `yytext` et les fonctions `yyrestart` et `yylex` définies par `LEX`.

Dans un fichier `LEX`, les tokens sont décrits sous la forme d'expressions régulières auxquelles sont associées des actions sous la forme de code C. Lorsqu'un token est reconnu, il faut renvoyer le code du token à l'analyseur syntaxique. Pour les tokens spéciaux, le code est simplement le code ASCII du caractère. Pour les autres tokens déclarés à l'aide de `%token`, le code est simplement le nom du token (voir l'analyseur lexical de la figure A.2 associé à l'analyseur syntaxique de la figure A.1).

Si l'utilisateur désire faire reconnaître par l'analyseur lexical d'autres tokens que ceux déclarés avec `%token` dans une MTSAG, alors les codes associés ne doivent pas entrer en conflit avec des codes utilisés par l'analyseur syntaxique. Il est donc conseillé d'utiliser des codes négatifs, associés à des macros définies dans la section *déclarations C* d'une MTSAG. Notons que si une bande contient un token ne correspondant pas à un token déclaré par `%token`, alors ce token doit être absorbé dans le code d'initialisation des attributs physiques du premier token déclaré par `%token` qui le précède. Dans le cas contraire, l'analyseur syntaxique prendra connaissance d'un token qu'il ne sait pas traiter et affichera un message d'erreur. L'exemple de la figure A.5 est une illustration artificielle de ce principe.

La fin d'une bande est indiquée à l'analyseur syntaxique par l'analyseur lexical en retournant la valeur 0. Nous avons choisi dans les exemple précédents de retourner 0 lorsque la fin physique des fichiers décrivant les bandes (soit `<<EOF>>` pour `LEX`) était

```
%%
"+"             { return '+'; }
"*"             { return '*'; }
"("             { return '('; }
")"             { return ')'; }
"N"             { return nombre; }
[0-9]+          { return VALUE; }
<<EOF>>         { return 0; }
.               {}      /* Tout caractère inutile est supprimé */
```

(a) Analyseur lexical modifé. Le token `nombre` est simplement la lettre `N` et les valeurs sont représentées par un token VALUE.

```
%{
#define VALUE -1
%}

%nb_tapes        1
%attribute       "int" valeur
%token           nombre
{
  if (yylex () != VALUE)
    { fprintf (stderr, "erreur de syntaxe\n");   exit (1);  }
  else
    { extern char *yytext; sscanf (yytext, "%i", &$$.valeur); }
}
%token           '(' ')' '+' '*'
%terminal_attribs              { $$.valeur = $$.right[0].valeur; }
%default_nonterminal_attribs   <valeur>
%default_constraint
      {}
      { if ($$.valeur > $!.valeur) $!.valeur = $$.valeur; }

%%
..../* la suite est identique */
```

(b) Analyseur syntaxique modifé. Le token `nombre` (la lettre `N`), une fois reconnu par l'analyseur lexical, attend un token `VALUE` et lit la chaîne de caractères associée à ce token pour connaître l'attribut physique `valeur`. Le reste de la MTSAG est inchangé par rapport à la figure A.1.

Figure A.5: Nous avons modifié la grammaire et l'analyseur lexical des figures A.1 et A.2 pour que chaque nombre des expressions soit précédé de la lettre `N`. Ainsi, la chaîne `8*9` donnée en entrée à la MTSAG d'origine doit ici s'écrire `N8*N9`.

atteinte, mais il n'est pas obligatoire de procéder ainsi, et n'importe quelle expression régulière peut être choisie pour désigner la fin d'une bande.

Il n'y a qu'un seul analyseur lexical pour toutes les bandes d'une MTSAG.

A.6.2 Analyse syntaxique

L'analyse syntaxique est réalisée par la fonction `mtsagparse` dont le prototype est le suivante :

```
int mtsagparse (FILE *input[nb_tapes]);
```

Le paramètre `input` de la fonction est un tableau de pointeurs sur des flux d'entrée tels que `stdin` ou les ceux ouverts grâce à la fonction `fopen`. Le tableau contient autant de pointeurs qu'il y a de bandes (le nombre de bandes est indiqué par la macro `nb_tapes`). Il est possible d'utiliser un même flux pour plusieurs bandes.

Lorsqu'elle est appelée, la fonction `mtsagparse` transforme d'abord la suite de caractères à analyser contenus dans les flux pointés par `input` en une table de bandes, c'est-à-dire une table de tables de tokens (cf. section A.6.4). Il faut savoir que les bandes sont lues en totalité, de la gauche vers la droite, les unes après les autres. Ainsi, le premier token de `input[`i`]` est le premier token de la bande $i+1$ (les bandes sont numérotées de 1 à `nb_tapes` alors que `input` va de 0 à `nb_tapes-1`).

Quand l'analyse syntaxique s'est bien déroulée, c'est-à-dire lorsque `mtsagparse` a pu calculer un attribut pour une dérivation $S \rightarrow^* a_{\vec{1}} \ldots a_{\vec{n}}$, la valeur 0 est retournée, sinon la valeur 1 est retournée. Il est inutile de chercher à construire un arbre de dérivation d'un ensemble de bandes si `mtsagparse` n'a pas retourné 0.

Remarque A.3 Comme les contraintes de sélection peuvent supprimer des dérivations en fonction d'un critère extérieur à la grammaire, il peut exister des chaînes appartenant au langage généré par la MTSAG qui ne sont pas pour autant reconnues par `mtsagparse`.

Remarque A.4 Il est possible d'analyser plusieurs chaînes m-bandes différentes en appelant plusieurs fois `mtsagparse` et `free_parse_alloc` (cf. section A.6.7).

A.6.3 Reconstruction des arbres de dérivation

Si l'on ne tient pas compte des contraintes de sélection, le but de la fonction `mtsagparse` est de calculer, pour toute dérivation $A \rightarrow^* a_{\vec{i}+\vec{1}} \ldots a_{\vec{j}}$, un attribut optimal x de A, associé à un ensemble de dérivations optimales de A en $a_{\vec{i}+\vec{1}} \ldots a_{\vec{j}}$. L'attribut optimal est en réalité formé de un ou plusieurs attributs physiques.

Une fois l'analyse syntaxique effectuée, la fonction `build_parse_tree` peut être appelée pour constuire un des arbres de dérivation optimaux sélectionnés par les contraintes de sélection et d'optimisation. Le protoype de cette fonction est le suivant,:

```
parse_tree_t *build_parse_tree (int nonterminal, int left_pos, int right_pos)
```

Le paramètre `nonterminal` est le code du non-terminal A du paragraphe précédent (cf. section A.6.5), `left_pos` est le code associé à $\vec{\imath}$, et `right_pos` est le code associé à $\vec{\jmath}$ (cf. section 4.1.3 pour une explication du code des index m-bandes). La valeur retournée par la fonction est un pointeur de type `parse_tree_t *` sur la racine d'un sous-arbre de dérivation d'attribut optimal correspondant aux paramètres.

Le code du non-terminal de départ est indiqué par une macro spéciale dont la syntaxe est `START_NONTERMINAL` quelle que soit la MTSAG (cf. section A.6.5). De même, il existe une macro spéciale `COMPOSITE_LEN` pour obtenir le code de l'index m-bandes $\vec{n}$ tel que $a_{\vec{1}} \dots a_{\vec{n}}$ est la chaîne m-bandes analysée. Pour obtenir un arbre de dérivation optimal de la chaîne m-bandes analysée, il suffit donc d'écrire (cf. figures A.1 et A.3) :

```
tree = build_parse_tree (START_NONTERMINAL, 0, COMPOSITE_LEN);
```

Chaque sommet d'un arbre de dérivation correspond à un symbole X d'une dérivation. Les informations relatives à chaque sommet sont stockées dans la structure `parse_tree_t` suivante :

```
typedef struct parse_tree_t
{
  int symbol;                          /* symbole X du sommet */
  int left_pos, *decoded_left_pos;
  int right_pos, *decoded_right_pos;
  attrib_t *attrib;                    /* attributs du symbole */
  tape_token_t *token[nb_tapes];       /* lorsque X est un terminal, pointeurs
                                          sur les tokens de chaque bande */
  int print_char[nb_tapes];            /* lorsque X est un terminal, caractère
                                          associé à chaque token */
  int nb_subtrees;                     /* nombre de sous-arbres du sommet */
  struct parse_tree_t **subtree;       /* table de pointeurs sur les sous-arbres
                                          du sommet */
  struct parse_tree_t *parent;         /* sommet parent du sommet actuel */
  int subtree_pos;
} parse_tree_t;
```

Les champs `left_pos` et `right_pos` correspondent respectivement aux codes de deux index m-bandes $\vec{\imath}$ et $\vec{\jmath}$ tels que la sous-chaîne m-bandes associée au sommet est $a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$.

Le champ `decoded_left_pos` est un tableau de `nb_tapes` entiers correspondant aux composantes sur chaque bande de l'index m-bandes $\vec{\imath}$ associé à `left_pos`. Autrement dit, `decoded_left_pos[`$k-1$`]` est égal à $\vec{\imath}^{(k)}$. Le champ `decoded_right_pos` joue un rôle semblable pour `right_pos`.

Dans tous les cas, `attrib` contient un pointeur sur les attributs physiques du symbole associé au sommet (cf. section A.6.6 pour un exposé des macros permettant d'accéder aux attributs physiques à partir d'un pointeur).

Lorsque `nb_subtrees` est nul, le sommet n'a pas de sous-arbre, soit parce que X est un terminal, auquel cas les champs `token` et `print_char` sont significatifs, soit parce que X est un non-terminal A pour lequel `build_parse_tree` n'a pas pu trouver de

```
void exploration (parse_tree_t *p)
{
  while (1)
    {
      /* Insérer ici le code du traitement de p, pour un parcours préfixe */

      if (p->nb_subtrees)
        {
          p->subtree_pos = 0;
          p = p->subtree[0];
        }
      else if (p->parent)
        {
          while (p->parent)
            {
              parse_tree_t *q = p->parent;

              if (q->subtree_pos != q->nb_subtrees - 1)
                {
                  q->subtree_pos++;
                  p = q->subtree[q->subtree_pos];
                  break;
                }
              else
                p = q;
            }
        }

      /* Insérer ici le code du traitement de p, pour un parcours postfixe */

      if (p->parent == NULL)
        break;
    }
}
```

Figure A.6: Exemple de code non-récursif d'exploration d'un arbre de dérivation utilisant les champs `parent` et `subtree_pos` pour réaliser l'exploration.

dérivation ayant l'attribut optimal calculé par `mtsagparse`. Un tel échec signifie soit que le calcul des attributs physiques dans la grammaire est incorrect (cf. section A.5), soit que la contrainte $\mathcal{C}_A$ associée au non-terminal A du sommet ne vérifie pas toujours la relation $x = \mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x, x)$ pour un attribut optimal x.

Lorsque `nb_subtrees` est non nul, les sommets des sous-arbres du sommet actuel sont stockés séquentiellement dans le tableau `subtree`. Grâce aux champs `parent` et `subtree_pos`, il est possible d'écrire un code C non récursif d'exploration d'un arbre de dérivation (figure A.6). Un tel code peut se révéler utile car il est courant d'obtenir des arbres de dérivation d'une hauteur de plusieurs milliers. Un parcours récursif de l'arbre pourrait alors entraîner un dépassement de capacité de la pile sur certaines architectures trop optimistes quant à la taille de la pile nécessaire (PC notamment).

Lorsque le symbole X du sommet est un terminal $a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$, c'est-à-dire lorsque $\vec{\jmath} - \vec{\imath}$ est un index m-bande non nul ne contenant que des 0 ou des 1, le tableau `token` contient les pointeurs sur les tokens associés à ce terminal. Autrement dit, `token[`$k-1$`]` pointe sur le token $a^{(k)}_{\vec{\jmath}^{(k)}}$ si $\vec{\jmath}^{(k)} - \vec{\imath}^{(k)} = 1$, ou sur le token vide si $\vec{\jmath}^{(k)} - \vec{\imath}^{(k)} = 0$ (cf. p.262 pour la définition du type `tape_token_t` utilisé pour la déclaration du champ `token`).

Lorsque le symbole X du sommet est un terminal, `print_char[`$k-1$`]` est égal au code ASCII du *caractère indicatif* du token situé sur la $k^{\text{ème}}$ bande (p.251).

Remarque A.5 La fonction `build_parse_tree` construit le premier arbre tel que l'attribut à la racine de cet arbre, calculé de bas-en haut à partir des attributs des terminaux aux feuilles de l'arbres, est le même que celui calculé par `mtsagparse`. Mais un tel arbre peut être impossible à reconstruire s'il existe une contrainte d'optimisation $\mathcal{C}_A$ telle que $x \neq \mathcal{C}_A(\vec{\imath}, \vec{\jmath}, x, x)$ pour un attribut optimal x d'une dérivation $A \rightarrow^* a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$. Dans un tel cas, l'utilisateur doit gérer lui-même la création de cet arbre, à l'aide d'attributs physiques spéciaux pointant sur des sous-arbres. C'est d'ailleurs cette méthode qui est habituellement utilisée avec YACC lorsqu'un arbre de dérivation explicite est souhaité.

A.6.4 Stockage des tokens

Chaque token lu sur une bande est stocké dans une structure `tape_token_t` :

```
typedef struct tape_token_t
{
  char *syntax;
  int lex_code, mtsag_code;
  ....                         /* attributs physiques des tokens */
} tape_token_t;
```

Le champ `syntax` correspond à la suite de caractères lus sur la bande par l'analyseur lexical pour reconnaître le token.

Le champ `lex_code` est le code retourné par l'analyseur lexical pour le token, alors que `mtsag_code` est le code utilisé par `mtsagparse` pour ses opérations internes. Ces deux codes sont différents parce que les tokens spéciaux de la forme `'`*caractère*`'` ont un code d'analyse lexicale égal à leur code ASCII, alors que les autres tokens, comme `nombre` dans la figure A.1, ont un code défini par une macro identique à leur nom. Voir la figure A.1 p. 231 pour un exemple d'utilisation de `lex_code`.

Les autres champs de `tape_token_t` sont les attributs physiques des tokens déclarés et utilisés dans les codes C des déclarations `%token`.

Les tokens de chaque bande sont stockés dans une table `input_tape`. L'expression `input_tape[`$k-1$`][`i`]` représente la structure `tape_token_t` du $i^{\text{ème}}$ token de la bande k. Le premier token de la bande k porte le numéro 1 et le dernier token porte le numéro indiqué par `input_tape_len[`$k-1$`]`. La table `input_tape_len` contient donc les longueurs des bandes. Les tokens trouvés en `input_tape[`k`][0]` et `input_tape[`k`][input_tape_len[`k`]+1]` sont deux tokens vides dont les valeurs de `lex_code` et `mtsag_code` sont toutes les deux nulles et dont les attributs physiques ne sont pas initialisés.

Lorsque les tokens de la bande k sont lus, l'expression `tape` (resp. `tape_len`) est un alias de `input_tape[`$k-1$`]` (resp. `input_tape_len[`$k-1$`]`) utilisable dans le code C des déclarations `%token`. Voir la figure 3.15 p. 125 pour un exemple d'utilisation de `tape` et `tape_len`.

A.6.5 Noms des tokens et des symboles de la grammaire

Lorsque le token vide est déclaré à l'aide de `%empty` *nom*, la macro donnant accès au code du token vide dans les portions de code C d'une MTSAG est précisément *nom*. Lorque la déclaration `%empty` n'apparaît pas dans une MTSAG, MTSAG2C utilise une macro par défaut appelée `__empty_token__`. Dans tous les cas, le code du token vide est 0.

Les tokens "normaux" (par opposition aux tokens spéciaux, formés d'un caractère entre guillemets simples) sont représentés dans le code C par une macro portant le nom du token. Les tokens spéciaux sont représentés dans le code C par une macro `CHAR_`i, où i est le code ASCII du token écrit en décimal. Cependant, pour faciliter l'écriture des analyseurs lexicaux, il est possible de retourner directement le code ASCII du token spécial lorsqu'il a été reconnu sur une bande.

Si $T_1, \ldots, T_m$ sont les macros de m tokens $a_1, \ldots, a_m$, alors le code du terminal m-bandes `[`$a_1 \ldots a_m$`]` est représenté par une macro `_`T_1`_`$T_2 \ldots$ `_`T_m.

Le code d'un non-terminal est représenté dans le code C par une macro portant son nom. De plus, le code du non-terminal de départ est obtenu grâce à la macro `START_NONTERMINAL`.

A.6.6 Attributs physiques des symboles

Nous avons déjà vu dans la section A.6.4 que les attributs physiques des tokens sont stockés dans une structure `tape_token_t`, et que cette structure `tape_token_t` est accessible pour tout token $a_i^{(k)}$ à l'aide de l'expression `input_tape[`$k-1$`][`i`]`.

La situation est passablement plus complexe pour les terminaux et non-terminaux d'une MTSAG, car un type différent est créé pour chaque groupe de symboles utilisant les mêmes attributs physiques, et tous ces types sont rassemblés dans une même union dont le type est `attrib_t` (cf. section 4.3 p.192). Pour faciliter l'accès aux attributs physiques d'un symbole quel que soit ce symbole, des macros ont été créées. Si *name* est un attribut physique déclaré à l'aide de `%attribute`, et si `p` est un pointeur sur la stucture stockant les attributs physiques d'un symbole X, alors la macro `ATTRIB_`*name*`(p)` retourne la valeur de *name* pour la structure pointée par `p` associée à X.

Les pointeurs sur les structures de stockage des attributs sont typiquement obtenus grâce au champ `attrib` de `parse_tree_t` (p.260) après construction d'un arbre de dérivation, et grâce à la fonction `retrieve_attrib` dont le prototype est le suivant :

```
attrib_t *retrieve_attrib (attrib_t *attrib, int symbol_no,
                           int left_pos, int right_pos);
```

Le paramètre `symbol_no` est le code du symbole X (cf. section A.6.5) dont on souhaite connaître un attribut optimal pour toutes les dérivations $X \to^* a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$, après analyse faite par `mtsagparse`. Le code de $\vec{\imath}$ correspond au paramètre `left_pos` et celui de $\vec{\jmath}$ au paramètre `right_pos`.

Si le symbole X ne possède aucune dérivation en $a_{\vec{\imath}+\vec{1}} \dots a_{\vec{\jmath}}$, alors la fonction `retrieve_attrib` retourne un pointeur `NULL`. S'il existe une telle dérivation, alors il existe une dérivation optimale et `retrieve_attrib` retourne un pointeur sur une structure `attrib_t` correspondante, et recopie également le résultat dans la structure pointée par le premier paramètre `attrib`, si ce paramètre n'est pas le pointeur `NULL`.

A.6.7 Macros et fonctions diverses

`nb_tapes` nombre de bandes ;

`nb_tokens` nombre de tokens ;

`nb_terminals` nombre de terminaux ;

`nb_symbols` nombre total de symboles ;

`IS_TERMINAL(X)` retourne 1 si X est un terminal, 0 sinon ;

`START_NONTERMINAL` code du non-terminal de départ. Utilisable avec les fonctions `retrieve_attrib` et `build_parse_tree` ;

`nb_rules` nombre de règles, après expansion des tokens abrégés ;

`COMPOSITE_LEN` code de l'index m-bandes $\vec{n}$ d'une chaîne d'entrée m-bandes $a_{\vec{1}} \dots a_{\vec{n}}$. Utilisable comme valeur possible du paramètre `right_pos` des fonctions utilisateur `retrieve_attrib` et `build_parse_tree` ;

`free_parse_alloc` En fin d'analyse, libère la mémoire allouée par `mtsagparse` pour préparer une autre analyse. Une fois la mémoire libérée, il est impossible d'appeler les fonctions `build_parse_tree` et `retrieve_attrib`. Les arbres de dérivation déjà construits par `build_parse_tree` ne sont pas libérés.

A.7 Ligne de commande MTSAG2C

Une invocation de MTSAG2C a la forme suivante :

`mtsag2c` *options grammaire* `-o` *analyseur*`.c`

Elle signifie que MTSAG2C va lire la grammaire *grammaire* et utiliser les *options* pour générer un fichier C *analyseur*`.c`. Les options de MTSAG2C sont les suivantes :

-dynamic force MTSAG2C à ne pas générer statiquement les tables d'analyse et à inclure dans *analyseur.c* les fonctions nécessaires à la génération dynamique de ces tables d'analyse (cf. section 4.2.2 p.189 pour une explication de la méthode utilisée pour générer dynamiquement les états). La génération dynamique des tables ralentit légèrement le code généré et doit être évitée, sauf si la combinatoire du problème empêche une génération statique des tables.

-space force MTSAG2C à générer un code C plus clair contenant de nombreux commentaires facilitant sa lecture en cas de problème. Lorsque la génération des états est statique, cette option permet en particulier de savoir quelle est la forme des états $[\Delta \rightarrowtail \alpha]$ générés par MTSAG2C.

-stat xxx modifie le code généré de façon à faire afficher automatiquement par `mtsagparse` un certain nombre d'informations de nature statistique liées à l'analyse. La nature des informations affichées dépend du paramètre `xxx` donné à `-stat`.

Toutes les options précédentes peuvent apparaître dans une déclaration `%mode` de la MTSAG, et en particulier l'option `-dynamic` qui doit absolument être utilisée avec certaines MTSAG.

Plusieurs options `-stat` peuvent apparaître sur la ligne de commande de MTSAG2C, le paramètre `xxx` de chaque option pouvant prendre une des valeurs suivantes :

alloc Affiche le nombre d'octets occupés par diverses structures du programme. Le champ le plus intéressant est `attributes` car il représente la place occupée par les attributs physiques de tous les non-terminaux réduits en cours d'analyse. Il s'agit en fait de la place prise par la table $(\mathfrak{T}_{\vec{\imath},\vec{\jmath}})_{1\leqslant\vec{\imath}<\vec{\jmath}\leqslant n}$ (cf. section 2.2.1 p.81). Voici un exemple d'affichage :

```
Final memory allocation :
total allocation        =  13970040 100.00 %
tape                    =     69884   0.50 %
attributes              =  11553244  82.70 %
items                   =    326720   2.34 %
influence               =    930924   6.66 %
chains of attributes    =    244980   1.75 %
queue                   =     16580   0.12 %
entries                 =    437128   3.13 %
Check_Symbol            =    314776   2.25 %
parsing tables          =     75804   0.54 %
```

Si l'option `-stat progress` a été utilisée, alors l'allocation est affichée tous les cinquante terminaux lus par `mtsagparse`.

item Affiche le nombre total d'états utilisés et alloués en cours d'analyse syntaxique, puis la proportion des utilisations et des allocations pour chaque état. En fin de table, la proportion d'états utilisés par rapport au nombre d'états générés est affichée (c'est un indicateur de l'efficacité de la génération des états et d'un éventuel intérêt d'une option `-dynamic`). La dernière ligne est liée à la taille de la table $T^{\cup}$ (cf. p.183 pour une définition de cette table). Si le nombre d'ensembles utilisés est très faible par rapport au nombre total d'ensembles, alors une option `-dynamic` ou une réécriture de la grammaire peuvent se révéler utile.

```
Total number of used items: 48358
Total number of allocated items: 4176
                      used              allocated
   item    0 :          0    0.00 %          0    0.00 %
   item    1 :         96    0.20 %         96    2.30 %
   item    2 :         96    0.20 %          0    0.00 %
   item    3 :          0    0.00 %          0    0.00 %
   item    4 :          0    0.00 %          0    0.00 %
   item    5 :        167    0.35 %        167    4.00 %
   item    6 :        167    0.35 %          0    0.00 %
   item    7 :          0    0.00 %          0    0.00 %
   item    8 :        111    0.23 %        111    2.66 %
   .
   .
   .
   item   71 :       2451    5.07 %          0    0.00 %
   item   72 :        391    0.81 %          0    0.00 %
   item   73 :        740    1.53 %          0    0.00 %
   item   74 :        628    1.30 %          0    0.00 %
   item   75 :        624    1.29 %          0    0.00 %
   item   76 :       2701    5.59 %          0    0.00 %
   item   77 :        436    0.90 %          0    0.00 %
   item   78 :        798    1.65 %          0    0.00 %
   item   79 :        707    1.46 %          0    0.00 %
   item   80 :        687    1.42 %          0    0.00 %
   item   81 :      24094   49.82 %          0    0.00 %
Percentage of item numbers used :  72/91   79.12 %
Total number of sets : 5   Percentage of sets used : 3  60.00 %
```

Les états utilisés mais non alloués sont en fait des états uniquement réductibles, qui ne sont donc pas alloués par l'analyseur (cf. section 4.5.1 p.201).

attrib Permet de déterminer l'importance relative des différents non-terminaux dans l'allocation des attributs, que ce soit en nombre d'allocations ou en taille d'allocations, car tous les non-terminaux n'ont pas les mêmes attributs.

```
Memory allocation for attributes of nonterminals :
                       Size in bytes     Number of allocations
        total :      109272 / 100.00 %          8057 / 100.00 %
  symbol    5 :       14040 /  12.85 %          1755 /  21.78 %
  symbol    6 :       12900 /  11.81 %          1075 /  13.34 %
  symbol    7 :       49020 /  44.86 %          2451 /  30.42 %
  symbol    8 :         900 /   0.82 %            75 /   0.93 %
  symbol    9 :       32412 /  29.66 %          2701 /  33.52 %
```

Les numéros des symboles affichés correspondent aux entiers retournés par les macros portant le nom de chaque non-terminal (cf. section A.6.5). En phase d'optimisation d'une MTSAG, il faut prioritairement traiter les non-terminaux dont les attributs physiques occupent la plus grande partie de la mémoire allouée. Parmi les optimisations possibles, il faut compter le remplacement des règles récursives à droite par d'autres règles non-récursives à droite, ou l'introduction de contraintes de sélection supprimant toutes les dérivations qui s'écartent trop des valeurs attendues (cf. section 3.2 p.93).

useful Permet de connaître l'efficacité de l'analyseur avec la grammaire donnée. Un algorithme d'analyse syntaxique est d'autant plus efficace qu'il génère moins d'états inutiles et fait moins de réductions inutiles. Dans notre cas, les réductions inutiles se traduisent par des calculs inutiles d'attributs de non-terminaux. Nous utilisons ici la notion d'inutilité a posteriori définie p.23.

En fin d'analyse, une routine parcourt la forêt d'arbres de dérivation et note l'ensemble des non-terminaux connectés à la racine de la forêt. Elle affiche ensuite le résultat de ce parcours sous forme d'une table donnant l'utilité globale des non-terminaux, puis une décomposition par non-terminal.

```
Usefulness of reduced nonterminals
                   useful / allocated     % useful   % useful/total useful
  total       :      7666 /    8057         95.15 %
  symbol    5 :      1594 /    1755         90.83 %     20.79 %
  symbol    6 :      1075 /    1075        100.00 %     14.02 %
  symbol    7 :      2444 /    2451         99.71 %     31.88 %
  symbol    8 :        75 /      75        100.00 %      0.98 %
  symbol    9 :      2478 /    2701         91.74 %     32.32 %
```

Glossaire des termes biologiques

ADN noté DNA en anglais.

$\Longrightarrow$ désoxyribonucléique (acide).

alignement rapprochement visuel de deux ou plusieurs structures ayant pour but de faciliter la mise en évidence des ressemblances ou différences significatives d'une structure à l'autre.

Dans le cas des séquences, disposition en lignes (contenant éventuellement des gaps) des structures primaires de deux ou plusieurs molécules (protéines, ADN, ARN).

$\Longrightarrow$ sec 3.1 p. 86, 3.3 p. 95, et 3.1 p. 86.

alignement global recherche d'un alignement de l'intégralité des séquences initiales dont la ressemblance est maximale.

$\Longrightarrow$ **alignement local, alignement multiple**; fig. 3.2 p. 89.

alignement local recherche d'un alignement de sous-séquences des séquences initiales dont la ressemblance est maximale.

$\Longrightarrow$ **alignement global, alignement multiple**; p. 88; (Smith & Waterman, 1981).

alignement multiple alignement d'au moins trois molécules.

$\Longrightarrow$ **alignement global, alignement local**; p. 87.

appariement association par liaisons hydrogènes de deux bases d'un acide nucléique ; cette association spécifique a lieu entre une base purique et une base pyrimidique, dites bases complémentaires. Les appariements habituels selon Watson et Crick sont AT (AU dans les ARN) et CG. L'appariement GU est également fréquent dans les ARN.

$\Longrightarrow$ fig. 1.2 p. 9.

ARN noté RNA en anglais.

$\Longrightarrow$ **ribonucléique (acide)**.

base en génétique, terme employé pour base nucléique (base purique ou base pyrimidique).

brin chaîne linéaire non ramifiée d'un acide nucléique.

$\Longrightarrow$ **double-brin, simple-brin**.

CM de Covariance Model, pour modèle de covariance.

$\Longrightarrow$ p. 282, sec 3.10 p. 156.

codon triplet de nucléotides désigné par les initiales des trois bases respectives et dont l'ordre séquentiel constitue l'information utilisée lors du processus de traduction.

consensus (colonne) colonne d'un alignement multiple pour laquelle la proportion de séquences y possédant une base est très supérieure à la proportion de séquences y possédant un gap.

$\Longrightarrow$ p. 135

consensus (structure) structure secondaire dont les éléments sont communs à une majorité de structures secondaires d'une famille de protéines ou d'ARN.

$\Longrightarrow$ p. 65, 97, 110, 147, 156, 170

désoxyribonucléique (acide) macromolécule formée par un enchaînement de désoxyribonucléotides et qui constitue le matériel génétique de toutes les cellules eucaryotes (noyau, mitochondries, chloroplastes), des cellules procaryotes (bactéries) et de certains virus ; elle est donc le support essentiel de l'hérédité.

$\Longrightarrow$ **ribonucléique (acide)**.

désoxyribonucléotide unité élémentaire constitutive des acides ribonucléiques, formée par la liaison d'un sucre désoxyribose, d'un acide orthophosphorique et d'une base purique ou pyrimidique.

$\Longrightarrow$ **ribonucléotide**.

double-brin (ADN) forme la plus courante de la molécule d'ADN constituée de deux brins : les séquences nucléotidiques de ces deux brins sont complémentaires.

$\Longrightarrow$ **brin, simple-brin**.

enzyme substance protéique qui accélère une réaction biochimique. L'Académie Française préfère dire "une enzyme", mais le vocable "un enzyme" est également accepté.

gap dans un alignement, insertion d'un ou plusieurs trous entre deux nucléotides ou acides aminés afin de maximiser le score de l'alignement.

homologie état d'entités ayant même fonction et même structure. Degré d'identité dans la succession des monomères existant entre séquences de molécules d'acides nucléiques ou séquences de protéines.

homologue qui à trait à l'homologie. Plus strictement, deux séquences sont dites homologues lorsqu'elles ont un ancêtre commun et des fonctions communes.

messager (ARN) forme particulière d'ARN ayant pour rôle de transporter l'information génétique depuis l'ADN jusqu'au lieu approprié de la synthèse des protéines. Les ARN messagers sont également notés ARNm (ou mRNA en anglais).

$\Longrightarrow$ **ribosomal (ARN), transfert (ARN)**.

modèle stochastique modèle simplifié de calcul de la structure secondaire d'un ARN fondé sur les grammaires stochastiques.

$\Longrightarrow$ **modèle thermodynamique**; sec. 1.7.3 p. 64 et 3.9 p. 140.

modèle thermodynamique modèle simplifié de calcul de l'énergie libre d'un ARN opérant par décomposition en boucles de la structure secondaire de l'ARN et sommation de l'énergie libre attribuée à chaque boucle.

$\Longrightarrow$ **modèle stochastique**; sec. 1.7.2.1 p. 51.

mutation compensatoire dans une paire de bases, mutation d'une des bases induite par la mutation de l'autre afin de maintenir l'appariement.

nœud $\Longrightarrow$ **pseudo-nœud**.

paire de bases $\Longrightarrow$ **appariement**.

phase de lecture structure séquentielle de l'ARN messager débutant par un codon d'initiation de la traduction. A partir de ce codon, le ribosome parcourt l'ARNm en phase, de triplet en triplet, jusqu'aux codons de terminaison. On parle également de cadre de lecture.

$\Longrightarrow$ fig. 3.6 p. 92.

phylogénèse reconstitution hypothétique de l'évolution des espèces et autres taxons, qui les ferait dévier les uns des autres depuis les organismes les plus simples jusqu'aux plus complexes selon la taxinomie actuelle.

phylogénétique branche de la génétique qui étudie les modifications d'ordre génétique qui se produisent au sein des espèces (animales, végétales). Qui à rapport à la phylogénèse.

protéine corps appartenant à la classe des macromolécules organiques azotées, de poids moléculaire très élevé (de 10000 à plus de 500000), qui entre pour une forte proportion dans la constitution des êtres vivants et libère par hydrolyse des acides aminés (aminoacides).

pseudo-nœud une structure secondaire forme un pseudo-nœud lorsqu'il existe deux paires de bases (i, k) et (j, l) de la structure vérifiant $i < j < k < l$, où i, j, k, l sont les positions respectives sur la structure primaire des bases des deux paires.

$\Longrightarrow$ fig. 1.4 p. 10; sec. 3.13 p. 171.

purique (base) base hétérocyclique dérivée du noyau purine : il s'agit de l'adénine (notée A) et de la guanine (notée G) présentes dans l'ADN et l'ARN.

$\Longrightarrow$ **pyrimidique (base)**.

pyrimidique (base) base hétérocyclique dérivée du noyau pyrimidine : il s'agit de la cytosine (notée C) présente dans l'ADN et l'ARN, de la thymine (notée T) présente dans l'ADN, et de l'uracile (notée U) présent dans l'ARN.

$\Longrightarrow$ **purique (base)**.

repliement $\Longrightarrow$ **replier**.

replier action de rapprocher des éléments distants d'une molécule chimique afin de former des liaisons supplémentaires propres à diminuer l'énergie libre totale de la molécule d'une part, et à placer la molécule dans une configuration nécessaire à l'exécution correcte de ses fonctions biochimiques d'autre part.

ribonucléique (acide) macromolécule formée par un enchaînement de ribonucléotides résultant de la transcription de l'ADN, présente dans les cellules eucaryotes, procaryotes, ainsi que dans certains virus.

$\Longrightarrow$ **désoxyribonucléique (acide)**.

ribonucléotide unité élémentaire constitutive des acides ribonucléiques, formée par la liaison d'un sucre ribose, d'un acide orthophosphorique et d'une base purique ou pyrimidique.

$\Longrightarrow$ **désoxyribonucléotide**.

ribosomal (ARN) un ARN est ribosomal (on dit aussi ribosomique) lorsqu'il entre dans la composition d'un ribosome. Les ARN 5S, 16S, 28S sont trois grandes familles d'ARN ribosomaux. Les ARN ribosomaux sont également notés ARNr (ou rRNA en anglais).

$\Longrightarrow$ **messager (ARN), traduction, transfert (ARN)**.

séquence pour un polymère, suite orientée des constituants monomériques. Par exemple, pour un acide nucléique, on parle de séquence nucléotidique; pour une protéine ou un polypeptide, de séquence peptidique.

$\Longrightarrow$ **structure primaire**.

simple-brin (ADN ou ARN) forme la plus courante de la molécule d'ARN constituée d'un seul brin. Forme également recontrée dans l'ADN, localement ou chez certains virus.

$\Longrightarrow$ **brin, double-brin**.

structure la structure désigne la disposition des parties d'un ensemble abstrait, d'un phénomène, d'un système complexe, généralement envisagée comme caractéristique de cet ensemble et comme durable. La structure primaire d'une molécule (protéine, ADN, ARN) désigne la représentation sous forme de chaîne de caractères de la suite de ses composants élémentaires (acides aminés, acides ribonucléiques, acides désoxyribonucléiques) vus comme lettres d'un alphabet. La structure tertiaire de cette même molécule désigne l'agencement particulier dans l'espace de ses composants élémentaires, afin de permettre à la molécule d'assurer sa fonction. Enfin, sa structure secondaire est une représentation planaire des appariements constatés dans la structure tertiaire.

$\Longrightarrow$ fig. 1.3 p. 10.

structure primaire $\Longrightarrow$ **structure**; def. 1.18 p. 51.

structure secondaire $\Longrightarrow$ **structure**; def. 1.19 p. 51.

structure tertiaire $\Longrightarrow$ **structure**.

taxon unité systématique (telle qu'une famille, un genre, une espèce).

traduction processus qui, à partir d'une chaîne d'ARNm, de ribosomes et d'ARN de transfert chargés d'acides aminés, synthétise une chaîne polypeptidique dont la séquence correspond à celle des codons de l'ARN messager.

$\Longrightarrow$ **transcription**.

transcription processus qui, à partir d'une matrice d'ADN, de ribonucléotides, synthétise l'ARN à l'aide de l'ARN polymérase et d'autres enzymes appropriées. L'ARN synthétisé est simple brin et complémentaire du brin codant de l'ADN.

$\Longrightarrow$ **traduction**.

transfert (ARN) ARN capable de se charger d'un acide aminé spécifique et de le placer à l'extrémité d'une chaîne polypeptidique en formation sur le ribosome, grâce à l'interaction de son anticodon et du codon correspondant de l'ARNm. Cette molécule permet donc l'adaptation du message à quatre bases des acides nucléiques à celui à vingt acides aminés des protéines. Les ARN de transfert sont également notés ARNt (ou tRNA en anglais).

$\Longrightarrow$ **messager (ARN), traduction, ribosomal (ARN)**.

Glossaire des termes informatiques

algorithme CYK algorithme tabulaire de Cocke–Younger–Kasami.

$\Longrightarrow$ **tabulaire**; p. 22, 67, 207.

algorithme de Earley algorithme tabulaire de Earley, très populaire en analyse du langage naturel du fait de sa simplicité et de ses bonnes propriétés. Il en existe de nombreuses versions améliorées.

$\Longrightarrow$ **tabulaire**; p. 22; sec. 1.3.2 p. 23 et 5.1 p. 205; thm. 1.4 p. 37;

algorithme GCP (Generalized Common Prefix) algorithme dérivé de l'algorithme de Earley et reposant sur une factorisation des préfixes gauches communs des règles d'une grammaire.

$\Longrightarrow$ p. 28, 32, 44, 80, 180, 202.

ambiguïté une grammaire non-contextuelle G est dite ambiguë s'il existe un mot de $L(G)$ possédant deux arbres de dérivation différents. Si un tel mot n'existe pas, la grammaire est dite non-ambiguë.

Un langage L est non-ambigu s'il existe une grammaire non-contextuelle non-ambiguë G telle que $L = L(G)$. Dans le cas contraire, le langage L est dit fondamentalement ambigu.

Une grammaire est à ambiguïté exponentielle lorsque le nombre d'arbres de dérivation d'un mot du langage croît en moyenne exponentiellement avec la longueur de ce mot.

$\Longrightarrow$ **grammaire**; rem. 1.3 p. 18; sec. 1.3.1 p. 21; fig. 1.1 p. 38 et 1.2 p. 48.

analyse lexicale lecture du flux de caractères présentés en entrée lors d'une analyse syntaxique et regroupement de ces caractères en lexèmes (ou tokens) qui seront ensuite fournis à l'analyseur syntaxique.

$\Longrightarrow$ **analyseur lexical**.

analyse syntaxique détermination explicite de la structure syntaxique d'une ou plusieurs chaînes de lexèmes ou de tokens dans un langage donné.

$\Longrightarrow$ **analyseur syntaxique, reconnaissance**.

analyse syntaxique bottom-up stratégie d'analyse syntaxique reposant sur la construction d'un arbre de dérivation en partant des feuilles et en tentant de reconnecter tous les sous-arbres déjà construits jusqu'à obtenir un arbre complet.

L'algorithme CYK est purement bottom-up. L'algorithme GCP est un algorithme essentiellement bottom-up, mais avec un contrôle top-down. Les théories modernes tendent à faire disparaître la frontière entre bottom-up et top-down (Schöbel-Theuer, 1996).

$\Longrightarrow$ **algorithme CYK, algorithme GCP, analyse syntaxique bottom-up, LR(k)**

analyse syntaxique top-down stratégie d'analyse syntaxique reposant sur la construction d'un arbre de dérivation en partant de la racine (non-terminal de départ) et en dérivant successivement tous les non-terminaux restants jusqu'à obtenir une chaîne de terminaux correspondant à la chaîne analysée.

$\Longrightarrow$ **analyse syntaxique bottom-up, LL(k)**.

analyseur lexical algorithme ou programme chargé de l'analyse lexicale.

$\Longrightarrow$ **analyse lexicale**.

reconnaissance détermination implicite de la structure syntaxique d'une ou plusieurs chaînes de lexèmes ou de tokens dans un langage donné.

$\Longrightarrow$ **analyse syntaxique**.

analyseur syntaxique algorithme ou programme chargé de l'analyse syntaxique. Un analyseur syntaxique prend généralement en entrée une ou plusieurs suites de lexèmes retournés par l'analyseur lexical. Il produit un arbre de dérivation ou une forêt de dérivation. Par extension, un analyseur syntaxique désigne tout algorithme ou programme déterminant implicitement ou explicitement la structure syntaxique d'une ou plusieurs chaînes de lexèmes.

$\Longrightarrow$ **analyse syntaxique**.

arbre de dérivation dans le cas d'une grammaire non-contextuelle, représentation graphique planaire, implicite ou explicite, sous forme d'arbre ordonné d'une série de dérivations du non-terminal de départ S en une chaîne de V^*. Le non-terminal S est la racine de l'arbre et les symboles de la chaîne dérivée sont les feuilles de l'arbre.

$\Longrightarrow$ **arbre ordonné, forêt de dérivation, grammaire**; fig. 1.5 p. 15.

arbre ordonné graphe orienté dont les nœuds vérifient les propriétés suivantes :

- chaque nœud a un exactement un parent, sauf un unique nœud n'ayant aucun parent et appelé la racine de l'arbre,
- tous les nœuds ont un nombre quelconque de fils, l'ordre de ces fils étant significatif. Les nœuds sans fils sont appelés les feuilles de l'arbre.

$\Longrightarrow$ **arbre de dérivation**.

attributs $\Longrightarrow$ **grammaire attribuée**.

attributs hérités $\Longrightarrow$ **grammaire attribuée**.

attributs synthétisés $\Longrightarrow$ **grammaire attribuée**.

automate représentation formelle d'un calcul par le biais d'un ensemble d'états (généralement fini), d'une ou plusieurs bandes orientées contenant des symboles, et d'une ou plusieurs têtes de lecture et/ou écriture sur chacune de ces bandes. Certaines bandes contenant des données de départ sont explicitement désignées comme étant les bandes d'entrée (input tapes) ou, plus simplement, l'entrée (input). Il existe toujours au moins une bande d'entrée. Certaines bandes contenant les données calculées sont explicitement désignées comme étant les bandes de sortie (output tapes) ou, plus simplement, la sortie (output). Il peut n'y avoir aucune bande de sortie, ou certaines bandes d'entrée peuvent également être des bandes de sortie.

Il existe toujours un état particulier désigné comme étant l'état de départ de l'automate et plusieurs états désignés comme étant des états acceptants de l'automate. L'état de départ peut également être acceptant. À un instant donné, un automate ne peut se trouver que dans un seul état.

Une action d'un automate peut être un changement d'état, le déplacement d'une ou plusieurs têtes de lecture et/ou écriture, l'écriture d'un ou plusieurs symboles sur une ou plusieurs bandes de sortie. L'ensemble des actions possibles de l'automate à un instant donné est entièrement déterminé par l'état courant, les positions des têtes de lecture et/ou écriture et le contenu des bandes.

Une séquence d'actions est acceptante lorsqu'elle permet de lire la totalité de l'entrée en laissant l'automate dans un état acceptant. Une entrée est acceptée (ou reconnue) s'il existe une séquence acceptante pour cette entrée. L'ensemble des entrées acceptées (ou reconnues) est le langage reconnu par l'automate.

automate à pile automate possédant un nombre fini d'états et une bande de sortie indépendante utilisée comme zone de stockage intermédiaire appelée pile. La tête de lecture/écriture sur cette pile ne peut se déplacer que d'une unité dans chaque sens et elle ne peut lire que le symbole situé immédiatement avant elle (*pushdown automaton*) ou n'importe lequel des symboles situés avant elle (stack automaton).

La tête de sortie ne peut se déplacer dans le sens positif (push) qu'en écrivant un symbole sur la bande.

En général, le terme automate à pile est utilisé pour désigner un *pushdown automaton.*

$\Longrightarrow$ **automate**; sec. 5.4 p. 218.

automate déterministe automate dont les ensembles d'actions de tous les états sont réduits à zéro ou une action pour des contenus des bandes et positions des têtes déterminés.

$\Longrightarrow$ **automate, automate non-déterministe**.

automate fini automate possédant un nombre fini d'états et zéro bandes de sortie. Un automate fini est souvent utilisé comme automate accepteur pour déterminer si une entrée appartient à un langage donné.

Le terme *automate fini* est également utilisé dans un sens plus restrictif pour désigner un automate possédant une seule bande en entrée et une tête de lecture se déplaçant dans un seul sens sur la bande. Les automates fini non-déterministes sont équivalents aux automates finis déterministes.

$\Longrightarrow$ **automate**.

automate non-déterministe automate possédant des états pour lesquels il existe des ensembles d'actions contenant au moins deux éléments pour des contenus des bandes et positions des têtes déterminés.

$\Longrightarrow$ **automate, automate déterministe**.

automate stochastique automate dont chaque action possède une probabilité. La probabilité d'une séquence d'actions est égale au produit des probabilités de chaque action. La probabilité d'une entrée acceptée est égale à la somme des probabilités des séquences d'actions acceptant l'entrée.

$\Longrightarrow$ **automate**.

CFG de Context-Free Grammar.

$\Longrightarrow$ **grammaire non-contextuelle**.

Chomsky (forme normale) une grammaire non-contextuelle G est dite sous forme normale de Chomsky si toutes ses productions sont de la forme $A \rightarrow BC$ ou $A \rightarrow a$.

Toute grammaire non-contextuelle epsilon-free peut être transformée en une grammaire équivalente sous forme normale de Chomsky, éventuellement au prix d'une augmentation quadratique de son nombre de productions.

$\Longrightarrow$ **algorithme CYK**; rem. 1.15 p. 47; p. 207; sec. 5.3 p. 217.

contrainte de sélection forme de contrainte utilisée avec les grammaires S-attribuées et permettant de rejeter un ensemble de sous-arbres de dérivation en fonction d'un certain critère portant généralement sur les attributs synthétisés du non-terminal à la racine de ces sous-arbres. Les contraintes de sélection peuvent potentiellement modifier le langage généré par une grammaire.

$\Longrightarrow$ **contrainte d'optimisation**; sec. 1.6.2 p. 45 et 3.2 p. 93; def. 1.17 p. 45 et 2.14 p. 79; rem. 1.13 p. 45.

contrainte d'optimisation forme de contrainte utilisée avec les grammaires S-attribuées ambiguës et permettant de ne choisir qu'un seul sous-arbre de dérivation parmi un ensemble de sous-arbres par le biais d'une optimisation sur un attribut synthétisé.

$\Longrightarrow$ **contrainte de sélection**; sec. 1.5 p. 41; def. 2.13 p. 79.

contrainte syntaxique expression sous la forme de règles de dérivation des contraintes présentes dans un modèle donné.

$\Longrightarrow$ sec. 1.2.3 p. 19.

dérivation $\Longrightarrow$ **grammaire**.

dérivation (arbre) $\Longrightarrow$ **arbre de dérivation**.

dérivation (forêt) $\Longrightarrow$ **forêt de dérivation**.

équivalence (des grammaires) deux grammaires sont équivalentes lorsqu'elles génèrent le même langage. Il ne faut pas confondre équivalence et équivalence structurale des grammaires non-contextuelles (mêmes arbres de dérivation, mais étiquetés différemment).

$\Longrightarrow$ **grammaire, grammaire non-contextuelle**.

état d'analyse $\Longrightarrow$ **tabulaire**; sec. 1.3.2 p. 23; def. 1.6 p. 26 et 1.15 p. 43; (Schöbel-Theuer, 1996)

expression regulière une expression régulière permet de décrire un langage régulier à l'aide des parenthèses, symbole de concaténation (souvent implicite), choix (souvent représenté par le symbole |), et clôture transitive (souvent représentée par le symbole $*$). Des notations supplémentaires peuvent être introduites afin de faciliter la description d'expressions régulières complexes. L'expression régulière $((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)|0$ décrit l'ensemble des entiers naturels écrits en base 10. On trouve également cette expression régulière écrite sous la forme équivalente $[1-9][0-9]* \mid 0$.

$\Longrightarrow$ **langage régulier, lexème**.

forêt de dérivation ensemble des arbres de dérivation d'une chaîne de V^*. Lorsqu'une grammaire est à ambiguïté exponentielle, la forêt est généralement représentée sous forme compactée afin d'éviter une croissance exponentielle de sa taille.

$\Longrightarrow$ **arbre de dérivation, grammaire, tabulaire**.

grammaire une grammaire (générative) est composée d'un ensemble fini de terminaux V_T, d'un ensemble fini de non-terminaux V_N tel que $V_N \cap V_T = \emptyset$, d'un ensemble fini P de **productions** (dites aussi **règles de réécriture** ou règles de **dérivation**), et d'un symbole de départ $S \in V_N$. On note $V = V_N \cup V_T$ le **vocabulaire** de la grammaire.

Chaque production de P est de la forme $\beta A \beta' \to \alpha$, où α, β et β' appartiennent à V^*. Les productions sont donc aussi des **relations (de dérivation)** sur $V^* V_N V^* \times V^*$.

grammaire attribuée la définition suivante est tirée de (op den Akker *et al.*, 1990) :

Une grammaire attribuée G sur un domaine sémantique D est une grammaire non-contextuelle G_0 complétée par des attributs et des règles sémantiques. Un domaine sémantique D est une paire (Ω, Φ), où Ω est un ensemble d'ensembles (les ensembles des valeurs des attributs) et Φ est une collection d'applications de la forme $f : V_1 \times \cdots \times V_m \to V_0$, où $m \geqslant 0$ et $V_i \in \Omega$, $0 \leqslant i \leqslant m$.

L'ensemble des symboles d'attributs est noté A et est partitionné en I_A (symboles d'attributs hérités) et S_A (symboles d'attributs synthétisés). Pour chaque symbole d'attribut $b \in A$, un ensemble $V(b) \in \Omega$ contient l'ensemble des valeurs possibles des attributs correspondant à b. Il y a un ensemble fini fixé de symboles d'attributs associé à chaque symbole de la grammaire. Un attribut est une paire $X.a$, où $X \in V$ et $a \in A$. Nous supposons qu'aucun symbole d'attribut hérité n'est associé aux terminaux et au symbole de départ. Les ensembles d'attributs de chaque symbole sont ordonnés linéairement, les attributs hérités précédant les attributs synthétisés.

Une production $p : X_0 \to X_1 \ldots X_n$ a une occurrence d'attribut $k.b$, $0 \leqslant k \leqslant n$, si $X_k.b$ est un attribut. Une occurrence d'attribut $k.b$ de p est appelée une occurrence d'entrée, si $b \in I_A$ et $k = 0$, ou si $b \in S_A$ et $k > 0$. Autrement, $k.b$ est appelée une occurrence de sortie. Pour chaque occurrence de sortie $k.b$ de p, il y a exactement une règle sémantique $k.b = f(j_1.a_1, \ldots, j_m.a_m)$, où chaque $j_i.a_i$ est une occurrence d'entrée de p, et f est une fonction de la forme $f : V_1 \times \cdots \times V_m \to V_0$ dans Φ telle que $V_0 = V(b)$ et $V_i = V(a_i)$ pour $1 \leqslant i \leqslant m$.

$\Longrightarrow$ **grammaire**.

grammaire contextuelle ou sensible au contexte. Grammaire dont les productions sont de la forme $\beta A \beta' \to \beta \alpha \beta'$, où $A \in V_N$ et $\alpha \in V^*$ est non vide, à l'exception éventuelle d'une unique production $S \to \epsilon$ dont l'occurrence implique que S n'apparaît à droite d'aucune production de la grammmaire.

$\Longrightarrow$ **grammaire non-contextuelle**.

grammaire non-contextuelle grammaire dont les productions sont de la forme $A \rightarrow \alpha$, où $A \in V_N$. Les notions de grammaire non-contextuelle et d'automate à pile non-déterministe sont duales.

$\Longrightarrow$ **automate à pile, grammaire contextuelle**; def. 1.1 p. 17.

grammaire non-contextuelle propre grammaire non-contextuelle vérifiant les conditions supplémentaires suivantes :

- la grammaire ne contient ni symbole ni production inutile, c'est-à-dire n'apparaissant au cours d'aucune dérivation de S ou ne pouvant pas ultimement générer une chaîne terminale ;
- Il n'existe aucune production $A \rightarrow \epsilon$. On dit dans ce cas que la grammaire est sans dérivation vide, ou epsilon-free ;
- Il n'existe aucune dérivation $A \rightarrow^{+} A$. La grammaire est donc non-circulaire.

$\Longrightarrow$ **grammaire, grammaire non-contextuelle**; def. 1.2 p. 17; rem. 1.1 p. 18 et 2.2 p. 77.

grammaire non-contextuelle stochastique ou SCFG, désigne une grammaire non-contextuelle dont chaque dérivation possède une probabilité. Généralement, les probabilités de choix d'une dérivation sont indirectement connues par l'intermédiaire des distributions des chaînes du langage généré par cette grammaire. L'apprentissage d'une SCFG consiste à calculer les probabilités de dérivation à partir des distributions des chaînes générées.

$\Longrightarrow$ sec. 1.7.3 p. 64 et 3.9 p. 140.

grammaire S-attribuée ou SAG (pour S-attribute grammar). Une grammaire S-attribuée est une grammaire attribuée ne comportant pas d'attributs hérités.

$\Longrightarrow$ **grammaire attribuée**; def. 1.13 p. 39.

HMM de Hidden Markov Model.

$\Longrightarrow$ **modèle de Markov caché**.

homomorphisme application f d'un ensemble E possédant une loi interne $\star$ vers un ensemble F possédant une loi interne $\times$ et vérifiant $f(a_1 \star a_2) = f(a_1) \times f(a_2)$.

$\Longrightarrow$ def. 2.6 p. 73.

langage un langage est un ensemble de n-uplets de mots reconnus par un automate.

Le langage $L(G)$ généré par une grammaire G est l'ensemble des chaînes de terminaux dérivables du symbole de départ S : $L(G) = \{u \in V_T^* \mid S \rightarrow^* u\}$.

$\Longrightarrow$ **automate, grammaire**.

langage régulier langage reconnu par un automate fini ou, de façon équivalente, décrit par une expression régulière.

$\Longrightarrow$ **automate fini, expression régulière**.

lexème unité lexicale minimale d'un langage ou d'une grammaire. L'analyse lexicale convertit des chaînes de caractères en chaînes de lexèmes. Les lexèmes sont souvent décrits à l'aide d'expressions régulières.

$\Longrightarrow$ **analyse lexicale, expression régulière**.

LL(k), LR(k), LALR(k) une grammaire est LL(k) lorsque les mots du langage qu'elle décrit peuvent être reconnus de façon purement top-down à l'aide d'un automate déterministe lisant au plus k symboles en avant de la tête de lecture.

Les grammaires $LR(k)$ sont non ambiguës et sont reconnues par les automates à pile déterministes. Tout langage déterministe (i.e. reconnu par un automate à pile déterministe) possède une grammaire LR(1). Les grammaires $LL(k)$ sont incluses dans les grammaires $LR(k)$.

Les grammaires $LALR(k)$ sont une forme restreinte de grammaires $LR(k)$ pour lesquelles les tables d'état de l'automate à pile déterministe associé comportent le même nombre d'états que les grammaires $LR(0)$.

$\Longrightarrow$ **analyse syntaxique bottom-up, analyse syntaxique top-down, automate déterministe**.

modèle de covariance ou covariance model. Forme particulière d'automate stochastique possédant une bande d'entrée et une tête de lecture et dont les états sont globalement organisés en arbre devant être parcouru de haut en bas et de la gauche vers la droite. L'arbre des états représente la structure secondaire consensus d'une famille d'ARN.

$\Longrightarrow$ sec. 3.10 p. 156.

modèle de Markov caché ou HMM, désigne un automate fini stochastique dont les probabilités de transition d'un état vers l'autre sont indirectement connues par l'intermédiaire des distributions des chaînes reconnues par ces automates. L'apprentissage d'un HMM consiste à calculer des probabilités de transition à partir des distributions des chaînes générées.

$\Longrightarrow$ **automate stochastique**. section 3.8 p. 133.

monoïde ensemble E possédant une loi associative $\star$ interne avec un élément neutre e vérifiant pour tout élément a de E, $e \star a = a \star e = a$.

monoïde libre cas particulier de monoïde dont l'ensemble E est appelé alphabet et la loi associative, concaténation formelle des éléments de E. L'élément neutre est la concaténation vide. On note E^* le monoïde libre généré par E.

$\Longrightarrow$ **monoïde**; def. 1.13 p. 39 et 2.5 p. 73.

non-terminal $\Longrightarrow$ **grammaire**.

production $\Longrightarrow$ **grammaire**.

programmation dynamique l'idée générale sous-tendant la programmation dynamique est de décomposer un problème P à résoudre en une série de sous problèmes plus faciles à résoudre et amenant à une solution de P. Soit $Q_0, \ldots, Q_n$ cette série de sous-problèmes. Le sous-problème Q_0 doit être soluble sans information complémentaire. Q_1 doit être un problème que nous savons résoudre soit indépendamment de Q_0, soit à l'aide de la solution de Q_0. Plus généralement, Q_i doit être soluble étant données les solutions des sous-problèmes $Q_0, \ldots, Q_{i-1}$. La solution du dernier problème Q_n doit être équivalente à une solution de P ou doit permettre de trouver une solution à P. En général, le plus difficile est de trouver une décomposition convenable d'un problème P.

$\Longrightarrow$ sec. 1.7.2 p. 49, 3.3 p. 95, 3.5 p. 117.

règle (de réécriture) $\Longrightarrow$ **grammaire**.

SCFG de Stochastic Context Free Grammar.

$\Longrightarrow$ **grammaire non-contextuelle stochastique**.

swap déplacement d'un programme ou d'une partie d'un programme de la mémoire à accès rapide vers une mémoire à accès lent (swap out) ou vice versa (*swap in*).

$\Longrightarrow$ p. 38.

tabulaire type d'algorithme d'analyse syntaxique décomposant une chaîne analysée $a_1 \ldots a_n$ (ou $a_{\vec{1}} \ldots a_{\vec{n}}$ en version m-bandes) en sous-chaînes $a_i \ldots a_j$ et stockant dans une table indexée par i et j (explicitement ou implicitement) une représentation des dérivations “valides” de cette sous-chaîne sous la forme d'états d'analyse.

La table d'analyse des algorithmes tabulaires permet de factoriser en au plus $O(n^2)$ entrées la forêt de dérivation d'une chaîne $a_1 \ldots a_n$, même si la grammaire utilisée est à ambiguïté exponentielle.

$\Longrightarrow$ **ambiguïté, analyse syntaxique, forêt de dérivation**; def. 1.3 p. 23 et 1.7 p. 26; p. 21.

terminal $\Longrightarrow$ **grammaire**.

token $\Longrightarrow$ **lexème**.

vocabulaire $\Longrightarrow$ **grammaire**.

Bibliographie

Abdeddaim, S. (1996). *Comparaison de Séquences Biologiques*. Thèse de doctorat, Université Pierre et Marie Curie. cité page(s) 86, 170

Aho, A. V., Sethi, R., Ullman, J. D. (1988). *Compilers. Principles, Techniques, and Tools*. Addison-Wesley. cité page(s) 25, 211

Aho, A. V., Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, tm. 1. Prentice-Hall, Inc. cité page(s) 17, 18, 18, 22, 22, 24, 25, 27, 28, 210, 211, 212, 212

op den Akker, R., Melichar, B., Tarhio, J. (1990). The Hierarchy of LR-Attributed Grammars. Dans Deransart, P., Jourdan, M., réds., *Attribute Grammars and their Applications*, tm. 461 de *lncs*, pp. 13–28. Springer-Verlag. cité page(s) 216, 280

Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D. (1990). A basic local alignment search tool. *Journal of Molecular Biology* 215:403–410. cité page(s) 225

Altschul, S. F., Lipman, D. J. (1989). Trees, stars, and multiple biological sequence alignment. *SIAM Journal on Applied Mathematics* 49:179–209. cité page(s) 87

Apostolico, A., réd. (1993). *Proceedings of the Fourth Symposium on Combinatorial Pattern Matching*, tm. 684 de *Lecture Notes in Computer Science*. Springer-Verlag. cité page(s) 295, 296

Bafna, V., Muthukrishnan, S., Ravi, R. (1995). Computing Similarity between RNA strings. Dans Galil & Ukkonen (1995), pp. 1–16. cité page(s) 122, 123

Baldi, P., Chauvin, Y., Hunkapiller, T., McClure, M. A. (1994). Hidden Markov models of biological primary sequence information. *Proceedings of the National Academy of Sciences of the USA* 91:1059–1063. cité page(s) 133

Bar-Hillel, Y. (1964). *Language and information*. Addison-Wesley, Reading, Massachusetts. cité page(s) 18

van Belkum, A., Abrahams, J. P., Pleij, C. W. A., Bosch, L. (1985). Five pseudoknots are present at the 204 nucleotides long 3' noncoding region of tobacco mosaic virus RNA. *Nucleic Acids Research* 13:7673–7686. cité page(s) 9, 171 van Belkum, A., Bingkun, J., Rietveld, K., Pleij, C. W. A., Bosh, L. (1987). Structural similarities among valine-accepting tRNA-like structures in tymoviral RNAs and elongator tRNAs. *Biochemistry* 26:1144–1151. cité page(s) 175

Billot, S., Lang, B. (1989). The Structure of Shared Forests in Ambiguous Parsing. Rap. tech. 1038, INRIA, Rocquencourt, France. cité page(s) 227

Birney, E. (1996). Dynamite: A generic specification for dynamic programming methods. Poster of the International Symposium on Theoretical and Computational Genome Research, Heidelberg. cité page(s) 226

Boyer, R., Moore, S. (1977). A fast string searching algorithm. *Communications of the ACM* 20:762–772. cité page(s) 225

Brown, M. (1995). RNA Pseudoknot Modeling Using Intersections of Stochastic Context Free Grammars with Applications to Database Search. Available by anonymous ftp at ftp.cse.ucsc.edu in /pub/rna. Presented at Pacific Symposium on Biocomputing Jan. 1996. cité page(s) 18, 64, 171, 172

Carillo, H., Lipman, D. J. (1988). The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics* 48:1073–1082. cité page(s) 87

Cary, R. B., Stormo, G. D. (1995). Graph-Theoretic Approach to RNA Modeling Using Comparative Data. Dans *Third International Conference on Intelligent Systems for Molecular Biology*, pp. 75–80. AAAI Press. cité page(s) 9, 171

Chan, L., Zuker, M., Jacobson, A. B. (1991). A computer method for finding common base paired helices in aligned sequences: application to the analysis of random sequences. *Nucleic Acids Research* 19:353–358. cité page(s) 164

Chan, S. C., Wong, A. K. C., Chiu, D. K. Y. (1992). A survey of multiple sequence comparison methods. *Bulletin of Mathematical Biology* 54:563–598. cité page(s) 86

Chao, K.-M. (1994). Computing all Suboptimal Alignments in Linear Space. Dans Crochemore & Gusfield (1994), pp. 31–42. cité page(s) 91

Chao, K.-M., Hardison, R. C., Miller, W. (1993). Constrained sequence alignment. *Bulletin of Mathematical Biology* 55:503–524. cité page(s) 91

Chao, K.-M., Miller, W. (1995). Linear-Space Algorithms that Build Local Alignments from Fragments. *Algorithmica* 13:106–134. cité page(s) 225

Chao, K.-M., Zhang, J., Ostell, J., Miller, W. (1995). A local alignment tool for very long DNA sequences. *Computing Applications in the Biosciences* 11(2):147–153. cité page(s) 91

Vauchaussade de Chaumont, M. (1985). *Nombre de Strahler des arbres, langages algébriques et dénombrement de structures secondaires en biologie moléculaire.* Thèse de maître, Université de Bordeaux I. cité page(s) 13

Chevalet, C., Michot, B. (1992). An algorithm for comparing RNA secondary structures and searching for similar substructures. *Computing Applications in the Biosciences* 8(3):215–225. cité page(s) 97

Chrobak, Li (1988). $k+1$ heads are better than k heads for PDA. *Journal of Computer and System Sciences* 37(2):144–155. cité page(s) 218, 221, 222

Chrobak, M. (1985). Hierarchies of one-way multihead automata languages. Dans Brauer, W., réd., *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, tm. 194 de *Lecture Notes in Computer Science*, pp. 101–110. Springer Verlag. cité page(s) 218, 221

Corpet, F., Michot, B. (1994). RNAlign program: alignment of RNA sequences using both primary and secondary structures. *Computing Applications in the Biosciences* 10(4):389–399. cité page(s) 117, 118, 119, 168

Crochemore, M., Gusfield, D., réds. (1994). *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching*, tm. 807 de *Lecture Notes in Computer Science*. Springer-Verlag. cité page(s) 286, 289, 290, 293

ten Dam, E., Pleij, K., Draper, D. (1992). Structural and functional aspects of RNA pseudoknots. *Biochemistry* 31:11665–11676. cité page(s) 9, 171

Dayhoff, M. O., Schwartz, R. M., Orcutt, B. C. (1978). A model of evolutionary change in proteins. Dans Dayhoff, M. O., réd., *Atlas of Protein Sequence and Structure*, tm. 5 supplement 3, pp. 345–352. National Biomedical Research Foundation, Washington DC. cité page(s) 86

Deransart, P., Jourdan, M., Lorho, B., réds. (1988). *Attribute Grammars*, tm. 323 de *Lecture Notes in Computer Science*. cité page(s) 213

Donnelly, C., Stallman, R. (1990). *BISON, The YACC-compatible Parser Generator. Version 1.12*. Free Software Foundation. cité page(s) 49, 230

Dumas, J. P., Ninio, J. (1982). Efficient algorithm for folding and comparing nucleic acid sequences. *Nucleic Acids Research* 10:197–206. cité page(s) 12

Earley, J. (1968). *An Efficient Context-Free Parsing Algorithm*. Thèse de doctorat, Carnegie-Mellon University, Pittsburgh. cité page(s) 25

Earley, J. (1970). An Efficient Context-Free Parsing Algorithm. *Communication of the ACM* 13(2):94–102. cité page(s) 22, 38, 38

Eddy, S. R. (1995). Multiple alignment using hidden Markov models. Dans *Third International Conference on Intelligent Systems for Molecular Biology*, pp. 114–120. AAAI Press. cité page(s) 96, 133

Eddy, S. R. (1996). Hidden Markov models. Dans *Current Opinions in Structural Biology*. in press, available in ftp://genome.wustl.edu/pub/eddy/hmm-review.ps.Z. cité page(s) 133, 137

Eddy, S. R., Durbin, R. (1994). RNA sequence analysis using covariance models. *Nucleic Acids Research* 22(11):2079–2088. cité page(s) 11, 64, 66, 156, 158, 164

Florentz, C., Briand, J. P., Gierge, R. (1984). Possible functional role of viral tRNA-like structures. *FEBS Letter* 176:295–300. cité page(s) 9, 171

Fox, G. E., Woese, C. R. (1973). 5S RNA secondary structure. *Nature* 256:505–507. cité page(s) 97

Francon, J., Viennot, G., Vuillemin, J. (1978). Description and Analysis of an Efficient Priority Queue Representation. Dans *Proceedings FOCS*, tm. 19, pp. 1–7, Ann Arbor MI. cité page(s) 194

Freier, S. M., Kierzek, R., Jaeger, J. A., Sugimoto, N., Caruthers, M. H., Neilson, T., Turner, D. H. (1986). Improved free-energy parameters for predictions of RNA duplex stability. *Proceedings of the National Academy of Sciences of the USA* 83:9373–9377. cité page(s) 11, 11, 11, 52

Fuellen, G. (1996). A Gentle Guide to Multiple Alignment version 2.02. An HTML version is available at http://www.techfak.uni-bielefeld.de/bcd/Curric/MulAli/mulali.html. cité page(s) 87

Fujiwara, Y., Asogawa, M., Konagaya, A. (1994). Stochastic motif extraction using hidden Markov model. Dans *Second International Conference on Intelligent Systems for Molecular Biology*, pp. 121–129. AAAI Press. cité page(s) 133

Galil, Z., Ukkonen, E., réds. (1995). *Proceedings of the Sixth Symposium on Combinatorial Pattern Matching*, tm. 937 de *Lecture Notes in Computer Science*. Springer-Verlag. cité page(s) 285, 289

Garcia-Arenal, F. (1988). Sequence and structure at the genome 3' end of the U2-strain of Tobacco mosaic virus, a histidine-accepting tobamovirus. *Virology* 167:201–206. cité page(s) 175

Gaspin, C., Bourret, P., Westhof, E. (1995). Détermination assistée par ordinateur de structures secondaires d'ARN. *Technique et Science Informatique* 14(2):141–160. cité page(s) 12

Gesteland, R. F., Atkins, J. F., réds. (1993). *The RNA World*. Cold Spring Harbor Laboratory Press. cité page(s) 7

Gonnet, G. H., Baeza-Yates, R. (1991). *Handbook of Algorithms and Data Structures*. Addison-Wesley. cité page(s) 194

Graham, S. L., Harrison, M. A., Ruzzo, W. L. (1980). An Improved Context-Free Recognizer. *ACM Transactions on Programming Languages and Systems* 2(3):415–462. cité page(s) 4, 23, 31, 208, 212

Grahne, G., Nykänen, M., Ukkonen, E. (1994). Reasoning about Strings in Databases. Dans *13th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 303–312. cité page(s) 227, 227

Grate, L. (1995a). Automatic RNA Secondary Structure Determination with Stochastic Context-Free Grammars. Dans *Third International Conference on Intelligent Systems for Molecular Biology*, pp. 136–144. AAAI Press. cité page(s) 64, 67, 164, 169, 170, 170

Grate, L. (1995b). Source of Rebecca Underwood's U1 grammar. Personnal communication. cité page(s) 69

Grate, L., Hebster, M., Hughey, R., Haussler, D., Mian, I. S., Noller, H. (1994). RNA modeling using Gibbs sampling and stochastic context free grammars. Dans *Second International Conference on Intelligent Systems for Molecular Biology*. AAAI Press. cité page(s) 64

Gupta, S. K., Kececioglu, J. D., Schäffer, A. A. (1995). Making the Shortest-Paths Approach to Sum-of-Pairs Multiple Sequence Alignment More space Efficient in Practice. Dans Galil & Ukkonen (1995), pp. 128–143. cité page(s) 87

Han, K., Kim, H. (1993). Prediction of common folding structures of homologous RNAs. *Nucleic Acids Research* 21:1251–1257. cité page(s) 11, 98, 164

Harrison, M. A. (1978). *Introduction to Formal Language Theory*. Addison-Wesley. cité page(s) 17, 206

Harrison, M. A., Ibarra, O. H. (1968). Multi-Tape and Multi-Head Pushdown Automata. *Information and Control* 13:433–470. cité page(s) 218, 220, 220, 220, 220, 221, 222, 223, 223

Higgins, D. G., Bleasby, A. J., Fuchs, R. (1992). CLUSTAL V: improved software for multiple sequence alignment. *Computing Applications in the Biosciences* 8(2):189–191. cité page(s) 87

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18:341–343. cité page(s) 91, 225

Huang, X. (1994a). A context dependent method for comparing sequences. Dans Crochemore & Gusfield (1994), pp. 54–63. cité page(s) 91

Huang, X. (1994b). On global sequence alignment. *Computing Applications in the Biosciences* 10(3):227–235. cité page(s) 87

Huang, X., Miller, W. (1991). A time-efficient, linear-space local similarity algorithm. *Advances in Applied Mathematics* 12:337–357. cité page(s) 91

Ibarra, O. H. (1967). *Generalizations of pushdown automata.* Thèse de doctorat, University of California, Berkeley. cité page(s) 218, 220, 221, 222, 223, 223

Jaeger, J. A., Turner, D. H., Zuker, M. (1989). Improved predictions of secondary structures for RNA. Dans *Proceedings of the National Academy of Sciences of the USA*, pp. 7706–7710. cité page(s) 11, 52

James, B. D., Olsen, G. J., Pace, N. R. (1989). Phylogenetic comparative analysis of RNA secondary structures. *Methods in Enzymology* 180:227–239. cité page(s) 9, 11, 117

Jazayeri, M., Ogden, W. F., Rounds, W. C. (1975). The intrinsically exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM* pp. 691–706. cité page(s) 215

Jiang, T., Wang, L., Zhang, K. (1994). Alignment of trees - an alternative to tree edit. Dans Crochemore & Gusfield (1994), pp. 75–86. cité page(s) 98, 127, 132, 133, 133

Jourdan, M., Parigot, D. (1990). Techniques for Improving Grammar Flow Analysis. Dans *European Symposium on Programming*, tm. 432 de *Lecture Notes in Computer Science*, pp. 240–255. Springer-Verlag. cité page(s) 211

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory* 2:127–145. Correction: Mathematical Systems Theory 5: 95–96, 1971. cité page(s) 39, 213

Knuth, D. E. (1992). *Literate Programming.* Sanford University, Stanford, California. cité page(s) 179

Knuth, D. E., Morris, J. H., Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing* 6:323–350. cité page(s) 122, 225

Konings, D. A. M., Hogeweg, P. (1989). Pattern analysis of RNA secondary structure — similarity and consensus of minimal energy folding. *Journal of Molecular Biology* 207:597–614. cité page(s) 122

Krogh, A., Brown, M., Mian, I. S., Sjölander, K., Haussler, D. (1994a). Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology* 235:1501–1531. cité page(s) 133

Krogh, A., Mian, I. S., Haussler, D. (1994b). A hidden Markov model that finds genes in *e. coli* DNA. *Nucleic Acids Research* 22(22):4768–4778. cité page(s) 133

Kruseman Aretz, F. E. J. (1989). A New Approach To Earley's Parsing Algorithm. *Science of Computer Programming* 12:105–121. cité page(s) 23, 31, 209

Kulp, D., Haussler, D., Reese, M. G., Eeckman, F. H. (1996). A Generalized Hidden Markov Model for the Recognition of Human Genes in DNA. Dans *Fourth International Conference on Intelligent Systems for Molecular Biology*, pp. 134–142. AAAI Press. cité page(s) 138

Lang, B. (1992). Recognition can be Harder than Parsing. Dans *Second International Workshop on Tree Adjoining Grammars.* cité page(s) 227

Le, S.-Y., Nussinov, R., Maizel, Jr., J. V. (1989a). Tree Graphs of RNA secondary structures and their comparisons. *Computers and Biomedical Research* 22:461–473. cité page(s) 98

Le, S.-Y., Owens, J., Nussinov, R., Chen, J.-H., Shapiro, B., Maizel, Jr., J. V. (1989b). RNA secondary structures: comparison and determination of frequently recurring substructures by consensus. *Computing Applications in the Biosciences* 5(3):205–210. cité page(s) 98

Le, S. Y., Shapiro, B. A., Chen, J. H., Nussinov, R., Maizel, Jr., J. V. (1991). RNA pseudoknots downstream of the frameshift sites of retroviruses. *Genet Anal Tech Appl* 8:191–205. cité page(s) 9, 171

Lee, J. (1985). a YACC grammar for ANSI C. draft version of the ANSI C standard. Available by anonymous ftp at ftp.uu.net in usenet/net.sources/ansi.c.grammar.Z. cité page(s) 49, 93

Leermakers, R. (1992). Recursive ascent parsing : from Earley to Marcus. *Theoretical Computer Science* 104:299–312. cité page(s) 208

Lefebvre, F. (1994). Modélisation d'empreintes génétiques et application à la construction d'une carte physique du génome humain. Rap. tech. 1439, LIX, École Polytechnique, 91129 Palaiseau Cedex, France. cité page(s) 4

Lefebvre, F. (1995a). Modèle pour le calcul rapide de contigs. *Technique et science informatiques* 14(1):35–46. cité page(s) 4

Lefebvre, F. (1995b). An Optimized Parsing Algorithm Well Suited to RNA Folding. Dans *Third International Conference on Intelligent Systems for Molecular Biology*, pp. 222–230. AAAI Press. cité page(s) 5, 183

Lefebvre, F. (1996a). A grammar-based unification of several alignment and folding algorithms. Dans *Fourth International Conference on Intelligent Systems for Molecular Biology*, pp. 143–154. AAAI Press. cité page(s) 5

Lefebvre, F. (1996b). *Literate programming source of MTSAG2C.* LIX, École Polytechnique, 91128 Palaiseau Cedex, France. available in ftp://lix.polytechnique.fr/pub/lefebvre. cité page(s) 179, 183, 186

Leo, J. M. (1991). A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science* 82:165–176. cité page(s) 25, 209

Lewis, P., Rosenkrantz, D. J., Stearns, R. E. (1974). Attributed translations. *Journal of Computer and System Sciences* 9:279–307. cité page(s) 39, 215

Lipman, D. J., Altschul, S. F., Kececioglu, J. D. (1989). A tool for multiple sequence alignment. *Proceedings of the National Academy of Sciences of the USA* 86:4412–4415. cité page(s) 87

Lowe, T., Eddy, S. (1996). tRNAscan-SE: A Package for tRNA Identification Combining the Speed, Sensitivity, and Selectivity of Three Complementary Detection Methods. Poster of the Fourth International Conference on Intelligent Systems for Molecular Biology. cité page(s) 65, 133

McClure, M. A., Smith, C., Elton, P. (1996). Parametrization studies for the SAM and HMMER methods of hidden Markov model generation. Dans *Fourth International Conference on Intelligent Systems for Molecular Biology*, pp. 155–164. AAAI Press. cité page(s) 137

McLean, P., Horspool, R. N. (1996). A Faster Earley Parser. Dans Gyimóthy, T., réd., *Compiler Construction*, tm. 1060 de *Lecture Notes in Computer Science*, pp. 281–293. Springer-Verlag. cité page(s) 208

Myers, E. W. (1991). An Overview of Sequence Comparison Algorithms in Molecular Biology. Rap. tech. 29, University of Arizona. Available by anonymous ftp at ftp.cs.arizona.edu in /people/gene/PAPERS. cité page(s) 74, 86

Myers, E. W., Miller, W. (1988). Optimal alignments in linear space. *Computing Applications in the Biosciences* 4:11–17. cité page(s) 91

Nederhof, M.-J. (1993). A Multidisciplinary View on PLR Parsing. Dans *Twentieth Workshop on Language Technology 6*. cité page(s) 23, 209

Noller, H. F., Kop, J. A., Wheaton, V., Brosius, J., Gutell, R. R., Kopylov, A. M., Dohme, F., Herr, W., Stahl, D. A., Gupta, R., Woese, C. R. (1981). Secondary structure model for 23S ribosomal RNA. *Nucleic Acids Research* 9(22):6167–6189. cité page(s) 97

Nucleic Acids Research vol 24 number 1 (1996). Special Issue on Databases. cité page(s) 226

Papanicolaou, C., Gouy, M., Ninio, J. (1984). An energy model that predicts the correct folding of both the tRNA and the 5S RNA molecules. *Nucleic Acids Research* 12:31. cité page(s) 11, 52

Paxon, V., Gong, K. (1993). *FLEX, Fast lexical analyzer generator 2.4.5.* Free Software Foundation. cité page(s) 232

Pearson, W. R., Lipman, D. J. (1988). Improved Tools for Biological Sequence Comparison. *Proceedings of the National Academy of Sciences of the USA* 85:2444–2448. cité page(s) 97, 105

van de Peer, Y., Nicolaï, S., de Rijk, P., de Wachter, R. (1996). Database on the structure of small ribosomal subunit RNA. *Nucleic Acids Research* 24(1):86–91. cité page(s) 226

Peltola, H., Söderlund, H., Ukkonen, E. (1986). Algorithms for the search of amino acid patterns in nucleic acid sequences. *Nucleic Acids Research* 14(1):99–107. cité page(s) 93

Pevzner, P. A. (1992). Multiple alignment, communication cost, and graph matchings. *SIAM Journal on Applied Mathematics* 52:1763–1779. cité page(s) 87

Piastra, M., Bolognesi, R. (1991). An Efficient Context-Free Parsing Algorithm with Semantic Actions. Dans *Proceedings of the Second Congress of the Italian Association for Artificial Intelligence on Trends in Artifical Intelligence*, tm. 549 de *Lecture Notes in Computer Science*, pp. 271–280. Springer-Verlag. cité page(s) 209

Pleij, C. W. A. (1993). RNA Pseudoknots. Dans Gesteland, R. F., Atkins, J. F., réds., *The RNA World*, pp. 609–613. Cold Spring Harbor Laboratory Press. cité page(s) 9

Pleij, C. W. A. (1995). Structure and function of RNA pseudoknots. *Genetic Engineering* 17:67–80. cité page(s) 9, 171

Ramsey, N. (1994). Literate Programming Simplified. *IEEE Software* 11(5):97–105. cité page(s) 179

Rytter, W. (1995). Context-free recognition via shortest paths computation : a version of Valiant's algorithm. *Theoretical Computer Science* 143:343–352. cité page(s) 206

Sagot, M.-F. (1996). *Ressemblance lexicale et structurale entre macromolécules - Formalisation et approches combinatoires.* Thèse de doctorat, Université de Marne-la-Vallée, France. cité page(s) 86, 87

Sakakibara, Y. (1990). LEARNING CONTEXT-FREE GRAMMARS FROM STRUCTURAL DATA IN POLYNOMIAL TIME. *Theoretical Computer Science* 76:223–242. cité page(s) 5

Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C., Haussler, D. (1994a). Recent methods for RNA modeling using stochastic context-free grammars. Dans Crochemore & Gusfield (1994), pp. 289–306. cité page(s) 64, 143, 144

Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C., Haussler, D. (1994b). Stochastic Context-Free Grammars for tRNA Modeling. *Nucleic Acids Research* 22:5112–5120. cité page(s) 64, 65, 67, 67, 143

Sakakibara, Y., Brown, M., Mian, I. S., Underwood, R. C., Haussler, D. (1994c). Stochastic context free grammars for RNA modeling. Dans *Proceedings of the Hawaii International Conference on System Sciences*, Los Alamitos, CA. IEEE Computer Society Press. cité page(s) 64

Sakakibara, Y., Brown, M., Underwood, R. C., Mian, I. S., Haussler, D. (1993). Stochastic Context-Free Grammars for Modeling RNA. Rap. tech. UCSC-CRL-93-16, University of California, Santa Cruz, Santa Cruz CA 95064 USA. cité page(s) 39, 64, 66

Salomaa, A. (1973). *Formal Languages*. Academic Press, Inc. cité page(s) 17, 222

Sankoff, D. (1985). Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics* 45(5):810–825. cité page(s) 98, 100, 100, 101, 101, 102

Sankoff, D., Kruskal, J. B. (1983). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparisons*. Addison-Wesley, Reading, MA. cité page(s) 87

Schöbel-Theuer, T. (1996). Towards a Unifying Theory of Context-Free Parsing. Rap. tech. 6, Universität Stuttgart, Institut für Informatik. cité page(s) 276, 279

Searls, D. B. (1992). The linguistics of DNA. *American Scientist* 80:579–591. cité page(s) 13

Searls, D. B., Murphy, K. P. (1995). Automata – Theoretic Models of Mutation and Alignment. Dans *Third International Conference on Intelligent Systems for Molecular Biology*, pp. 341–349. AAAI Press. cité page(s) 5, 74, 89, 92, 234

Shapiro, B. A. (1988). An algorithm for comparing multiple RNA secondary structures. *Computing Applications in the Biosciences* 4(3):387–393. cité page(s) 122

Shapiro, B. A., Zhang, K. (1990). Comparing multiple RNA secondary structures using tree comparisons. *Computing Applications in the Biosciences* 6(4):309–318. cité page(s) 97, 98, 127, 127, 132, 133

Smith, T. F., Waterman, M. S. (1981). Identification of Common Molecular Subsequences. *Journal of Molecular Biology* 147:195–197. cité page(s) 88, 269

Sobel, E., Martinez, H. (1986). A multiple sequence alignment program. *Nucleic Acids Research* 14:363–374. cité page(s) 87

Sprinzi, M., Steegborn, C., Hübel, F., Steinberg, S. (1996). Compilation of tRNA sequences and sequences of tRNA genes. *Nucleic Acids Research* 24(1):68–72. cité page(s) 96, 105, 109, 114, 114, 117, 117, 149, 157

Stolcke, A., Omohundro, S. (1994). Inducing Probabilistic Grammars by Bayesian Model Merging. Dans *International Conference on Grammatical Inference*, tm. 862 de *Lecture Notes in Computer Science*, pp. 106–118. Springer-Verlag. cité page(s) 5

Subbiah, S., Harrison, S. C. (1989). A Method for Multiple Sequence Alignment with Gaps. *Journal of Molecular Biology* 20:539–548. cité page(s) 87

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2):146–160. cité page(s) 197

Taylor, W. R. (1987). Multiple sequence alignment by a pairwise algorithm. *Computing Applications in the Biosciences* 3(2):81–87. cité page(s) 87

Thompson, J., Higgins, D. G., Gibson, T. J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research* 22:4673–4680. cité page(s) 87

Thompson, J. D. (1995). Introducing variable gap penalties to sequence alignment in linear space. *Computing Applications in the Biosciences* 11(2):181–186. cité page(s) 91

Tomita, M. (1987). *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston. cité page(s) 208

Tomita, M. (1988). Graph-structured Stack and Natural Language Parsing. Dans *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, pp. 249–257. cité page(s) 208

Turner, D. H., Sugimoto, N., Jaeger, J. A., Longfellow, C. E., Freier, S. M., Kierzek, R. (1987). Improved parameters for prediction of RNA structure. No. 52 dans Cold Spring Harbor Symposia on Quantitative Biology, pp. 123–133. cité page(s) 11, 52

Underwood, R. C. (1994). Stochastic Context-Free Grammars for Modeling Three Spliceosomal Small Nuclear Ribonucleic Acids. Rap. tech. UCSC-CRL-94-23, PhD thesis, University of California, Santa Cruz, Santa Cruz CA 95064 USA. cité page(s) 64, 65, 69, 150, 151

Valiant, L. G. (1975). General Context-Free Recognition in Less than Cubic Time. *Journal of Computer and System Sciences* 10:308–315. cité page(s) 206

Vingron, M., Argos, P. (1989). A fast and sensitive multiple sequence alignment algorithm. *Computing Applications in the Biosciences* 5(2):115–121. cité page(s) 87

Vingron, M., Pevzner, P. A. (1993). Multiple Sequence Comparison and n-Dimensional Image Reconstruction. Dans Apostolico (1993), pp. 243–253. cité page(s) 87

Voisin, F. (1988). A bottom-up adaptation of Earley's parsing algorithm. Dans Deransart, P., Lorho, B., Maluszyński, J., réds., *Proceedings of the International Workshop on Programming Languages Implementation and Logic Programming*, tm. 348 de *Lecture Notes in Computer Science*. cité page(s) 23, 209

Wang, C., Le, S. Y., Ali, N., Siddiqui, A. (1995). An RNA Pseudoknot is an essential structural element of the internal ribosome entry site located within the hepatitis C virus 5' noncoding region. *RNA* 1:526–537. cité page(s) 9, 171

Wang, L., Gusfield, D. (1996). Improved Approximation Algorithms for Tree Alignment. Dans Hirschberg, D., Myers, G., réds., *Proceedings of the Seventh Symposium on Combinatorial Pattern Matching*, tm. 1075 de *Lecture Notes in Computer Science*, pp. 220–233. Springer-Verlag. cité page(s) 87

Waterman, M. S. (1978). Secondary structure of single-stranded nucleic acids. *Studies in Foundations and Combinatorics, Advances in Mathematics Supplementary Studies* 1:167. cité page(s) 14

Williams, A. L., Tinoco, Jr., I. (1986). A dynamic programming algorithm for finding alternative RNA secondary structures. *Nucleic Acids Research* 14:299–315. cité page(s) 98

Winker, S., Overbeek, R., Woese, C. R., Olsen, G. J., Pfluger, N. (1990). Structure detection through automated covariance search. *Computing Applications in the Biosciences* 6(4):365–371. cité page(s) 164

Woese, C. R., Pace, N. R. (1993). Probing RNA Structure, Function, and History by Comparative Analysis. Dans Gesteland, R. F., Atkins, J. F., réds., *The RNA World*, chap. 4, pp. 91–118. Cold Spring Harbor Laboratory Press. cité page(s) 9, 11, 97

Zhang, K. (1993). A New Editing based Distance between Unordered Labeled Trees. Dans Apostolico (1993), pp. 254–265. cité page(s) 133

Zuker, M. (1989a). On finding all suboptimal foldings of an RNA molecule. *Science* 244(7):48–52. cité page(s) 12

Zuker, M. (1989b). The Use of Dynamic Programming Algorithms in RNA Secondary Structure Prediction. Dans Waterman, M. S., réd., *Mathematical Methods for DNA Sequences*, chap. 7, pp. 159–184. CRC Press. cité page(s) 12, 39, 50, 50, 56, 69, 98

Zuker, M., Sankoff, D. (1984). RNA secondary structures and their prediction. *Bulletin of Mathematical Biology* 46:591–621. cité page(s) 12

Table des matières