N° d'ordre: 00000

THÈSE

présentée

Université Toulouse III – Paul Sabatier

pour l'obtention du

Doctorat de l'Université Toulouse III – Paul Sabatier
Mention Informatique

par

Matthias Zytnicki

Équipe d'accueil : INRA – BIA
École Doctorale : EDIT

Titre de la thèse :

Localisation d'ARN non-codants par réseaux de contraintes pondérées

À soutenir devant la commission d'examen :

M. :	Martin	Cooper	Président
MM. :	Christian	Bessière	Rapporteurs
	François	Major	
MM. :	Alain	Denise	Examinateurs
	Marie-France	Sagot	
MM. :	Christine	Gaspin	Directeurs de thèse
	Thomas	Schiex	

Table des matières

Introduction

Depuis quelques dizaines d'années, la biologie moléculaire connaît un essor important grâce aux récents progrès techniques. La possibilité toute nouvelle d'observer les échelles moléculaires et d'interagir avec les éléments de cette taille a ouvert tout un pan de la biologie. Cette découverte est d'autant plus fascinante qu'elle nous apprend les mécanismes les plus élémentaires de la naissance, de la croissance, de la reproduction, de la sénescence mais aussi des maladies. Fait relativement rare, l'essor de la biologie moléculaire est connue du grand public, par l'intermédiaire de grandes campagnes comme le Téléthon ou la lutte contre le cancer.

Bon nombre de scientifiques, de toutes disciplines, ont suivi cet engouement. La sollicitation générale des contributions scientifiques répond bien sûr au défi que lance l'observation au niveau moléculaire. On ne compte plus les revues mêlant cette discipline à une autre science : *Biometrics*, *Biometrika*, *BMC Biochemistry*, *BMC Biotechnology*, *BMC Chemical Biology*, *Journal of Nanobiotechnology*, etc. L'informatique n'est pas en reste avec des revues internationales comme *BMC Bioinformatics*, *Bioinformatics*, *Journal of Computational Biology*... Les collaborations entre les deux domaines scientifiques ont ouvert un grand nombre d'axes d'études. On peut par exemple citer l'annotation de génomes, l'analyse des réseaux de régulation, la biologie évolutive, l'analyse des données d'expression, etc. Parmi ces domaines, l'analyse de séquences est un champ de recherche toujours très actif. Cela est d'autant plus vrai depuis le séquençage — c'est-à-dire la lecture des séquences génomiques — du premier organisme, le bactériophage Φ-X174 par Fred Sanger en 1977 [SAB+77]. En effet, les méthodes de séquençage de génomes complets sont à ce point abouties que le nombre de nucléotides stockés dans une banque de données comme EMBL[1] [KAA+07] double tous les dix-huit mois environ. Considérant la taille de l'information collectée, la compréhension des mécanismes régis par l'ADN séquencé et l'interprétation des données génomiques stockées dans ces bases de données deviennent maintenant impossible sans l'intervention de l'ordinateur.

L'analyse de séquences d'ADN s'est longtemps concentrée sur l'étude des protéines et de la partie de l'ADN codant pour ces protéines. En effet, les protéines étaient supposées être les principaux acteurs cellulaires, prenant part à l'activité de régulation, catalyse, signalisation, transport, etc. La partie codante n'occupant que 2 % du génome humain, on pensait que les 98 % restant était de la « matière noire », sans fonction biologique précise. Or le séquençage complet et l'analyse des génomes a permis de faire apparaître

[1] EMBL est accessible à l'adresse http://www.ebi.ac.uk/embl/.

que cette partie non-codante assurait plusieurs fonctions. Les vingt dernières années ont vu par exemple l'émergence d'un intérêt pour certaines régions qui étaient transcrites mais pas traduites, donnant des ARN non codants pour des protéines. Le rôle de ces ARN dits non-codants s'est révélé ces dernières années de plus en plus important et il s'avère qu'ils interviennent dans de nombreux mécanismes cellulaires. L'importance de ces molécules est d'ailleurs soulignée par deux événements majeurs : l'attribution de *breakthrough of the year 2002* par le journal *Science* [Cou02] et du prix Nobel de médecine à Andrew Z. Fire et Craig C. Mello, deux chercheurs ayant travaillé sur certains ARN non-codants.

À la différence des ARN codant pour des protéines, appelés ARN messagers, la connaissance des ARN est assez parcellaire. Un premier pas vers la compréhension des fonctions de ces ARN non-codant est sans aucun doute la recherche systématique de ceux-ci. Pour cela, il est tout d'abord possible de demander l'aide d'un biologiste. Mais la taille des séquences génomiques considérées descend rarement en dessous du million de nucléotides et atteint souvent les centaines de millions. Il faut donc faire appel à des méthodes informatiques. Une possibilité consiste à réutiliser les méthodes utilisées pour la détection de gènes de protéines. Ces travaux cherchent à trouver toutes les régions de l'ADN codant pour des protéines, mais ces méthodes s'appliquent mal à la recherche d'ARN non-codants. Il faut donc se tourner vers d'autres méthodes.

Il existe trois approches principales de recherche d'ARN non-codants, plus complémentaires que concurrentes. Chacune considère la question sous un angle particulier. La recherche *ab initio* recherche des informations dans la séquence génomique discriminant la présence d'ARN non-codants. L'analyse comparative suppose que les régions non-codantes fonctionnelles sont relativement conservées parmi des organismes proches et recherche donc ces régions présents dans plusieurs organismes. Une troisième voie suppose que la fonction d'un ARN non-codant est portée par sa forme tri-dimensionnelle, et que l'on connaît cette forme pour un ensemble d'ARN non-codants ayant une fonction biologique commune. Cette forme peut être représentée de façon abstraite par un objet appelé signature. L'approche consiste ici à trouver dans une nouvelle séquence génomique toutes les régions qui peuvent adopter la signature associée à une fonction.

Notre thèse s'inscrit dans la troisième démarche. Avant d'en préciser le contenu soulignons deux aspects concernant le contexte général de ce travail. En premier lieu, on pourrait penser qu'une simple recherche de mots avec erreurs peut convenir. Cette comparaison ne fonctionne pas car la recherche de signatures est en fait un problème NP-complet [Via04]. Ensuite, considérant le nombre significatif de logiciels existant, il serait cohérent de penser que la question a déjà été traitée de manière totalement satisfaisante. Ce n'est pas tout à fait le cas.

Parmi les logiciels existants, on peut tout d'abord trouver des outils appelés « outils spécifiques », comme CITRON [LDM94], FAStRNA [EML96], tRNAscan-SE [LE97] ou snoRNA [LE99] qui recherchent uniquement les membres d'une seule famille donnée. Nous préfèrerons ici nous concentrer sur les logiciels généralistes, qui peuvent *a priori* rechercher n'importe quel ARNnc, pourvu que l'on leur donne une signature de la famille à laquelle ils appartiennent. Les outils généralistes sont en effet les seuls à pouvoir modéliser les ARNnc venant d'être découverts. Les logiciels généralistes les plus connus,

comme COVE [DEKM98], utilisent un formalisme probabiliste complexe appelé grammaire hors-contexte probabiliste [SBH+94, ED94]. Il a pour inconvénient d'être relativement lent et d'utiliser un grand nombre de paramètres à estimer et nécessite donc un grand nombre de séquences pour caractériser une signature. D'autres logiciels s'orientent plus vers une recherche de motifs systématique pour localiser une signature. On peut par exemple citer RnaMot [GHC90, LGC94], Palingol [BKV96], PatScan [DLO97] et RnaMotif [MEG+01, MC01]. Mais ces outils s'appuient sur un formalisme parfois mal défini. De plus, si certains d'entre eux attribuent un score aux solutions trouvées — ce qui est particulièrement utile si l'on a un grand nombre de solutions —, cette gestion des coûts n'est pas toujours présente, ou parfois de façon incomplète. Certains outils permettent des descriptions de signatures plus fines que d'autres : il n'est parfois pas possible de spécifier des répétitions de mots et d'établir un (G+C)% minimal dans une région donnée. Enfin, aucun des logiciels ne permet de modéliser des interactions entre deux séquences distinctes. Ces interactions sont pourtant importantes car la fonction d'un ARN est souvent d'interagir avec un autre ARN en vue de le transformer, comme c'est le cas pour certains ARN nucléolaires.

Notre but sera donc de concevoir un outil donnant un coût à chaque solution, permettant des signatures discriminantes et autorisant des interactions entre deux séquences différentes. Nous tenterons aussi d'avoir un programme au moins aussi rapide et expressif que ceux qui existent.

Pour atteindre ces objectifs, notre travail s'est appuyé sur les réseaux de contraintes pondérées. Il s'agit d'un formalisme d'intelligence artificielle étendant les réseaux de contraintes classiques, qui avaient été utilisés par Patricia Thébault dans sa thèse [Thé04] pour répondre à la même question biologique. Les réseaux de contraintes ont l'avantage de proposer un grand nombre d'algorithmes permettant de rechercher les solutions rapidement. Ce formalisme est relativement expressif et permet de décrire précisément des signatures, y compris celles qui contiennent des interactions inter-moléculaires. Mais à la différence des réseaux de contraites classiques, les réseaux de contraintes pondérées permettent en plus de gérer des coûts.

Afin de modéliser efficacement le problème de la recherche de signatures d'ARN par le formalisme des réseaux de contraintes pondérées, notre travail de thèse s'est concentré sur les points suivants.

Les réseaux de contraintes pondérées sont le plus souvent utilisés avec des techniques dites de filtrage qui, dans le cas de la recherche d'ARN, permettent de localiser rapidement les régions contenant potentiellement des ARN. Il n'existait pas de telles techniques utilisables sur un problème aussi particulier que la recherche d'ARN, où les séquences génomiques ont une taille pouvant dépasser le million. Nous avons donc défini une technique de filtrage appelée « cohérence d'arc aux bornes » et nous l'avons appliquée à notre problème [ZGS06a].

Nos recherches nous ont par ailleurs conduit à élaborer un autre algorithme de filtrage pour les réseaux de contraintes pondérées appelé « cohérence d'arc existencielle » [HLdGZ05]. Ce filtrage ne s'applique pas au cas de la recherche d'ARN non-

codants, mais il donne de bons résultats sur des réseaux plus classiques.

Un autre avantage des réseaux de contraintes pondérées est que l'on peut totalement dissocier les algorithmes de recherche de solutions proprement dits du filtrage basé sur des algorithmes de *pattern-matching*, qui recherchent un à un les éléments de signature. Il nous a donc fallu implanter un algorithme pour chaque élément de signature utilisable. Nous avons recherché dans la littérature des méthodes efficaces pour résoudre ce type de problème. Nous avons aussi élaboré d'autres algorithmes, si ceux-ci nous paraissaient plus efficaces que ceux que nous connaissions [ZGS06b].

DARN !, un logiciel relativement complet, a été développé [ZGS07]. Il permet non seulement d'effectuer des recherches d'ARN non-codants mais aussi d'associer des scores aux solutions. Il permet également de modéliser un grand nombre d'éléments de structure.

DARN ! est complété par un outil additionnel, encore dans un état préliminaire, qui peut construire automatiquement des signatures, c'est-à-dire déduire des signatures à partir de séquences-exemples entrées par l'utilisateur. Il permet en conséquence d'éviter d'entrer manuellement des signatures.

Au terme de notre travail nous avons pris soin de comparer les performances de DARN ! avec des logiciels existants. Nous avons pu montrer que l'approche était tout à fait compétitive avec les outils existants. Nous avons de plus montré que notre approche était particulièrement sensible et spécifique, notamment grâce aux nombreux éléments de structure qu'elle permet d'utiliser. Enfin, les temps de réponse de DARN ! sont particulièrement bas et permettent d'analyser des génomes entiers en quelques secondes.

Structure du manuscrit

La structure du manuscrit se nourrit à la fois de biologie moléculaire et d'informatique. Le premier chapitre présente le contexte de biologie moléculaire nécessaire à la compréhension des chapitres ultérieurs. Il insiste en particulier sur les représentations de l'ARN et la notion de signature qui est centrale dans ce travail. Différentes méthodes de localisation des ARN non-codants y sont présentées.

Parmi ces méthodes, celle qui est utilisée dans le cadre ce cette thèse est décrite en détail dans le chapitre deux. Il s'agit de la recherche d'occurrences de signature d'ARN. Le chapitre décrit les méthodes, formalismes et algorithmes existants. Il évalue la portée et les limites des différentes approches. Il campe les objectifs d'une approche améliorant les méthodes existantes en montrant les exigences d'expressivité, de souplesse et de rapidité que doit satisfaire l'outil développé dans le cadre de la thèse. Par expressivité nous entendons la capacité à modéliser les structures complexes, par souplesse la possibilité de spécifier simplement ce que l'on recherche.

Le chapitre trois détaille les travaux de D. Gilbert et P. Thébault qui ont utilisé le formalisme des réseaux de contraintes. Les apports et les limites de ces travaux sont étudiés. L'absence de gestion de coûts dans leur modèle nous a conduit à utiliser un formalisme étendant les réseaux de contraintes appelé réseaux de contraintes pondérées. La modélisation de la recherche d'ARN sous ce dernier formalisme est présentée ; elle

est la base de la spécification de notre outil DARN !. Ce chapitre décrit également les fonctions de coûts globales pour les textes génomiques que nous utilisons pour modéliser les éléments de structure.

Comme les réseaux de contraintes pondérées ne disposaient pas de techniques de filtrage adéquates et comparables à celles des réseaux de contraintes, il a fallu combler ce manque. Le chapitre quatre présente la « cohérence d'arc aux bornes » qui est la technique de filtrage que cette thèse a élaborée. Il donne notamment les complexités théoriques des algorithmes présentés et propose diverses spécialisations de la cohérence d'arc aux bornes. Cette technique est au cœur de la conception et du développement de DARN !. Dans ce chapitre, nous présentons également un autre algorithme de filtrage appelé « cohérence d'arc existentielle ». Ce filtrage ne s'applique pas directement au cas de la recherche d'ARN, mais donne de bons résultats sur des réseaux plus classiques.

Alors que la cohérence d'arc aux bornes concerne la recherche de solutions proprement dite, les techniques de *pattern-matching* permettent de rechercher les éléments de structure des ARN non-codants. Le chapitre cinq présente les algorithmes et les diverses optimisations qui sont implantés dans notre outil.

Le chapitre six complète les précédents en présentant les caractéristiques d'un outil additionnel de construction automatique de signature. Ce module construit une signature à partir d'un alignement donné en entrée pour générer une signature compréhensible par DARN !. Ce chapitre développe les méthodes et les choix effectués pour définir les éléments de structure et générer une signature pertinente. Cet outil est encore dans un état préliminaire.

Les résultats pratiques ainsi que les performances de DARN ! et du module d'apprentissage sont présentés dans le chapitre sept. Celui-ci montre les capacités de nos logiciels en termes de sensibilité, de spécificité et de rapidité. L'apport de la cohérence d'arc aux bornes est évalué par rapport à la propriété de cohérence locale la plus utilisée, à savoir la cohérence d'arc. Des comparaisons avec les performances d'autres logiciels sont données. Les tests montrent que la possibilité de modéliser un grand nombre d'éléments de structure (comme le duplex et les interactions non canoniques) permet d'obtenir une meilleure spécificité.

Au terme de ce manuscrit, après avoir dressé un bilan de notre approche, nous proposerons quelques perspectives de travaux futurs en vue d'améliorer le logiciel dans l'espoir qu'il puisse être encore plus utile à la communauté biologiste et bio-informaticienne.

Chapitre 1

Contexte biologique

Ce premier chapitre présente le sujet sur lequel nous allons travailler : l'ARN. Nous ne ferons que survoler le sujet, car les détails des éléments biologiques sont beaucoup trop complexes pour pouvoir être résumés ici. Nous présenterons simplement les notions qui nous seront utiles pour la compréhension des chapitres ultérieurs. Nous commencerons par donner quelques informations élémentaires sur la nature chimique, physique et biologique de l'ARN. Nous tenterons de définir ce qu'est un ARN non-codant, et nous décrirons quelques-uns de ces ARN. Dans une seconde partie, nous exposerons les approches bio-informatiques de recherche d'ARN, qui sont la recherche *ab initio*, l'analyse comparative et la recherche de signatures.

1.1 L'ARN

Ici sont présentées les principales propriétés de l'ARN. Le livre [Lew97] pourra être consulté pour toute notion qui n'aura pas été abordée ici.

1.1.1 Présentation générale

L'ARN, ou acide ribonucléique, est une molécule composée d'une succession de quatre nucléotides : l'adénine, la cytosine, la guanine et l'uracile, respectivement notées A, C, G et U. Un ARN a un *sens*, et il est habituellement représenté de l'extrémité dite 5' à l'extrémité dite 3', comme sur la figure 1.1.

Chaque nucléotide est composée d'un sucre, d'un pont phosphate qui le lie au prochain nucléotide, ainsi que d'une base azotée, qui représente la seule partie variable. On distingue trois côtés à une base azotée, comme représenté sur la figure 1.2 : le côté Watson-Crick, le côté Hoogsten, et le côté sucre.

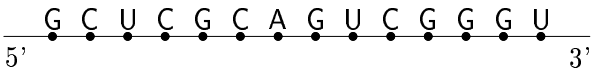


FIG. 1.1 – Représentation simple d'un brin d'ARN orienté.

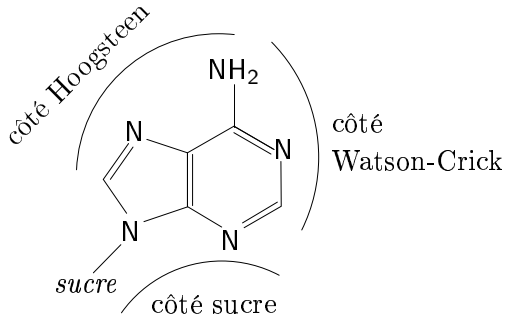


FIG. 1.2 – Les trois côtés de l'adénine.

L'ARN se trouve en général sous forme monocaténaire, c'est-à-dire sous forme d'un brin unique, et sa taille excède rarement les quelques milliers de nucléotides. Un nucléotide d'ARN peut développer des interactions avec d'autres nucléotides de la même molécule (la liaison est appelée « intra-moléculaire ») ou d'une autre (la liaison est alors « inter-moléculaire »). Les plus fréquentes sont les liaisons Watson-Crick (G–C et A–U) ou *wobble* (G–U). Mais un nucléotide peut théoriquement s'apparier avec un autre selon ses trois côtés, et selon deux orientations différentes ; l'orientation est dite *cis* si les axes des deux brins d'ARN interagissant ont le même sens, elle est dite *trans* dans le cas contraire. En revanche, pour des raisons de conformation dans l'espace, toutes les interactions ne sont pas équivalentes. On peut toutefois dégager, pour chaque type d'interaction, des classes d'interaction « géométriquement » semblables et considérées comme équivalentes. Ces classes d'interactions équivalentes sont consignées dans les *matrices d'isostéricité* (voir tableau 1.1 et [LSW02]). Les liaisons Watson-Crick habituelles (G–C et A–U) sont en fait des liaisons Watson-Crick–Watson-Crick en *cis*. On peut vérifier sur le tableau correspondant que ces liaisons appartiennent à la même famille d'isostericité (famille 1), ce qui confirme que ces liaisons sont interchangeables.

Le *k-turn* [LLMW05] (voir figure 1.3) est une structure présente dans de nombreux ARNnc (comme dans les petits ARN à boîte C/D ou H/ACA, présentés peu après). Cette structure contient de nombreuses liaisons non-canoniques. La signalétique employée dans la figure 1.3 pour désigner les côtés interagissant dans une interaction non canonique est donnée sur les matrices du tableau 1.1. Les ronds désignent des interactions du côté Watson-Crick, les triangles, du côté sucre, les carrés, du côté Hoogsteen ; les symboles noirs désignent des interactions en *cis*, les symboles blancs désignent des interactions en *trans*. Nous verrons par la suite que les interactions non-canoniques s'avèrent importantes pour modéliser correctement certaines structures d'ARNnc.

Enfin, afin de simplifier l'analyse des données ARN, l'union internationale de chimie pure et appliquée (IUPAC) a défini un ensemble de nucléotides *ambigus*, qui désignent non pas un seul nucléotide, mais un ensemble de nucléotides. Ceux-ci sont souvent utilisés lorsqu'il existe une incertitude sur la nature d'un nucléotide. Par exemple, le nucléotide R désigne le nucléotide A ou G. Cet alphabet de nucléotides ambigus est donné dans le tableau 1.2 (le nucléotide X ne fait pas partie de la norme IUPAC, mais nous l'emploierons par la suite). Ces nucléotides ambigus sont utilisés dans de nombreux logiciels et se révèlent utiles pour désigner une incertitude sur la nature d'un nucléotide.

●

Watson-C. \ Watson-Crick *cis*	A	C	G	U
A	4	2	3	1
C	2	6	1	5
G	3	1		2
U	1	5	2	6

○

Watson-C. \ Watson-Crick *trans*	A	C	G	U
A	4	3		1
C	3	6	2	5
G		2	4	3
U	1	5	3	6

●-■

Watson-C. \ Hoogsteen *cis*	A	C	G	U
A			3	3
C		2	1	1
G	3		4	
U	1		1	2

○-□

Watson-C. \ Hoogsteen *trans*	A	C	G	U
A	4		4	
C	2	1	2	
G			5	4
U	1		3	2

●-▶

Watson-C. \ Sucre *cis*	A	C	G	U
A	1	1	1	1
C	2	2	2	2
G	3	3	5	3
U	4	4	4	4

○-▷

Watson-C. \ Sucre *trans*	A	C	G	U
A	1	1	1	1
C	1	1	1	1
G		2		2
U	3	3	4	3

■

Hoogsteen \ Hoogsteen *cis*	A	C	G	U
A			2	
C			1	
G	2	1	1	
U				

□

Hoogsteen \ Hoogsteen *trans*	A	C	G	U
A	1	1	2	2
C	1		1	2
G	2	1	3	
U	2	2		

■-▶

Hoogsteen \ Sucre *cis*	A	C	G	U
A	$\frac{1}{2}$	1	2	1
C	1	$\frac{1}{2}$	1	$\frac{1}{2}$
G	1		1	
U	2	1	$\frac{1}{2}$	1

□-▷

Hoogsteen \ Sucre *trans*	A	C	G	U
A	1	1	1	1
C	1	1		1
G			2	
U	2		2	

▶

Sucre \ Sucre *cis*	A	C	G	U
A	1	1	1	1
C	1	1	1	1
G	1	1	1	1
U	1	1	1	1

▷

Sucre \ Sucre *trans*	A	C	G	U
A	1	1	1	1
C				
G	2	2	2	2
U				

TAB. 1.1 – Les tables d'isostéricité des différents types d'interactions. $\frac{1}{2}$ indique que l'interaction est rencontrée dans les classes 1 et 2, un blanc indique qu'une telle interaction ne semble pas exister.

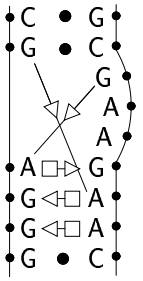


FIG. 1.3 – Un *k-turn* [LLMW05].

Symbole	Signification
R	A ou G
Y	C ou U
W	A ou U
S	C ou G
M	A ou C
K	G ou U
B	C ou G ou U
D	A ou G ou U
H	A ou C ou U
V	A ou C ou G
N	A ou C ou G ou U
X	aucun

TAB. 1.2 – Symboles des nucléotides ambigus.

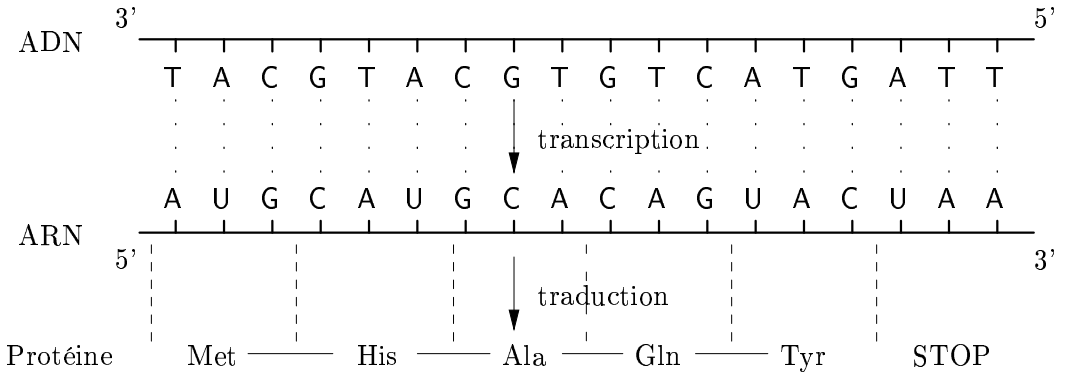


FIG. 1.4 – Mécanismes de transcription et traduction.

1.1.2 Fonctions de l'ARN

L'ARN a souvent été considéré comme un *messager* entre l'ADN, qui stocke l'information génétique, et les protéines, qui représentent les molécules actives. De fait, cette fonction est assurée par l'ARN messager (ARNm), dans les processus de *transcription* et *traduction*, comme indiqué sur la figure 1.4.

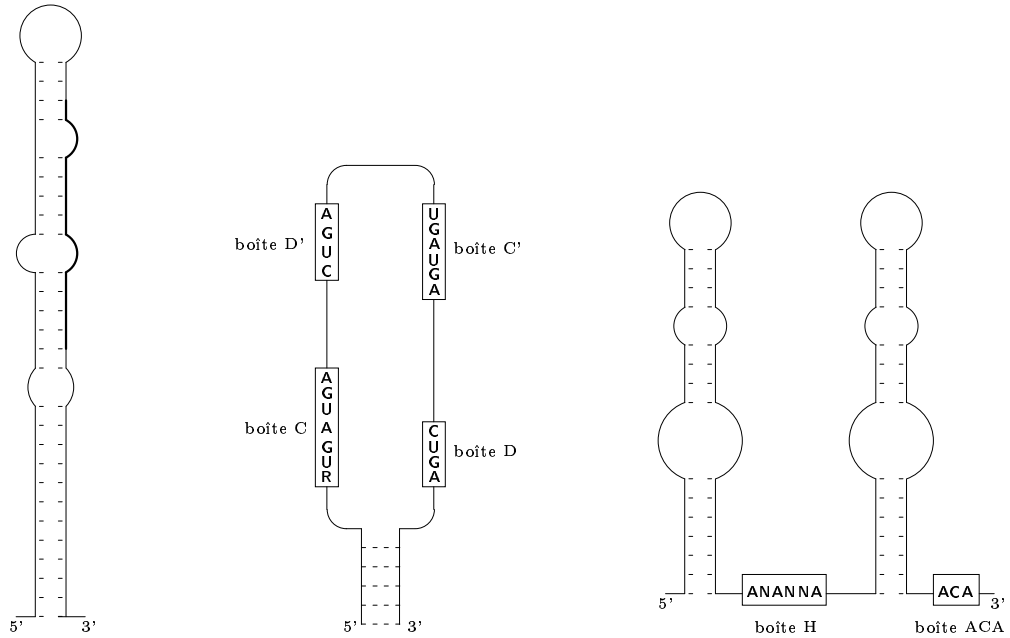
La transcription désigne l'élaboration d'un ARN dit *messager* (ARNm) à partir d'une région d'ADN que l'on nomme *gène*. L'ADN est aussi une molécule composée de nucléotides, mais où l'uracile a été remplacée par la thymine, les deux nucléotides jouant le même rôle dans les interactions Watson-Crick. Lors de la transcription, les deux molécules de l'ADN s'écartent. Des nucléotides d'ARN viennent interagir pour former des liaisons Watson-Crick avec l'un des brins, et s'unir entre eux pour créer ainsi peu à peu un ARNm. Ainsi, à partir d'un gène, on peut déduire l'ARNm synthétisé. Réciproquement, si l'on connaît un ARNm, alors on pourra rechercher quel gène l'aura synthétisé.

La traduction est la synthèse d'une protéine à partir d'un ARNm. Les nucléotides sont alors pris par trois (un triplet de nucléotides est appelé *codon*), et à chaque triplet correspond un acide aminé. Le nombre de triplets possibles est $4^3 = 64$, le nombre d'acides aminés est 20. Plusieurs codons différents peuvent synthétiser le même acide aminé.

Il existe aussi de nombreux ARN dits *ARN fonctionnels* ou *non-codants* (ARNnc), puisqu'ils ne sont pas traduits en protéines. Les plus anciennement connus sont les *ARN de transfert* (ARNt) et les *ARN ribosomiques* (ARNr), qui interviennent dans les mécanismes de traduction.

Récemment [CNB96, BSF96, LFA99], de nouveaux types d'ARN ont été découverts. Nous en présenterons deux grandes familles :

- les micro ARN (miARN ; la figure 1.5(a) représente un miARN précurseur, d'où sera extrait le miARN, délimité en trait plus gras) [Ruv01],
- les petits ARN nucléolaires (pARNno, ou snoRNA pour *small nucleolar RNA*) à boîte C/D (voir figure 1.5(b)) ou H/ACA (voir figure 1.5(c)) [BCH02].



(a) micro ARN

(b) petit ARN nucléolaire à boîte C/D

(c) petit ARN nucléolaire à boîte H/ACA

FIG. 1.5 – Quelques ARN non-codants.

Les récentes découvertes de nouvelles fonctions assurées par les ARN [Edd01] ont relancé la recherche de ces molécules et afin de trouver ces ARNnc, de nouveaux formalismes doivent être proposés pour modéliser ces nouvelles connaissances.

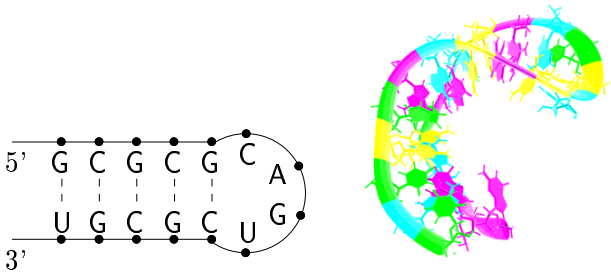
Il s'avère que la fonction d'un ARN est le plus souvent portée par sa forme tridimensionnelle. Cette forme n'étant en général pas connue entièrement, on a recours à des représentations appelées « structures » qui modélisent la forme d'un ARN. Les paragraphes suivants décrivent les différents types de représentations d'un ARNnc.

1.1.3 Représentations et familles d'ARN

Il existe trois représentations principales de l'ARN, que nous présenterons de la moins informative à la plus informative. Ces représentations nous permettrons d'appréhender la notion de famille d'ARN.

1.1.3.1 La structure primaire

Il s'agit simplement de la séquence orientée d'ARN, où les nucléotides sont représentés par des lettres (voir figure 1.1). Aucune interaction n'apparaît. Un ensemble de nucléotides adjacents est appelé un *mot*.



(a) Représentation secondaire

(b) Représentation tertiaire

FIG. 1.6 – Différentes représentations d'un même ARN

1.1.3.2 La structure secondaire

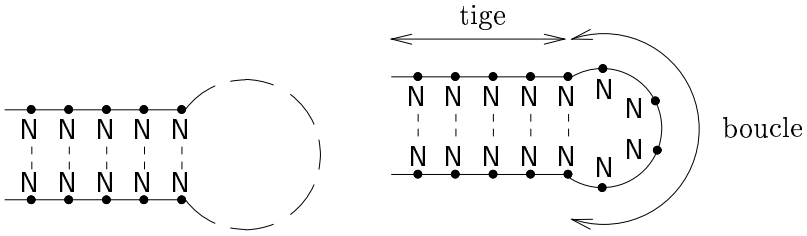
La structure secondaire représente la molécule d'ARN avec ses interactions, de sorte que la figure peut être représentée dans un plan, comme sur la figure 1.6(a). Il est d'usage de rapprocher les nucléotides qui s'apparient entre eux, si bien que la séquence n'apparaît plus sous forme linéaire, mais sous une forme plus complexe appelée *repliement* en structure secondaire.

Dans cette représentation, la structure secondaire décompose en général un ARN en *éléments de structure*. Par exemple, un ensemble d'appariements consécutifs est appelé une *hélice* (voir figure 1.7(a)), et les nucléotides situés entre les deux brins de l'hélice constituent une *boucle* d'hélice. Si la boucle contient peu de nucléotides, on parle de *tige-boucle* pour nommer l'hélice et sa boucle, comme sur la figure 1.7(b). Une hélice peut aussi contenir des nucléotides non appariés. Si ces nucléotides sont sur un seul brin, la boucle est appelée *renflement* ; si ces nucléotides sont situés sur les deux brins, en vis-à-vis, elle est appelée *boucle interne* (voir figure 1.7(c)). Il est aussi possible qu'une hélice se sépare en deux hélices distinctes, comme sur la figure 1.7(d). Il se crée alors entre les différentes hélices une *boucle multiple*.

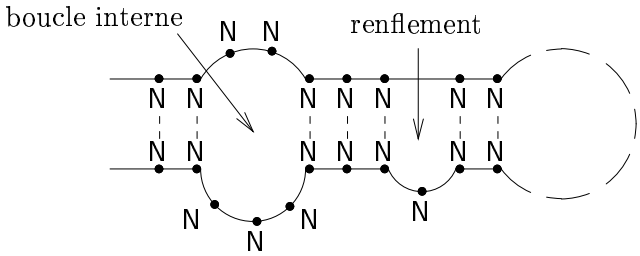
La structure secondaire permet aussi de représenter les interactions qui ont lieu entre deux molécules d'ARN différentes. Un ensemble d'interactions faisant participer deux molécules différentes est appelé un *duplex*, comme représenté sur la figure 1.7(e).

1.1.3.3 La structure tertiaire

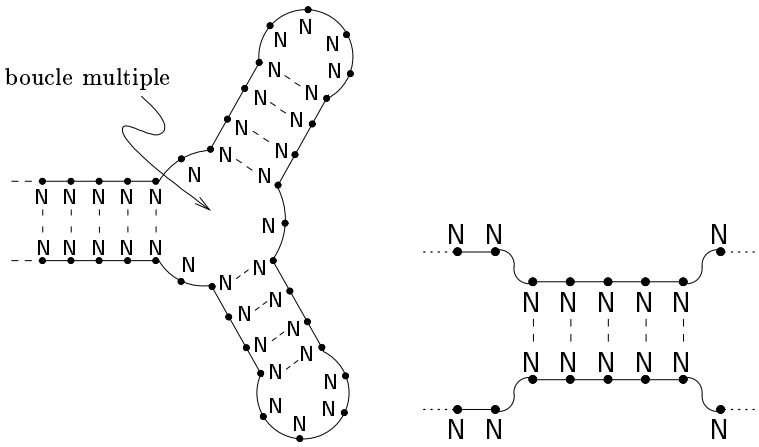
La structure tertiaire donne toutes les interactions connues dans la molécule. C'est en général une forme en trois dimensions (voir figure 1.6(b)), qui contient éventuellement des triplets d'interactions (c'est-à-dire des nucléotides qui interagissent avec deux autres nucléotides simultanément), des pseudo-nœuds (voir figure 1.8(a)), des tiges-boucles embrassées (voir figure 1.8(b)), etc.



(a) Simple hélice (b) Tige-boucle

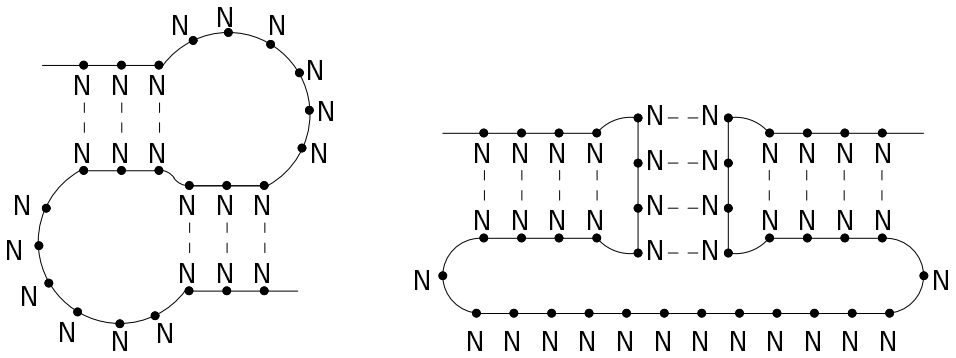


(c) Hélice dégénérée



(d) Boucle multiple (e) Duplex

FIG. 1.7 – Différents éléments de structure secondaire.



(a) Pseudo-nœud

(b) Tiges-boucles embrassées

FIG. 1.8 – Différents éléments de structure tertiaire.

1.1.3.4 Les familles d'ARN non-codant

Une *famille* d'ARN est un ensemble d'ARN possédant une fonction biologique commune. Différents membres d'une même famille d'ARN peuvent se retrouver dans des organismes totalement différents. Les ARNt, ARNr et pARN sont autant d'exemples de familles d'ARN que l'on retrouve aussi bien chez les eucaryotes que les archées.

Il est généralement admis que la fonction biologique d'un ARNnc est liée à sa structure tertiaire. Une famille peut donc être caractérisée et représentée par l'ensemble des éléments des trois types de structure. Un *alignement* est la représentation de plusieurs membres d'une même famille où les éléments de séquences conservées sont alignés en tenant compte de la conservation de la structure. À titre d'exemple, les figures 1.9(a) et 1.9(b) donnent une représentation de la structure de l'ARNt ainsi qu'un alignement de plusieurs séquences d'ARNt.

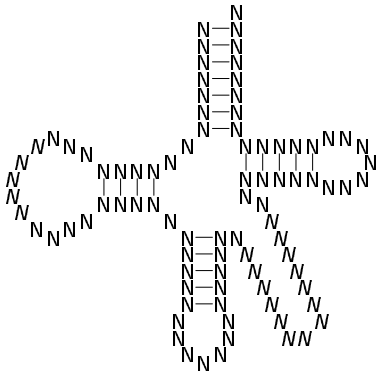
Enfin, on appelle *signature* l'ensemble des propriétés des structures primaire, secondaire et tertiaire qui discriminent une famille d'ARN. Notre hypothèse de travail sera la suivante :

> *Si une séquence génomique respecte la signature d'une famille donnée, alors elle doit vraisemblablement appartenir à cette famille d'ARN et posséder une fonction identique.*

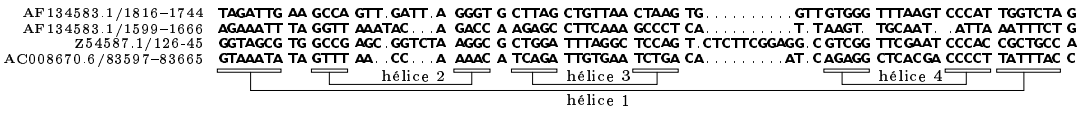
1.1.4 Modèle thermodynamique et structure secondaire

L'analyse thermodynamique de l'ARN étudie sa stabilité. Nous le verrons par la suite, la thermodynamique est à la base de nombreuses approches de recherche d'ARN non-codants.

L'analyse thermodynamique d'une molécule d'ARN permet de calculer une estimation de son énergie libre à partir de la connaissance de sa structure secondaire. Une



(a) Une représentation de la structure secondaire de la famille ARNt — les nucléotides en italique peuvent être optionnels



(b) Un alignement de plusieurs ARNt de l'homme ; la structure est donnée par RFAM

FIG. 1.9 – Représentation de la structure et alignement d'ARNt.

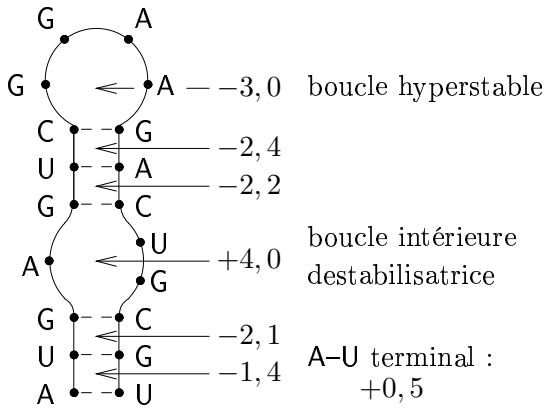


FIG. 1.10 – Calcul de l'énergie libre d'un repliement.

molécule a, par nature, tendance à adopter une structure lui conférant une énergie libre la plus basse possible. On dit qu'un repliement est *stable* si l'énergie de la molécule atteint un minimum. La recherche sur les ARN s'intéresse en général aux repliements stables, car les molécules non stables ont une durée de vie relativement éphémère et donc une activité réduite.

Les facteurs principaux de stabilisation d'une molécule d'ARN sont les liaisons hydrogène entre deux nucléotides au sein d'un appariement (les interactions G–C étant plus stabilisatrices que les A–U, elles-mêmes plus stabilisatrices que les G–U), et les interactions de Van der Waals entre deux nucléotides consécutifs, souvent situés dans une hélice, traduisant une énergie d'empilement.

De plus, les éléments de structure secondaire tels que la boucle, le renflement et la boucle interne ont tendance à déséquilibrer la molécule, l'élévation d'énergie augmentant avec la taille de l'élément de structure (sauf pour le cas des tétraboucles, les boucles contenant quatre nucléotides, qui sont plus stables que les triboucles).

Les données existantes[1] considèrent en général à part les hélices et les autres éléments de structure. Dans les hélices, les énergies de liaison et les énergies d'empilement sont en pratique indissociables. Les interactions sont donc prises deux par deux et l'on attribue aux quadruplets de nucléotides la somme des énergies de liaison et d'empilement. Un score global, fonction du nombre de nucléotides qui les composent, est donné aux boucles, renflements et boucles internes (voir figure 1.10). Toutefois, certaines boucles de quatre nucléotides particulièrement stables, nommées *tétraboucles hyperstables*, reçoivent un score plus bas. La stabilité de ces boucles est due à la présence d'interactions non-canoniques à l'intérieur même de ces boucles (voir figure 1.11).

Il existe deux derniers types d'interactions, qui, même si on les rencontre plus rarement, sont relativement stables : les *triplex* qui sont des triplets d'interactions (que l'on retrouve dans le *k-turn*, voir figure 1.12(a)), et les *quadruplex* [WH07], des quadruplets

[1] Accessibles sur http://www.bioinfo.rpi.edu/~zukerm/rna/energy/.

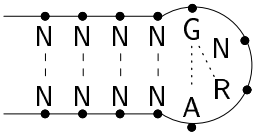
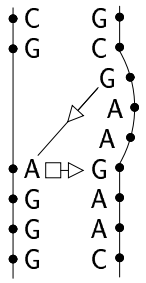
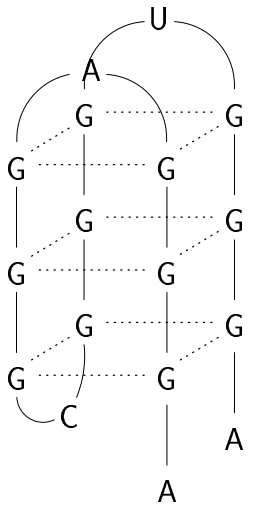


FIG. 1.11 – Une tige-boucle contenant la tétraboucle hyperstable GNRA. Les pointillés représentent les interactions non-canoniques dans la boucle qui stabilisent la structure.



(a) Le nucléotide A du *k-turn* s'apparie simultanément avec deux autres bases, pour former un triplex.



(b) Un ensemble quadruplex de guanines, formant une structure stable [WH07].

FIG. 1.12 – Deux dernières structures stables.

d'interaction (voir figure 1.12(b)).

1.2 Recherche bio-informatique des ARN

L'ARNomique est la science qui s'intéresse à la recherche de nouveaux ARNnc. Nous tenterons ici de décrire les grandes lignes des méthodes utilisées en bio-informatique pour découvrir ces ARN, et nous présenterons des logiciels s'appuyant sur ces méthodes. Notre but n'est évidemment pas de décrire dans le détail chacun des logiciels actuellement utilisés, mais de présenter l'idée sous-jacente et l'originalité des principales méthodes.

D'une manière générale, l'inférence et la recherche des ARNnc peuvent être décrites de trois manières différentes.

1.2.1 Recherche *ab initio*

La méthode *ab initio* a pour but de rechercher des ARNnc sans connaissance *a priori* de la structure, mais s'appuie sur des propriétés générales de ces types d'ARN. Nous nous appuierons pour ce paragraphe sur les exposés de [Sch03] et [Fon05].

Cette approche utilise l'information thermodynamique donnée sur une séquence. En effet, les structures secondaire et tertiaire portent une grande partie de la fonction biologique d'un ARNnc. Les ARNnc doivent donc être thermodynamiquement stables pour être actifs, et l'hypothèse de la recherche *ab initio* est que l'énergie libre d'un ARNnc est généralement relativement plus basse que la partie non fonctionnelle. Une information pouvant discriminer la présence d'un ARNnc est l'énergie libre. Une première possibilité pour trouver des ARNnc consiste à estimer l'énergie de toutes les sous-séquences d'ARN possibles, issues d'une séquence d'ADN, et de ne retenir que celles ayant une énergie particulièrement basse [LCB+88].

Il existe aussi quelques informations plus simples à chercher, et qui renseignent sur la stabilité d'une molécule. Les structures stables contiennent un grand nombre d'interactions, et les interactions les plus stables sont les liaisons G–C. Le taux de nucléotides C et G, appelé (G+C)% renseigne donc sur la stabilité de la molécule. L'hypothèse est ici que le (G+C)% est plus fort dans un ARNnc qu'ailleurs. On parle alors du *biais de composition*, qui permettrait de discriminer un ARN [Sch02a]. Cette méthode est très efficace dans les génomes d'archées hyperthermophiles riches en AT.

Il existe également le *biais de composition en dinucléotides*. Ce biais calcule la sous- ou sur-représentation des couples de dinucléotides (ApA, ApC, ApG, etc.) côte à côte dans la séquence. On fait ici l'hypothèse que la distribution en dinucléotides donne une indication sur l'énergie d'empilement d'une séquence [BWRvdP04].

Une autre information est la présence de motifs structuraux. La présence de tétraboucles hyperstables, par exemple, peut suggérer l'existence d'un ARNnc [CDH01].

La dernière information que l'on peut prendre en compte est la présence de signaux spécifiques que l'on trouve vers le début ou la fin d'un gène d'ARNnc. Ainsi, des motifs appelés *promoteurs* tels que TTGACA ou TATAAT sont présents chez *Escherichia coli* (une bactérie de la flore intestinale) et sont reconnus par le mécanisme de transcription comme des marqueurs situés en amont d'un gène (dans notre cas, à environ 35 et 10 nucléotides avant le début de l'ARNnc) [AHV+01]. Cette méthode a donné de bons résultats sur *Escherichia coli* mais elle est en général peu discriminante. D'une part ces mots sont courts et donc susceptibles de se trouver à de nombreux endroits dans la séquence et d'autre part, ils sont souvent dégénérés, c'est-à-dire que l'on trouve souvent des mots légèrement modifiés.

L'inconvénient majeur de ce type d'approche est qu'il est parfois difficile de trouver une information *ab initio* capable de détecter de manière relativement fiable des ARNnc. De plus, il semble que les paramètres de ces méthodes (comme la valeur-seuil du (G+C)%) dépendent fortement de l'organisme. C'est pour cela que cette approche est rarement utilisée seule, même si elle donne parfois de bons résultats.

1.2.2 Analyse comparative

Alors que l'approche *ab initio* ne considère qu'une seule séquence à la fois, l'analyse comparative utilise toujours plusieurs séquences. Elle suppose que si une région est fonctionnelle, alors la pression de sélection sur cette région sera plus forte, et celle-ci sera plus conservée que les régions non fonctionnelles qui l'entourent. Cette approche a pour objectif de retrouver les régions qui possèdent des structures primaire ou secondaire similaires. La définition de la « similarité », ainsi que les méthodes cherchant à trouver ces régions similaires varient selon les approches. On peut toutefois trouver quatre grandes méthodes. Nous nous appuierons dans cet exposé sur [GG04] et sur [Tou05].

La première méthode consiste à employer une adaptation de l'algorithme de Sankoff [San85], qui cherche à aligner la structure primaire et la structure secondaire de plusieurs séquences simultanément. Cet algorithme étant trop coûteux en temps pour être employé sur de longues séquences (la complexité est sexique par rapport à la taille des séquences à aligner), les approches se basant sur cet algorithme ont recours à des simplifications. Par exemple, Foldalign [GHS97] et Dynalign [MT02] supposent que l'utilisateur a trouvé les régions d'intérêt sur plusieurs séquences et prédisent une structure secondaire commune, sachant que Dynalign restreint le nombre de séquences à deux, mais prend en compte des données thermodynamiques. Carnac [TP04], quant à lui, utilise cette approche sur des régions qu'il a lui-même mises en évidence, autour de certaines tiges-boucles.

Une deuxième option consiste à utiliser en première passe un outil d'alignement multiple de séquences primaires. Les régions similaires alignées sont alors conservées et on utilise ensuite un deuxième outil, trouvant l'éventuelle structure secondaire commune des séquences alignées. À la différence de l'option précédente, l'alignement trouvé par les outils d'alignement de structure primaire n'est pas remis en cause dans la deuxième étape. On suppose ensuite qu'une nouvelle séquence appartient à la famille considérée si elle peut adopter la structure secondaire préalablement découverte. Bien évidemment, ce type d'outil n'est utilisable que si les structures primaires des séquences sont conservées. Les logiciels ClustalW [THG94] et M-Coffee [WOHN06] sont des exemples d'outils permettant l'alignement selon la structure primaire. On peut ensuite utiliser Pfold [KH99] ou RNAalifold [HFS02] pour trouver une structure commune aux sous-séquences conservées.

Une troisième approche prédit une structure secondaire de toutes les séquences, puis tente de trouver des structures secondaires conservées dans toutes les séquences. La méthode procède en deux temps. Des logiciels comme RNAfold [HFS+94] et Mfold [ZS81] calculent la structure secondaire d'une séquence, et RNA forester [HTGK03] ainsi que Marna [SB05] donnent les candidats. Le point faible est ici que la structure secondaire d'une séquence est en général difficilement prédictible.

Il existe enfin quelques outils qui prédisent directement l'existence d'ARN non-codants pour un ensemble de séquences. Les méthodes utilisées sont ici diverses. RNAz [WHS05] considère tout d'abord des alignements de familles connues, puis apprend quelques caractéristiques de ces alignements : énergie libre, taux de conservation de nucléotides dans les alignements, nombre de séquences dans l'alignement. RNAz classe ensuite les

séquences données en entrée en « ARNnc » ou « non ARNnc ». QRNA [RE01, RKJE01] considère deux séquences et les aligne selon trois modèles possibles utilisant les grammaires hors-contexte probabilistes (que nous présenterons dans le paragraphe 2.2.2.1) : un modèle codant, un modèle ARN, un modèle par défaut. Le modèle le plus probable, indiquant l'éventuelle présence d'un ARNnc, est ensuite donné à l'utilisateur.

1.2.3 Recherche de membres d'une famille d'ARN connue

À la différence des approches précédentes, la recherche de membres d'une famille d'ARN suppose une hypothèse forte : il existe une structure discriminante de la famille (l'ARNt est un bon exemple, car il possède quatre hélices qui lui confèrent une forme de trèfle). Le but de cette approche est alors de trouver de nouvelles régions d'une séquence donnée pouvant se conformer à la structure de la famille d'ARN. Cette approche fera l'objet de plus longs développements dans le chapitre suivant.

1.3 Conclusion

Nous avons vu que la recherche d'ARN non-codants pouvait suivre trois approches possibles : la recherche *ab initio*, l'analyse comparative et la recherche de membres d'une famille d'ARN. Ces approches étant en fait complémentaires, la meilleure méthode de recherche d'ARNnc consiste certainement à combiner ces trois approches. Néanmoins, considérant l'ampleur du travail, il semble difficile de les exploiter simultanément. sRNAPredict [LFDW05] et sRNAPredict2 [LBLW06] sont sans doute les premiers programmes se servant conjointement les trois approches. Mais, dans la pratique, le logiciel est un pipe-line utilisant notamment les logiciels BLAST [AGM$^+$90] et RNAMotif [MEG$^+$01] (des outils de recherche de signature respectivement en séquence et en structure que nous décrirons dans le chapitre suivant), QRNA [RE01, RKJE01] et TransTerm [EKW$^+$00] (un outil de recherche *ab initio* de terminateurs de transcription). L'exemple plaide ici plutôt pour la création de logiciels spécialisés dans chaque approche, mais connectables entre eux.

Dans le cadre de cette thèse, nous tenterons donc d'explorer la dernière voie, notamment dans le but d'autoriser une description plus fine des structures de familles d'ARN. Notre intrusion dans l'approche comparative se limitera à l'élaboration d'un module d'apprentissage de signature, que nous évoquerons plus longuement dans la partie 7. Celui-ci, étant donné un alignement d'ARNnc et sa structure conservée, sera capable d'en extraire une signature, et de l'exprimer en termes compréhensibles par notre logiciel de recherche d'ARN. La conception de ces outils a d'ailleurs entre autres été guidée par la volonté de les rendre utilisables dans des processus plus complexes de recherche d'ARNnc, comme sRNAPredict2, se servant de plusieurs programmes complémentaires.

Chapitre 2

La localisation de membres d'une famille d'ARN connue

2.1 Définition du problème

La localisation de membres d'une famille d'ARN dans une séquence génomique s'intéresse à la recherche de tous les ARN possédant une même fonction biologique. En supposant que la fonction biologique est donnée par la structure des ARN, il devient possible de décrire la structure conservée, appelée signature, de tous les membres de la famille.

On peut dégager deux types de méthodes principales pour cette recherche : la première est apportée par les logiciels dits *spécifiques*, la seconde, par les logiciels dits *généralistes*. Les premiers sont uniquement dédiés à la recherche d'une famille d'ARN précise et implantent la signature de cette famille. Les seconds peuvent *a priori* être utilisés pour toutes les familles et c'est à l'utilisateur de donner, à l'aide d'un langage, une signature. Chaque méthode à ses avantages et ses inconvénients : les outils spécifiques sont plus simples d'utilisation, plus rapides, plus précis et souvent plus pertinents dans les réponses qu'ils donnent, tandis que les outils généralistes permettent de rechercher différents types de familles, et d'en modifier la signature utilisée. Nous nous focaliserons ici sur la seconde approche, car il est souvent possible de spécialiser un outil généraliste, de manière à lui faire reconnaître une famille de façon spécifique. C'est par exemple le cas du logiciel généraliste COVE, spécialisé en tRNAscan-SE, très utilisé pour reconnaître les ARN de transfert.

La question que nous cherchons à résoudre est la suivante.

> *Dans cette approche, on suppose que le biologiste a une connaissance, même partielle, d'une famille d'ARNnc. La problème consiste à retrouver, dans une séquence génomique donnée, les membres d'une famille d'ARN décrite par une signature spécifiée par le biologiste.*

La recherche de membres d'une famille d'ARN pose le problème de la description des structures. Cette description doit être suffisamment précise pour discriminer une

famille d'ARN, mais elle doit pouvoir se comprendre et de modifier facilement par un utilisateur sans formation spécifique.

Nous présenterons dans la section suivante quelques programmes qui ont été développés pour répondre au problème de la recherche de membres d'une famille d'ARN. Afin d'évaluer la performance de chacun d'entre eux, on utilise souvent deux critères : la *spécificité* et la *sensibilité*. Par exemple, supposons que l'on recherche plusieurs ARNnc dans une séquence, et que l'on utilise pour cela un logiciel. Ce logiciel peut se tromper de deux manières différentes : soit il propose comme solution une sous-séquence qui s'avère ne pas être un ARNnc, il est alors dit peu spécifique, soit il « oublie » un ARNnc, il est alors dit peu sensible. Les définitions formelles sont les suivantes :

$$\text{spécificité} : \frac{VP}{VP+FP} \qquad \text{sensibilité} : \frac{VP}{VP+FN}$$

où :

- VP sont les *vrais positifs*, les solutions du logiciel qui font partie des ARNnc recherchés ;
- FP sont les *faux positifs*, les solutions du logiciel qui ne font pas partie des ARNnc recherchés ;
- FN sont les *faux négatifs*, les ARNnc recherchés qui ne font pas partie des solutions du logiciel.

2.2 Approches existantes

Les paragraphes suivants décrivent les programmes de recherche de membres d'une famille d'ARN. Deux grandes méthodes peuvent être distinguées : les approches non-probabilistes et les approches probabilistes. Suivant l'ordre historique, nous présenterons tout d'abord la première catégorie.

2.2.1 Recherches non-probabilistes

Tous les programmes non-probabilistes ont en commun d'utiliser des algorithmes de recherche de motifs développés dans les années 1970 ou 1980. Avant de présenter les logiciels proprement dits, nous décrirons rapidement les algorithmes sous-jacents.

2.2.1.1 Algorithmes

Nous donnons ici les algorithmes classiques concernant la recherche de motifs, en commençant par les motifs les plus simples : les mots.

Recherche de mots Supposons que l'on recherche le mot $M = \mathsf{ACGT}$ dans la séquence $S = \mathsf{ACGG}$. Manifestement, le mot M n'existe pas dans la séquence. En revanche, si l'on accepte une *substitution* (c'est-à-dire une transformation d'une lettre en une autre) du T en G, alors on retrouve bien le mot recherché. Une recherche autorisant quelques substitutions (dont le nombre maximal est en général donné par l'utilisateur)

est souvent préférée, car les séquences sont très rarement conservées, y compris celles qui codent pour une fonction commune.

Il existe un algorithme simple calculant le nombre minimal de substitutions entre M et S, appelé *distance de Hamming*. Il est décrit dans la fonction 1.

Fonction 1 – DistanceHamming(S : *sequence*, M : *sequence*) : entier

```
distance ← 0 ;
pour chaque i ∈ [1..|S|] faire
    si (S[i] ≠ M[i]) alors
        distance ← distance + 1 ;
retourner distance ;
```

Le modèle présenté est évidemment trop simpliste pour pouvoir être utilisé. En pratique, on autorise aussi les *insertions* et les *suppressions* de nucléotides par rapport à la séquence. On appelle *distance de Levenshtein* [Lev66] le nombre minimal de substitutions/insertions/suppressions (appelées *erreurs*). Cette distance peut se calculer à l'aide d'un algorithme de programmation dynamique classique, appelé algorithme de Needleman-Wunsch [NW70]. Dans sa forme la plus simple, il utilise une matrice m de taille $(|S|+1) \times (|M|+1)$ et chaque cellule $m[i,j]$ de la matrice stocke la distance de Levenshtein entre $S[1..i]$ et $M[1..j]$. La distance entre S et M se trouve donc dans la cellule $m[|S|,|M|]$. L'algorithme de programmation dynamique se base sur une récurrence simple sur le nombre minimum d'erreurs entre $S[1..i+1]$ et $M[1..j+i]$ qui est soit :

1. le nombre minimum d'erreurs entre $S[1..i]$ et $M[1..j]$ plus éventuellement une substitution si $S[i+1]$ et $M[j+1]$ sont différents,
2. le nombre minimum d'erreurs entre $S[1..i]$ et $M[1..j+1]$ plus une suppression correspondant à la disparition du nucléotide $S[i+1]$,
3. le nombre minimum d'erreurs entre $S[1..i+1]$ et $M[1..j]$ plus une insertion correspondant à l'apparition du nucléotide $M[j+1]$.

Les cas de base de l'algorithme de programmation dynamique correspondent au cas où $S[1..i]$ est comparé avec le mot nul ε (la distance est ici i), et au cas où $M[1..j]$ est comparé avec ε. L'algorithme est décrit par la fonction 2.

Un exemple de matrice, où $S = \mathsf{ACGTAC}$ et $M = \mathsf{ACTAA}$ est donné figure 2.1. La matrice de programmation peut aussi nous fournir un alignement de M par rapport à S, grâce à la procédure 3 (Trace). La trace d'une matrice de programmation dynamique révèle l'alignement donnant le coût optimal, comme sur la figure 2.1. Ainsi, dans l'exemple précédent, la trace est :

$$\begin{pmatrix} \mathsf{A} & \mathsf{C} & \mathsf{G} & \mathsf{T} & \mathsf{A} & \mathsf{C} \\ \mathsf{A} & \mathsf{C} & - & \mathsf{T} & \mathsf{A} & \mathsf{A} \end{pmatrix}$$

L'algorithme de Needleman-Wunsch est bien connu, et il existe plusieurs optimisations de cet algorithme. On peut tout d'abord remarquer que la récurrence de l'algorithme de programmation dynamique n'utilise que les deux dernières colonnes (ou

Fonction 2 – DistanceLevenshtein(S : *sequence*, M : *sequence*) : entier

pour chaque $i \in [0..|S|]$ **faire** $m[i,0] \leftarrow i$;
pour chaque $j \in [1..|M|]$ **faire** $m[0,j] \leftarrow j$;
pour chaque $i \in [1..|S|]$ **faire**
  pour chaque $j \in [1..|M|]$ **faire**

$$m[i,j] \leftarrow \min \left\{ \begin{array}{l} m[i-1,j-1] + \left\{ \begin{array}{ll} 0 & \text{si } M[i-1] = S[j-1] \\ 1 & \text{sinon} \end{array} \right. \\ m[i-1,j] + 1 \\ m[i,j-1] + 1 \end{array} \right\} ;$$

retourner $m[|S|,|M|]$;

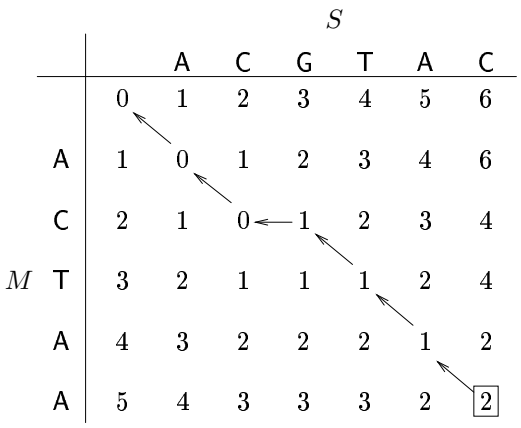


FIG. 2.1 – La matrice de programmation dynamique calculant la distance de Levenshtein entre $S = \mathsf{ACGTAC}$ et $M = \mathsf{ACTAA}$. Une trace est indiquée par les flèches, la solution est encadrée.

Fonction 3 – Trace(m : *matrice*) : chaîne de caractères

$trace \leftarrow \varepsilon$;
$i \leftarrow |S|$; $j \leftarrow |M|$;
tant que $((i,j) \neq (0,0))$ **faire**
  si $((i > 0) \wedge (m[i,j] = m[i-1,j] + 1))$ **alors**

$$trace \leftarrow trace \,\&\, \begin{pmatrix} S[i] \\ - \end{pmatrix} ;$$

  sinon si $((j > 0) \wedge (m[i,j] = m[i,j-1] + 1))$ **alors**

$$trace \leftarrow trace \,\&\, \begin{pmatrix} - \\ M[j] \end{pmatrix} ;$$

  sinon

$$trace \leftarrow trace \,\&\, \begin{pmatrix} S[i] \\ M[j] \end{pmatrix} ;$$

retourner $trace$;

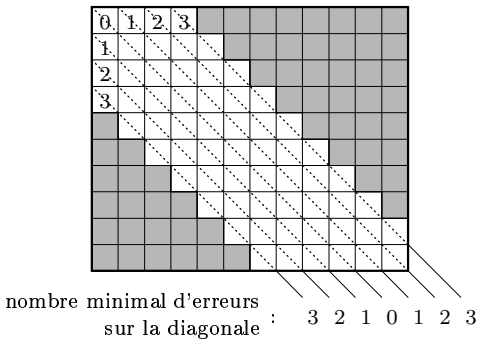


FIG. 2.2 – Matrice de programmation dynamique — l'optimisation ne calcule pas les cases grisées.

lignes) de la matrice. Il est alors possible de descendre la complexité spatiale à $\mathcal{O}(|M|)$ ou $\mathcal{O}(|S|)$. On peut utiliser alors l'agorithme de Hirschberg [Hir75] pour retrouver la trace de l'alignement.

Si l'on connaît de plus le nombre maximal d'erreurs k que l'on s'autorise, il est possible d'utiliser une autre optimisation, qui est en fait une simplification de [Ukk83]. Cette optimisation vient de l'observation selon laquelle toute cellule sur la diagonale principale peut au mieux être le coût d'un alignement sans insertion ni suppression, et donc peut donner un coût nul. Toute cellule sur une autre diagonale ne le peut pas, et donne un coût non nul (cf. figure 2.2). D'une manière générale, toute cellule située sur la i-ème diagonale reçoit un coût au moins égal à i. L'algorithme ne parcourt donc que les $2 \times k + 1$ diagonales autour de la diagonale principale.

Cette observation permet de réduire la complexité spatiale à $\mathcal{O}(k)$, et la complexité temporelle à $\mathcal{O}(|M|k)$ ou $\mathcal{O}(|S|k)$. Il existe bien d'autres optimisations, mais nous ne les détaillerons pas ici.

Il est à noter que l'algorithme de Needleman-Wunsch est aussi souvent utilisé pour savoir si la séquence M peut s'apparier avec le mot M en formant une hélice. Il suffit alors de remplacer la ligne :

$$\text{si } S[i-1] = M[j-1]$$

par :

$$\text{si } (S[i-1], M[j-1]) \in \{(\mathsf{A},\mathsf{U}), (\mathsf{U},\mathsf{A}), (\mathsf{C},\mathsf{G}), (\mathsf{G},\mathsf{C})\}$$

Il est possible d'étendre les algorithmes précédents au cas où la séquence est un alignement. Si la séquence S est composée de plusieurs séquences s_1, s_2, s_3 alignées, il est aussi possible d'utiliser l'algorithme de Needleman-Wunsch avec cet alignement. Supposons que l'on ait :

$$S = \begin{pmatrix} \mathsf{A} & \mathsf{C} & \mathsf{G} & \mathsf{T} \\ \mathsf{A} & \mathsf{G} & \mathsf{G} & \mathsf{T} \\ \mathsf{A} & \mathsf{T} & \mathsf{G} & \mathsf{A} \end{pmatrix}$$

alors, on peut poser $S = \mathsf{ABGW}$, en utilisant les nucléotides ambigus définis précédemment (voir le tableau 1.2). L'algorithme de Needleman-Wunsch ne nécessite que peu de

modifications pour prendre en compte ces nucléotides.

Le modèle présenté ici est encore loin d'être parfait, et il y manque une observation importante : S est en général beaucoup plus grand que M, et la recherche de motifs consiste en général à savoir s'il existe une ou plusieurs sous-séquences de S qui « ressemblent » à M. L'algorithme 4 (DistanceLevenshtein2), dérivé de celui de Needleman-Wunsch, permet de répondre à cette question. Il considère toutes les sous-séquences de S, et les compare avec M. En pratique, il n'y a que deux changements dans les formules de récurrence. Tout d'abord, un cas de base :

$$\forall i \in [0..|S|], m[i,0] \leftarrow 0$$

autorise l'alignement à commencer à partir de toutes les positions de S. Le cas terminal

$$distance = \min_{i \in [0..|S|]} m[i,|M|]$$

autorise l'alignement à se terminer sur tout nucléotide de S.

Fonction 4 – DistanceLevenshtein2(S : *sequence*, M : *sequence*) : entier

$distance \leftarrow \infty$;
pour chaque $i \in [0..|S|]$ **faire** $m[i,0] \leftarrow 0$;
pour chaque $j \in [1..|M|]$ **faire** $m[0,j] \leftarrow j$;
pour chaque $i \in [1..|S|]$ **faire**
  pour chaque $j \in [1..|M|]$ **faire**

$$m[i,j] \leftarrow \min \left\{ \begin{array}{l} m[i-1,j-1] + \left\{ \begin{array}{ll} 0 & \text{si } M[i-1] = S[j-1] \\ 1 & \text{sinon} \end{array} \right. \\ m[i-1,j]+1 \\ m[i,j-1]+1 \end{array} \right\} ;$$

  $distance \leftarrow \min\{distance, m[i,|M|]\}$;
retourner $distance$;

La figure 2.3 donne un exemple de matrice de programmation dynamique remplie par l'algorithme 4, où le mot GTCG est à rechercher dans la séquence CGTACGT.

Bien évidemment, il est particulièrement intéressant d'utiliser une des optimisations présentées pour l'algorithme de Needleman-Wunsch afin de réduire la complexité spatiale à $\mathcal{O}(|M|)$, au lieu du $\mathcal{O}(|S| \times |M|)$ de l'algorithme précédent.

Il existe encore quelques améliorations que l'on peut apporter à l'algorithme précédent. Tout d'abord, on peut décider que les substitutions ne doivent pas être pénalisées à la même hauteur que les insertions ou les suppressions. On peut aussi décider que toutes les substitutions ne doivent pas être pénalisées de la même manière (la substitution d'un A par un C pourrait être moins pénalisée que la substitution A par un U). Il est alors possible d'utiliser des logarithmes de probabilité pour quantifier le coût d'une substitution ([KH99, KE03] donnent des exemples de matrices de coût de substitutions).

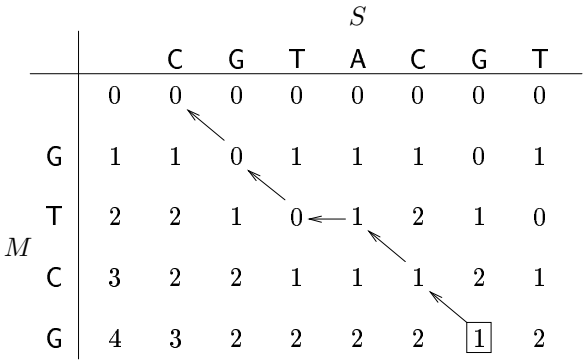


FIG. 2.3 – La matrice de programmation dynamique calculant la plus petite distance de Levenshtein entre les sous-séquences de $M = \mathsf{CGTACGT}$ et $M = \mathsf{GTCG}$. Une trace est indiquée par les flèches, la distance est encadrée.

Recherche d'expressions régulières Une possibilité pour décrire des motifs plus complexes est d'utiliser des expressions régulières :

Définition 2.1 *L'ensemble des expressions régulières sur un alphabet Σ se définit récursivement comme le plus petit ensemble respectant les propriétés suivantes :*
- *le mot vide ε est une expression régulière,*
- *chaque élément de Σ est une expression régulière,*
- *la concaténation de deux expressions régulières est une expression régulière,*
- *l'union de deux expressions régulières est une expression régulière,*
- *la fermeture transitive (c'est-à-dire la répétition un nombre arbitraire de fois, allant de zéro à l'infini) d'une expression régulière est une expression régulière.*

Il existe une syntaxe communément utilisée pour décrire une expression régulière :
- AT désigne la concaténation des lettres A et T ;
- (AT|TA) désigne l'union des expressions AT et TA, ou, en d'autres termes, indique que les mots AT et TA sont acceptés ;
- [AT] est parfois utilisé pour désigner (A|T) ;
- (AT)? est parfois utilisé pour désigner (AT|ε) ;
- (AT)* désigne la fermeture transitive de AT, c'est-à-dire la succession non bornée de mots AT ;
- (AT)$^+$ est parfois utilisé pour désigner AT(AT)*.

Les expressions régulières permettent de décrire précisément les alignements. Par exemple,

$$\begin{pmatrix} \mathsf{A} & \mathsf{C} & \mathsf{G} & \mathsf{T} & \mathsf{A} & \mathsf{C} \\ \mathsf{A} & \mathsf{C} & - & \mathsf{T} & \mathsf{A} & \mathsf{A} \end{pmatrix}$$

peut être décrit par AC(G)?TA[AC].

Les expressions régulières peuvent être recherchées en temps polynomial par rapport à la taille maximale des séquences de l'alignement grâce à un automate. Comme montré sur la figure 2.4, une séquence appartient à l'expression recherchée si l'on peut suivre un

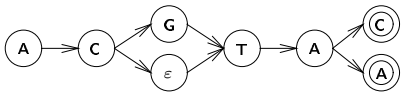


FIG. 2.4 – L'automate reconnaissant l'expression régulière AC(G)?TA[AC].

chemin partant de l'état de départ (de degré entrant nul) et finissant à un état d'arrivée (doublement entouré).

Ces expressions régulières sont très utilisées pour la recherche de motifs protéiques (cf. PROSITE [SCH+02b, dCSG+06]), mais beaucoup moins pour la recherche de motifs nucléotidiques, car elles ne peuvent pas décrire de façon satisfaisante des hélices. Il faut donc utiliser un autre formalisme, plus complexe, comme les expressions algébriques.

Recherche d'expressions algébriques Les automates sont peu à même de décrire la structure secondaire des ARNnc à cause du phénomène de mutations compensatoires. Considérons par exemple l'alignement de la figure 2.5(a). Cet alignement contient deux hélices. Dans l'hélice 1, on peut autoriser n'importe quel nucléotide à la position x_1, *à condition* que le nucléotide à la position x_2 puisse s'apparier avec le nucléotide en x_1. En d'autres termes, le nucléotide en x_1 peut muter, à condition que le nucléotide en x_2 mute aussi, de manière à conserver l'interaction Watson-Crick entre les deux positions. Les expressions algébriques permettent de capturer efficacement de phénomène de mutations compensatoires.

Les expressions algébriques peuvent se définir de la manière suivante :

Définition 2.2 *On définit un langage algébrique sur un alphabet Σ par une grammaire hors-contexte $\langle S, N, T, \delta \rangle$ où :*

- *N est un ensemble de caractères non-terminaux, dont l'intersection avec Σ est nulle ;*
- *T est un ensemble de caractères terminaux, un sous-ensemble de Σ ;*
- *S est l'axiome, un élément de N ;*
- *δ est un ensemble de règles de production, une application de N vers $(N \cup T)^*$.*

Un langage algébrique est l'ensemble des mots de terminaux qui peuvent être générés par la transformation récursive de S à la suite d'une série de transformations par les règles de production.

La grammaire représentée dans la figure 2.5(b) décrit par exemple la structure donnée figure 2.5(a). Dans la recherche de motif, on différencie les règles de transition des règles d'émission. Les règles de transition produisent généralement des non-terminaux et donnent la structure de l'ARN, tandis que les règles d'émission produisent des symboles terminaux et donnent la séquence. On distingue quatre types de non-terminaux principaux :

- le type S : ce sont les non-terminaux déterminant la structure,
- le type P : ils traduisent un appariement de nucléotides,
- le type G : ils traduisent un non-appariement de nucléotides, à gauche de l'arbre,

– le type D : ils traduisent un non-appariement de nucléotides, à droite de l'arbre.

Bien évidemment, il est possible d'ajouter des états pour signifier la présence éventuelle de brèches dans les hélices, d'insertions de nucléotides dans les séquences, etc.

Vérifier qu'une séquence appartient à une expression algébrique peut se faire en temps polynomial, grâce à un automate à pile (contenant une mémoire).

Les expressions algébriques ne permettent toutefois pas de décrire des motifs complexes, comme les pseudo-nœuds. Ces derniers ne peuvent d'ailleurs être reconnus par aucun algorithme polynomial, car trouver un pseudo-nœud est un problème NP-complet [Via04]. Certains outils présentés dans ce chapitre prennent en compte les pseudo-nœuds, mais ne s'appuient pas sur un formalisme clair. Nous verrons dans le chapitre suivant un formalisme capable de modéliser ce type de structure.

2.2.1.2 Outils

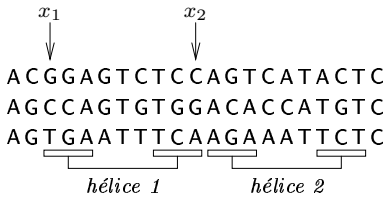
Nous présentons maintenant, par ordre chronologique, les outils utilisant des techniques évoquées dans le paragraphe précédent.

Les programmes de type FASTA et BLAST Les programmes du type FASTA [PL88] et BLAST [AGM+90] (comprenant entre autres WU-BLAST [GS93, SG94], BLAT [Ken02]) sont des outils de recherche de mots avec erreur dans une banque de données. Chaque solution se voit attribuer un score calculé grâce aux matrices de substitution, ainsi qu'une *E-value*, qui retranscrit l'espérance du nombre de solutions ayant un score au moins aussi bon (au vu par exemple de la taille du mot et de la base de données). Ainsi, une *E-value* particulièrement faible indique que la solution est exceptionnelle, pour un modèle de séquence aléatoire simple.

Dans la pratique, BLAST recherche des parties de la séquence-requête exactement conservées appelées *graines* dans la base de données. Il cherche ensuite à étendre ces occurrences exactement conservées pour aligner toute la séquence à rechercher. C'est une méthode approchée, qui peut, en de rares occasions, oublier certains candidats.

ANREP ANREP [MM93] est un logiciel permettant de reconnaître les expressions régulières dans les séquences génomiques ou protéomiques. La fermeture transitive — sans doute l'opération la moins intéressante biologiquement — n'a toutefois pas été implantée. Un système de score complet est modifiable par le biologiste : chaque substitution, insertion, suppression peut être exactement quantifiée.

RnaMot RnaMot [GHC90, LGC94] est sans doute l'un des premiers vrais outils de recherche de signatures d'ARN. La description de la signature, appelée *descripteur*, se décompose en trois parties. La première partie du descripteur donne les positions relatives des éléments de signature. Ceux-ci peuvent être des éléments de structure primaire (mots et espaceurs) ou secondaire (hélices). Dans RnaMot, les espaceurs spécifient une distance minimale et maximale entre deux autres éléments de signature. Ces espaceurs peuvent contenir des nucléotides ambigus, des nucléotides optionnels et des erreurs. Les hélices peuvent avoir des tailles de tige variables et contenir des mésappariements, des



(a) l'alignement

règles de transition			règles d'émission		
S	$\rightarrow$	$S_1 S_{10}$			
S_1	$\rightarrow$	G_2	G_2	$\rightarrow$	$\mathsf{A}W_2$
S_2	$\rightarrow$	G_3	G_3	$\rightarrow$	$\mathsf{C}W_3$
S_3	$\rightarrow$	P_4		$\mid$	$\mathsf{G}W_3$
S_4	$\rightarrow$	P_5	P_4	$\rightarrow$	$\mathsf{C}W_4\mathsf{G}$
S_5	$\rightarrow$	P_6	P_5	$\rightarrow$	$\mathsf{G}W_4\mathsf{C}$
S_6	$\rightarrow$	G_7		$\mid$	$\mathsf{C}W_4\mathsf{G}$
S_7	$\rightarrow$	G_8	P_6	$\rightarrow$	$\mathsf{A}W_5\mathsf{T}$
S_8	$\rightarrow$	G_9	G_7	$\rightarrow$	$\mathsf{A}W_7$
S_9	$\rightarrow$	ε		$\mid$	$\mathsf{G}W_7$
S_{10}	$\rightarrow$	D_{11}	G_8	$\rightarrow$	$\mathsf{T}W_8$
S_{11}	$\rightarrow$	P_{12}	G_9	$\rightarrow$	$\mathsf{C}W_9$
S_{12}	$\rightarrow$	P_{13}		$\mid$	$\mathsf{G}W_9$
S_{13}	$\rightarrow$	P_{14}		$\mid$	$\mathsf{T}W_9$
S_{14}	$\rightarrow$	G_{15}	D_{11}	$\rightarrow$	$W_{11}\mathsf{C}$
S_{15}	$\rightarrow$	G_{16}	$\ldots$		
S_{16}	$\rightarrow$	G_{17}			
S_{17}	$\rightarrow$	ε			

(b) la grammaire hors contexte

FIG. 2.5 – Une structure difficile à modéliser par une expression régulière.

renflements et des boucles internes. Il est de plus possible de spécifier la position de certains nucléotides sur une hélices et de modéliser des pseudo-nœuds.

La seconde partie du descripteur spécifie chaque éléments de signature : les éléments de structure primaire (mots et espaceurs) et les éléments de structure secondaire (hélices).

La troisième partie donne éventuellement les options du descripteur, dont certaines permettent de décrire plus finement un candidat. L'utilisateur peut par exemple choisir d'accepter les liaisons *wobble* et peut même spécifier le nombre maximum d'occurrences de ce type de liaisons dans un candidat. L'utilisateur peut aussi donner le nombre maximal de mésappariements et d'erreurs dans les hélices et les mots. Ce type d'option suggère déjà l'idée de *pénalités* données à un candidat, ainsi qu'un *seuil maximal* de pénalités.

Les auteurs ne mentionnent pas de formalisme sous-jacent à RnaMot, mais ils utilisent quelques techniques d'intelligence artificielle. L'utilisateur doit spécifier chaque élément de la signature, et l'outil effectue une recherche récursive arborescente. Informellement, une recherche arborescente suit le fonctionnement suivant. Elle commence par rechercher une position possible pour le premier élément de signature, puis recherche une position possible pour le deuxième élément de signature sachant que le premier a été placé, etc. Lorsque tous les éléments de signature ont été placés, on a une solution. S'il est impossible de placer le n-ième élément, ou que toutes les positions possibles de celui-ci ont été trouvées, alors on revient au $n-1$-ème élément, on lui trouve la prochaine position possible, et on continue la recherche.

RnaMot propose un dernier type d'option : l'ordre de recherche des éléments de signature. Les concepteurs avaient remarqué que, pour une signature donnée, commencer par rechercher un élément de structure dont la probabilité d'apparition était très faible accélère notamment la recherche des solutions. Il est donc implanté dans RnaMot une heuristique d'ordonnancement des éléments de structure. Cela suppose qu'il est possible d'estimer la probabilité d'apparition d'un élément de structure. C'est en pratique difficile, mais des hypothèses simplificatrices, ainsi que des coefficients de correction empiriques, permettent de trouver un estimateur donnant un ordonnancement souvent bon [Laf93]. De plus, il faut être capable d'inférer les positions relatives des éléments de structure les uns par rapport aux autres. Supposons par exemple que l'on doive trouver une signature composée d'un mot, puis d'une hélice, puis d'un mot. L'ordre de recherche des éléments de la signature peut suggérer de rechercher tout d'abord le premier mot, puis le second, et enfin l'hélice. Le programme trouve alors une position possible pour le premier mot, et tente de trouver une position possible pour le second mot. Il faut bien évidemment pour cela s'aider du fait que le premier mot a été placé. Dans l'implantation de RnaMot, dès lors qu'un élément à été placé, il est possible d'inférer un intervalle de placement pour chacun des autres éléments de structure.

RnaMot propose aussi deux autres mécanismes. Le premier est la recherche de *solutions optimales*. Il n'est en effet pas rare que RnaMot propose deux solutions qui traduisent un seul et même candidat. On peut par exemple penser à une recherche donnant deux candidats dont la seule différence est qu'un élément de structure du second candidat est décalé d'un nucléotide vers la droite. RnaMot détecte donc que deux solu-

tions sont en conflit lorsque les positions de début de ces deux solutions sont les mêmes. Lorsque deux solutions sont en conflit, il calcule pour chacune d'elle un score qui prend en compte la taille des hélices (les plus longues sont avantagées), une estimation simple de l'énergie des hélices, le nombre de mésappariements dans les hélices, et la présence de *mots optionnels* (qui ne discriminent pas une solution, mais améliorent le score d'une solution). Parmi les solutions en conflit, seule celle ayant le meilleur score est affichée, et les autres sont stockées dans un fichier alternatif. En revanche, le nombre d'erreurs dans les mots et le nombre de *wobble* n'est pas pris en compte dans le score d'une solution.

Un dernier mécanisme mis en place par RnaMot est la gestion des séquences de N. Pour différentes raisons, il est courant d'observer, dans les séquences génomiques que l'on peut télécharger, de longues répétitions du nucléotide N (pouvant être de l'ordre de quelques milliers), traduisant une incertitude du séquençage de la région. Recherchée telle quelle, et puisque que le nucléotide N traduit « A ou C ou G ou U », toute signature doit trouver au moins une occurrence dans une séquence de N. C'est rarement ce qui est voulu par l'utilisateur. Ce problème a été résolu de la manière suivante : si l'occurrence d'un élément de structure est tout entier dans une région de N, alors la solution est rejetée ; en revanche, si cette occurrence contient au moins un nucléotide qui n'est pas N, alors la solution est gardée.

Palingol Les auteurs de Palingol [BKV96] sont partis du principe que RnaMot n'était pas suffisamment expressif pour décrire des signatures particulièrement fines. Le but est ici d'offrir à l'utilisateur une boîte à outils complète pour spécifier une signature, tout en le soulageant de la partie technique du codage : analyse syntaxique du fichier d'entrée, recherche des mots en miroir-complément, etc. Les auteurs s'inscrivent ouvertement dans la logique de l'intelligence artificielle : l'utilisateur spécifie ce qu'il cherche, non la manière dont il le cherche. Palingol se présente donc comme un langage fonctionnel. Les éléments de structure sont spécifiés par des prédicats, ou *contraintes*, donnés par l'utilisateur.

Le descripteur se présente donc sous la forme de déclarations spécifiant chaque élément de structure. Une première partie du descripteur spécifie les hélices, qui sont les principaux éléments de structure utilisés par Palingol. On peut contraindre la taille des brins de ces hélices, ou la taille des boucles, et on peut aussi spécifier qu'un mot doit être retrouvé dans une hélice. L'utilisateur peut alors utiliser des opérateurs (comme $\leq$, $\neq$, ou le « et » logique) ainsi que des variables qu'il a créées, pouvant porter sur des positions, des longueurs, ou des nombres d'erreurs. La deuxième partie du descripteur donne les distance entre les hélices. La troisième partie spécifie les contraintes globales, portant sur plusieurs hélices. Tous ces outils permettent de décrire des signatures particulièrement complexes. Il est possible par exemple de spécifier une signature du type : « s'il n'existe pas d'hélice ici, alors on doit avoir le mot ACGT, et l'on donne au candidat une pénalité ».

Le logiciel peut aussi exploiter une connaissance plus fine de la structure primaire, donnée par une *matrice de poids position-spécifique* (dont le fonctionnement exact sera expliqué plus loin). Cette matrice considère un alignement de séquences, comptant souvent un petit nombre de nucléotides, et donne, pour chaque colonne de l'alignement, la proportion d'apparition de chaque nucléotide. Étant données une matrice de poids et

une séquence (de même taille que l'alignement), il est donc possible de donner un score à cette séquence, en fonction de l'adéquation avec la matrice. On peut ainsi spécifier une matrice de poids à Palingol, et demander qu'un mot dépasse un score minimal donné par rapport à cette matrice.

Cette qualité d'expressivité a comme contrepartie une certaine complexité du descripteur. La forme générale de celui-ci, bien que familière aux utilisateurs de LISP, n'en reste pas moins particulièrement difficile à appréhender, même pour des signatures simples.

Dans le détail, Palingol procède comme suit. Tout d'abord, toutes les hélices possibles (contenant des liaisons Watson–Crick sans erreur) de la séquence donnée en entrée sont systématiquement recherchées, avant même la recherche de la structure proprement dite. Ces hélices sont ensuite stockées pour une utilisation ultérieure. Ensuite, Palingol prend chaque contrainte dans l'ordre dans lequel l'utilisateur les a spécifiées et il essaie de les satisfaire.

PatScan PatScan [DLO97] est un outil du même type que RnaMot. Une caractéristique particulièrement intéressante de PatScan est qu'il peut être utilisé aussi bien sur les chaînes nucléotidiques que peptidiques.

Le descripteur de PatScan, sans doute plus simple que celui de RnaMot puisqu'il peut tenir en une seule ligne, est principalement composée de mots et d'espaceurs. Une amélioration importante de PatScan, peut-être empruntée à Palingol, réside dans le fait qu'une sous-séquence peut-être nommée, et utilisée comme variable. Lorsque l'on nomme une sous-séquence (de taille donnée), il est alors possible de spécifier que celle-ci doit se retrouver plus loin dans la séquence, ce qui permet de modéliser les répétitions. On peut aussi rechercher une sous-séquence conjointement avec son miroir-complément (pour retrouver une hélice), ou même une sous-séquence avec son miroir. Pour chaque élément de structure, PatScan demande de spécifier le nombre maximal d'insertions, de suppressions et de substitutions.

PatScan permet aussi de spécifier des *alternatives* : une signature peut par exemple contenir soit une hélice de taille 5, soit le mot ACGT. Comme Palingol, il propose à l'utilisateur les matrices de poids position-spécifique, et il donne la possibilité de spécifier la somme des tailles de sous-séquences nommées.

S'il veut voir apparaître certaines interactions non-canoniques, l'utilisateur doit tout d'abord spécifier une ou plusieurs règles d'appariement, dont chacune contient tous les appariements concernés (par exemple, {A–U, C–G, G–C, U–A} pour les liaisons Watson–Crick). Puis, pour chaque hélice, l'utilisateur donne une des règles d'appariement. Cette méthode semble très générale, dans la mesure où l'utilisateur peut spécifier n'importe quel type d'interaction.

PatScan est un outil particulièment expressif : il permet de modéliser des répétitions, des hélices, des mots en miroirs, des interactions non-canoniques,etc. Mais concernant la gestion des erreurs, le choix paraît discutable. En effet, le biologiste raisonne plus souvent en termes de nombre d'erreurs (une erreur étant soit une insertion, soit une suppression, soit une substitution), et si l'on spécifie que l'on veut une insertion, ou une suppression, ou une substitution, on peut s'attendre à une solution contenant les trois

erreurs. De plus, PatScan ne permet pas de mettre des pénalités globales.

RnaMotif RnaMotif [MEG+01, MC01] se présente comme une extension de RnaMot, où un système de scores a été ajouté. Ce système, à la différence de Palingol, peut s'écrire sous la forme d'un script inspiré de la syntaxe AWK. Un descripteur de RnaMotif se décompose en quatre parties.

La première partie spécifie certains paramètres de recherche, notamment la prise en compte ou non des liaisons *wobble*.

La deuxième partie donne la signature proprement dit. RnaMotif offre la possibilité à l'utilisateur de modéliser les pseudo-nœuds, des types d'interactions différents pour chaque hélice, l'interdiction de mésappariements aux bords des hélices, et la fraction de nucléotides non-appariés d'une hélice (plutôt que le nombre de nucléotides non-appariés), ce qui se révèle particulièrement utile lorsque la taille de l'hélice est peu contrainte. De plus, les éléments de séquence peuvent être décrits par des expressions régulières. Enfin, trois derniers éléments de structure sont aussi modélisables : les répétitions, les triplex et les quadruplex.

La troisième partie spécifie des interactions supplémentaires. Cette partie sert par exemple à spécifier des interactions non-canoniques ou des interactions de structure ternaire, et elle permet d'imposer un type d'interaction dans des hélices (par exemple pour les *k-turn*).

La dernière partie spécifie les coûts associés à un candidat. RnaMotif propose pour cela tout un langage pour décrire des structures de coût particulièrement complexes. L'utilisateur peut utiliser des stuctures algorithmiques (**if**, **for**), ainsi que des propriétés des éléments de structure (taille d'une hélice, nombre de mésappariements et surtout calcul de l'énergie d'une région) pour donner un coût. Le défaut de cette méthode (que l'on retrouve chez Palingol) est que la spécification d'un coût requiert toujours une certaine habileté, même pour des coûts simples. Par exemple, spécifier que le coût d'une solution est donné par le nombre de mésappariements des hélices n'est pas une chose facile.

L'algorithme de recherche de motifs de RnaMotif est relativement original. Les éléments de structure sont stockés dans une structure arborescente (plus précisément, une forêt), où deux éléments sont frères si l'un est après ou avant l'autre, et père/fils si l'un est imbriqué dans l'autre (la boucle d'une hélice est ainsi considérée comme une séquence fille de l'hélice). Cette structure permet d'estimer simplement le placement des éléments de structure, puisque les positions relatives de deux éléments de la forêt sont toujours comparables : soit l'un est imbriqué dans l'autre, soit l'un est après l'autre. Dans la pratique, RnaMotif commence la recherche par les mots les plus longs et les moins ambigus. Tous les candidats sont recherchés et affichés. RnaMotif s'accompagne en général d'un script Perl qui permet d'enlever tous les candidats sous-optimaux.

En revanche, la spécification des interactions supplémentaires casse la structure arborescente des éléments de structure et c'est pourquoi une interaction n'est vérifiée que lorsque les éléments de structure contenant cette interaction ont été trouvés. Il en est de même pour le calcul du score : celui-ci n'intervient qu'à la fin, lorsque tous les éléments de structure ont été trouvés. Ceci constitue sans doute une faiblesse de l'approche, dans

la mesure où le calcul du score durant la recherche pourrait aider à arrêter l'exploration beaucoup plus tôt. Une dernière limitation est que les renflements et boucles intérieures ne sont pas autorisées dans les hélices.

2.2.2 Méthodes probabilistes

L'ajout de probabilités aux méthodes précédentes permet d'attribuer un degré de confiance aux solutions données par une approche. Une probabilité faible traduit par exemple qu'une solution ne s'accorde que très peu au modèle donné. Les paragraphes suivant décrivent les modèles probabilistes utilisés dans la localisation de membres de familles d'ARNnc.

Comme dans la partie précédente, nous commencerons par décrire les algorithmes utilisés par les logiciels probabilistes, en donnant les équivalents probabilistes des algorithmes évoqués dans la partie précédente. Le lecteur pourra se référer à [DEKM98] pour de plus amples détails sur les méthodes probabilistes.

2.2.2.1 Modèles probabilistes et algorithmes

Recherche de mots L'ajout de probabilités aux algorithmes de base (comme Needleman-Wunsch) est relativement naturelle, si l'on veut pouvoir donner à l'utilisateur une valeur de vraisemblance indiquant à quel point un mot M « ressemble » à une sous-séquence d'une grande base de données. C'est encore plus intéressant si l'on a modélisé une famille d'ARNnc par un alignement A. On définit alors dans A les fréquences d'apparition des nucléotides pour chaque position. A est appelée une matrice de poids position-spécifique.

$$M : \begin{pmatrix} \mathsf{A} & \mathsf{C} & \mathsf{G} & \mathsf{T} \\ \mathsf{A} & \mathsf{G} & \mathsf{G} & \mathsf{T} \\ \mathsf{A} & \mathsf{T} & \mathsf{G} & \mathsf{A} \end{pmatrix}$$

A :

A	100 %	0 %	0 %	33 %
C	0 %	33 %	0 %	0 %
G	0 %	33 %	100 %	0 %
T	0 %	33 %	0 %	67 %

Il est maintenant possible de savoir la probabilité que M appartienne à l'alignement. Il suffit pour cela de multiplier les probabilités d'apparitions de chaque nucléotide de M entre eux. Par exemple, la probabilité que $M = \mathsf{ATGA}$ corresponde à l'alignement est de $100\ \% \times 33\ \% \times 100\ \% \times 33\ \% \approx 11\ \%$. Les matrices de poids modélisent une distribution de probabilité sur les mots. Dans la pratique, on préfère utiliser des logarithmes de probabilités, car ceux-ci s'additionnent (au lieu de se multiplier), ce qui permet d'éviter des problèmes d'*underflow*, dans la mesure où les probabilités peuvent arriver à des valeurs extrêmement proches de zéro.

Les matrices de poids position-spécifique n'ont en revanche pas la possibilité de modéliser efficacement les insertions et les suppressions dans les alignements. Il faut pour cela utiliser les chaînes de Markov cachées.

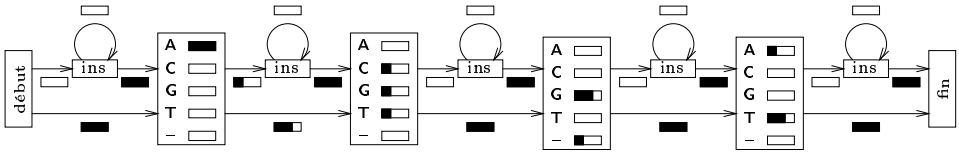


FIG. 2.6 – Une chaîne de Markov — les états sont les grands rectangles, les probabilités d'émissions sont donnés par les petits rectangles (plus ou moins noirs), les transitions sont matérialisées par les flèches et les probabilités de transition étiquettent ces flèches.

Recherche d'expressions régulières probabilistes Les équivalents probabilistes des automates sont les chaînes de Markov cachées (CMC, ou *HMM* en anglais pour *hidden Markov model*) [BP66]. Ce formalisme contient aussi des états pouvant émettre des nucléotides, ainsi que des transitions permettant de passer d'un état à un autre. Mais ici, il existe une distribution de probabilités d'émission de nucléotides pour chaque état, et une distribution de probabilités de transition. Ainsi, pour l'alignement suivant :

$$\begin{pmatrix} \mathsf{A} & \mathsf{A} & \mathsf{C} & \mathsf{G} & \mathsf{T} \\ \mathsf{A} & - & \mathsf{G} & \mathsf{G} & \mathsf{T} \\ \mathsf{A} & - & \mathsf{T} & - & \mathsf{A} \end{pmatrix}$$

on peut définir la chaîne de Markov représentée figure 2.6. Cette chaîne de Markov est simpliste, dans la mesure où par exemple toute séquence présentant un nucléotide A dans la troisième position de l'alignement aura une probabilité de vraisemblance nul, ce qui n'est sans doute pas ce que l'on souhaite. On peut alors introduire une connaissance *a priori* sur les probabilités d'émission et de transition, prenant en compte un bruit inhérent à toute séquence génomique.

L'algorithme de Baum-Welsch [BPSW70] permet d'estimer les distributions de probabilité d'une CMC à partir d'un alignement. Si l'on veut connaître la probabilité qu'une séquence donnée corresponde à la CMC, on peut utiliser l'algorithme *forward-backward* [BE67], et l'algorithme de Viterbi [Vit67] donne le meilleur alignement de cette séquence par rapport à la CMC.

Recherche d'expressions algébriques probabilistes Les grammaires hors-contexte probabilistes (GHCP, ou *SCFG* pour *stochastic context-free grammars*) [SBH+94], parfois appelées *modèles de covariance* [ED94] (car elles permettent de modéliser les mutations compensatoires), sont elles aussi beaucoup utilisées pour la détection d'ARNnc. Ces GHCP sont des grammaires hors-contexte, où chaque règle de transitions à partir d'un non-terminal est étiquetée par une distribution de probabilité. La grammaire donnée figure 2.7 décrit par exemple la structure donnée figure 2.5(a).

De plus, la probabilité qu'une séquence corresponde à une GHCP donnée peut être calculée grâce à l'algorithme *inside-outside* [LY90] et l'algorithme CYK [CS70, You67, Kas65] donne le meilleur alignement de cette séquence par rapport à la GHCP. Ces algorithmes ont une complexité temporelle quartique en le nombre de nucléotides de l'alignement, et linéaire en la taille de la séquence. La complexité spatiale ne dépend

règles de transition				règles d'émission			
S	$\rightarrow$	$S_1 S_{10}$	(100 %)				
S_1	$\rightarrow$	G_2	(100 %)				
S_2	$\rightarrow$	G_3	(100 %)	G_2	$\rightarrow$	$\mathsf{A}W_2$	(100 %)
S_3	$\rightarrow$	P_4	(100 %)	G_3	$\rightarrow$	$\mathsf{C}W_3$	(33 %)
S_4	$\rightarrow$	P_5	(100 %)		$\mid$	$\mathsf{G}W_3$	(67 %)
S_5	$\rightarrow$	P_6	(100 %)	P_4	$\rightarrow$	$\mathsf{C}W_4\mathsf{G}$	(100 %)
S_6	$\rightarrow$	G_7	(100 %)	P_5	$\rightarrow$	$\mathsf{G}W_4\mathsf{C}$	(67 %)
S_7	$\rightarrow$	G_8	(100 %)		$\mid$	$\mathsf{C}W_4\mathsf{G}$	(33 %)
S_8	$\rightarrow$	G_9	(100 %)	P_6	$\rightarrow$	$\mathsf{A}W_5\mathsf{T}$	(100 %)
S_9	$\rightarrow$	ε	(100 %)	G_7	$\rightarrow$	$\mathsf{A}W_7$	(33 %)
S_{10}	$\rightarrow$	D_{11}	(100 %)		$\mid$	$\mathsf{G}W_7$	(67 %)
S_{11}	$\rightarrow$	P_{12}	(100 %)	G_8	$\rightarrow$	$\mathsf{T}W_8$	(100 %)
S_{12}	$\rightarrow$	P_{13}	(100 %)	G_9	$\rightarrow$	$\mathsf{C}W_9$	(33 %)
S_{13}	$\rightarrow$	P_{14}	(100 %)		$\mid$	$\mathsf{G}W_9$	(33 %)
S_{14}	$\rightarrow$	G_{15}	(100 %)		$\mid$	$\mathsf{T}W_9$	(33 %)
S_{15}	$\rightarrow$	G_{16}	(100 %)	D_{11}	$\rightarrow$	$W_{11}\mathsf{C}$	(100 %)
S_{16}	$\rightarrow$	G_{17}	(100 %)	...			
S_{17}	$\rightarrow$	ε	(100 %)				

FIG. 2.7 – Une grammaire hors-contexte probabiliste.

pas de la taille de la séquence, mais elle est cubique en le nombre de nucléotides de l'alignement.

Du fait d'une limitation inhérente aux grammaires hors-contexte, les GHCP ne peuvent pas décrire les pseudo-nœuds ou des interactions de structure tertiaire. Le formalisme a alors été étendu pour prendre en compte certains pseudo-nœuds ([RE99, CMW03, MSS05] proposent trois extensions des GHCP répondant à cette question). Tous les détails concernant les CMC et les GHCP sont clairement expliqués dans [DEKM98].

2.2.2.2 Outils

Nous présenterons ici quelques outils de recherche d'ARNnc utilisant les formalismes présentés dans le paragraphe précédent.

COVE, Infernal et RSearch COVE [ED94] se présente comme un outil très général utilisant les GHCP (ou modèles de covariance). Les résultats en matière de sensibilité et de spécificité du modèle sont souvent très bons, au prix d'une complexité temporelle élevée.

Un outil particulièrement connu, tRNAscan-SE [LE97] utilise COVE pour rechercher les ARN de transfert (ARNt) dans des organismes de différents règnes. Le programme COVE a été entraîné sur des alignements de nombreux ARNt connus, mais dans la pratique, il semble être trop lent pour pouvoir être utilisé sur un génome. C'est pour

cela qu'il utilise en première passe deux autres logiciels, Eufindt et tRNAscan 1.3, avec des paramètres relâchés. COVE est ensuite utilisé sur les solutions des deux précédents programmes pour donner ses candidats, assortis d'un score de vraisemblance.

Les GHCP sont aussi utilisées pour trouver des petits ARN nucléolaires à boîte C/D chez la levure, grâce à Snoscan [LE99]. Pour répondre aux problèmes de rapidité, le logiciel couple les formalismes probabilistes avec une méthode gloutonne, recherchant spécifiquement les sites très conservés chez ces ARN.

COVE, renommé Infernal, sert aussi de logiciel de base permettant la recherche d'ARNnc sur le site RFAM [GJBM+03, GJMM+05]. Ce site recense un grand nombre d'ARNnc connus, classés par famille. Un alignement est ainsi constitué pour chaque famille, et l'on en tire une grammaire probabiliste. Il est ensuite possible de rechercher si un élément de cette grammaire probabiliste existe avec une probabilité suffisante dans une séquence fournie par l'utilisateur. Il est à noter que, pour des raisons de lenteur, BLAST est utilisé comme première passe à une requête, et Infernal est lancé sur toutes les solutions préalablement trouvées par BLAST.

Un dernier outil utilise les GHCP : il s'agit de RSearch [KE03], présenté comme l'équivalent de BLAST pour les ARNnc. Celui-ci prend un ARN avec sa structure secondaire, et permet de savoir si une base de données contient un ARN similaire. Pour cela, une grammaire est constituée à partir de l'ARN connu. La grammaire est légèrement relâchée pour que des insertions, des suppressions et des substitutions soient autorisées. En revanche, au moment où l'article est apparu (2003), une requête usuelle nécessitait plusieurs heures de calculs sur une centaine de CPU.

Erpin Erpin [GL01, LG03, LLG05] est une synthèse entre les approches non-probabilistes et probabilistes. Comme dans le premier type d'approche, les éléments de structure sont séparés en régions « hélices » et « simples brins », et leur contenu est respectivement analysé par des matrices de poids position-spécifiques (ce que signifie qu'insertions et suppressions ne sont pas autorisées dans les hélices) et des chaînes de Markov cachées. Ce découpage en éléments de structure analysés séparément permet tout d'abord de donner des résultats en un temps raisonnable, mais aussi de se libérer de la structure arborescente imposée par les GHCP puisque Erpin accepte par exemple les pseudo-nœuds.

En théorie, Erpin prend en entrée un fichier d'alignement où la structure secondaire (et éventuellement tertiaire) est indiquée. Il recherche un à un les éléments de structure dans la séquence donnée par l'utilisateur, puis donne les résultats assortis d'une *E-value* [LLFG05]. En pratique, Erpin est accessible *via* une page Web, qui recense un grand nombre d'alignements, prêts à l'utilisation, qui ont été conçus par différents spécialistes des ARNnc [LFL+04].

2.3 Conclusion

Comme on le voit, la recherche de signatures peut s'effectuer selon plusieurs formes. Nous allons donc éclaircir les choix que nous avons effectués lors de la création de notre

logiciel. Nous voulons tout d'abord permettre à l'utilisateur de spécifier une signature aussi caractéristique que possible. Nous devrons pour cela permettre de modéliser un grand nombre d'éléments de structure. Il devra ensuite être possible de créer et de modifier simplement une signature (comme par exemple accepter une erreur supplémentaire dans une hélice) ; la signature devra pour cela être lisible et compréhensible. Enfin, peu d'expertise en algorithmique devra être requise.

Ces choix éliminent *a priori* les approches synthétisant leurs propres signatures à partir des alignements : comment spécifier par exemple que l'on souhaite relâcher les contraintes sur la taille d'une hélice, sans ajouter des « fausses » séquences dans l'alignement ? Souvent, modifier la signature générée par le logiciel à partir de l'alignement n'est pas une solution non plus. Dans le cas des GHCP, cette signature est composée de probabilités, qui sont en nombre trop important pour être véritablement lisibles et modifiables par un utilisateur non expert (ajouter un état manuellement est par exemple une affaire délicate). De plus, le nombre de probabilités des approches probabilistes, généralement très grand par rapport au nombre de séquences données en entrée, amène souvent un problème de spécialisation. Snoscan [LE99], qui contient des CMC et des GHCP, a été entraîné à reconnaître des pARNno à boîte C/D chez la levure ; il donne malheurement de mauvais résultats sur un autre organisme (même relativement proche).

Nos buts nous orientent vers une signature spécifiée « en clair » par l'utilisateur, et donc vers un formalisme proposé par l'intelligence artificielle. Comme certains des logiciels précédents, nous allons considérer que les éléments de structure à retrouver sont des *contraintes*, qu'il faut chercher à satisfaire. Cette approche a été suivie dans le cadre de la thèse de Patricia Thébault [Thé04], et a donné lieu à plusieurs articles [TdGSG05, TdGSG06]. Il s'est avéré que ce formalisme permettait aussi de combler une lacune présentée par tous les autres logiciels : la spécification d'interactions inter-moléculaires. C'est d'autant plus important que certains ARNnc, comme les pARNno à boîte H/ACA et C/D n'ont pas une structure primaire et secondaire suffisamment caractéristique pour pouvoir être mise en évidence avec une spécificité acceptable ; prendre en compte l'appariement avec une molécule telle que l'ARNr est capitale.

Nous voulons aussi éviter les spécifications de coûts, pénalités ou scores compliqués ; il nous faudra donc un formalisme capable de modéliser ces scores simplement. Enfin, nous voulons que notre logiciel puisse décrire des motifs complexes, contenant éventuellement des pseudo-nœuds ; le modèle devra donc être au moins NP-complet.

C'est pour toutes ces raisons que notre choix s'est porté sur les *réseaux de contraintes pondérées* comme formalisme de représentation et de résolution.

Chapitre 3

Modélisation

3.1 Introduction

Dans ce chapitre, nous présentons l'approche développée dans le cadre de cette thèse répondant à la question de la localisation de signatures d'ARN. Rappelons que celle-ci consiste à retrouver dans une séquence génomique tous les membres d'une famille d'ARN donnée à partir de sa signature. On suppose pour cela que, premièrement, la famille est caractérisée de façon suffisamment précise par ses structures primaire, secondaire et tertiaire. Deuxièmement, on suppose que l'utilisateur peut décrire l'ensemble, appelé *signature*, des éléments conservés dans les membres de la famille.

Nous avons vu, dans le chapitre 2, comment plusieurs logiciels avaient répondu à cette question. Nous avons ainsi pu mettre en évidence certains éléments que l'on souhaitait trouver dans un outil de recherche d'ARN, et nous avons vu qu'aucun logiciel ne semblait parfaitement correspondre à nos attentes. Nous souhaitons pouvoir exprimer facilement une signature, composée éventuellement d'éléments comme des appariements entre deux séquences différentes ou des interactions non-canoniques. Nous voulons aussi pouvoir utiliser des coûts pour exprimer des préférences. Nous voulons aussi pouvoir enlever les solutions redondantes. Nous allons voir ici comment les réseaux de contraintes pondérés permettent une modélisation naturelle de ce problème.

Historiquement, les réponses faisant intervenir des formalismes proches des réseaux de contraintes éventuellement pondérées sont plutôt rares. Parmi eux, il existe Palingol [BKV96], qui s'appuie sur une logique formelle, décrite dans le paragraphe 2.2.1.2. Une équipe autour de David Gilbert [BG95, EGJ$^+$01] a aussi mené plusieurs travaux préliminaires faisant intervenir les réseaux de contraintes pour résoudre le problème de la localisation de signatures d'ARN. Par la suite, les travaux de thèse de Patricia Thébault [Thé04] ont abouti à la conception d'un cadre formel rigoureux et performant, ainsi qu'à un logiciel nommé MilPat utilisant les réseaux de contraintes. Ces travaux ont donné plusieurs résultats intéressants sur la recherche de nouveaux ARN non-codants [TdGSG06]. Notre thèse, en utilisant les réseaux de contraintes pondérées, s'inscrit dans le prolongement de ces développements.

Dans le reste du chapitre, nous décrirons tout d'abord les éléments constitutifs des

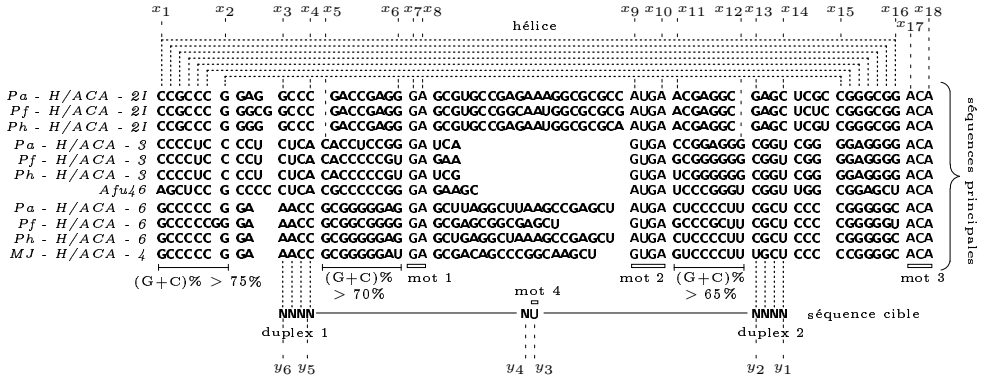


FIG. 3.1 – Un alignement de petits ARN à boîte H/ACA dans différents génomes d'archées.

structures d'ARN, comme les mots, les hélices, les duplex, etc. Nous définirons les réseaux de contraintes, et nous présenterons la modélisation et les travaux de D. Gilbert et de P. Thébault, qui ont eu recours à ce formalisme pour rechercher les signatures d'ARN. Nous présenterons ensuite les réseaux de contraintes pondérées et l'intérêt qu'ils peuvent représenter par rapport aux autres cadres. Nous expliciterons le modèle que nous choisirons, et nous dresserons la liste des éléments de structure que nous proposerons à l'utilisateur. Nous terminerons ce chapitre par quelques perspectives sur des extensions possibles de la modélisation.

3.2 Éléments de signature

Le but de nos recherches est de développer un outil capable de trouver toutes les *occurrences* d'une *signature* dans une séquence donnée. Une signature est ici l'ensemble des éléments structurels conservés dans tous les membres d'une même famille et une occurrence d'une signature est une séquence respectant les propriétés de la signature. Afin de construire la signature d'une famille, une possibilité consiste à rassembler le plus grand nombre possible de membres de cette famille, et de les *aligner*. Nous avons déjà présenté un alignement d'ARN de transfert dans le premier chapitre, sur la figure 1.9(b). Un alignement peut être vu comme un tableau où une ligne est un membre de la famille d'ARN considérée et une colonne représente éventuellement un élément de structure conservé. Dans ce chapitre, nous utiliserons l'alignement de la figure 3.1, qui rassemble quelques petits ARN à boîte H/ACA.

Au vu de l'alignement de la figure 3.1, connaissant les éléments de structure primaire, secondaire et tertiaire vus au chapitre 1, et en nous aidant des travaux de modélisation des logiciels vus dans le chapitre 2, il semble naturel de décomposer la signature des pARN à boîte H/ACA en éléments structurels. Deux éléments au moins viennent à l'esprit : le mot et l'hélice, qui sont utilisés dans la plupart des logiciels de recherche de signatures d'ARN. Nous allons dans les paragraphes suivants dresser une liste des éléments qui peuvent être utiles pour décrire une famille.

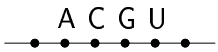


FIG. 3.2 – Un exemple de mot ACGU.

3.2.1 Le mot conservé

Le mot peut être défini comme une succession de nucléotides contigus. C'est un élément de structure primaire. Une représentation en est donnée figure 3.2. Dans certaines familles d'ARN, un mot peut avoir un rôle particulier et on observe alors que ce mot est présent dans tous les membres de la famille. Dans la pratique, il est rare d'observer des mots exactement conservés, car des variabilités existent. On peut décrire ces incertitudes sur le contenu d'un mot. Si l'on hésite entre deux nucléotides possibles pour une position précise dans un mot, on peut avoir recours aux nucléotides ambigus (voir tableau 1.2). Par exemple, RUGN indique que le mot peut commencer avec le nucléotide A ou G (représenté ici par R), et finir avec n'importe quel nucléotide. Pour exprimer une variation, on peut aussi donner un mot attendu, ainsi qu'un nombre d'*erreurs* tolérées. Ces erreurs peuvent être des substitutions, des insertions ou des suppressions de nucléotides dans le mot attendu.

Par exemple, on peut voir dans l'alignement de la figure 3.1 quatre mots conservés, près des étiquettes « mot 1 » à « mot 4 ». On peut par exemple constater que le mot GA se retrouve systématiquement dans toutes les séquences. Il est situé entre les positions (indiquées au-dessus de l'alignement) x_7 et x_8. Un deuxième mot est conservé entre les positions x_9 et x_{10}. Ce mot subit quelques variations sur son premier nucléotide : on observe parfois AUGA, parfois GUGA. On peut alors utiliser les nucléotides ambigus et appeler ce mot RUGA. Le mot ACA est aussi conservé entre les positions x_{17} et x_{18}.

On peut d'ores et déjà constater qu'il existe un problème d'interprétation de ce que peut être un « mot conservé ». En effet, dans un cas extrême, toute famille d'ARN peut être décrite de façon relâchée comme une suite suffisamment longue de nucléotides N. Mais une signature doit aussi être suffisamment discriminante pour pouvoir caractériser précisément l'ensemble des ARN qui la composent. Nous reviendrons sur ce problème par la suite.

3.2.2 L'hélice conservée

Une hélice est une succession d'interactions contiguës, comme représenté sur la figure 3.3(a). En général, on ne considère dans les hélices que des appariements de type Watson-Crick et éventuellement *wobble* (cf. paragraphe 1.1.4 pour un rappel sur l'analyse thermodynamique de l'ARN). Comme dans les mots, on peut tolérer des erreurs dans les hélices. On peut par exemple trouver des mésappariements, qui peuvent être en pratique des appariements dits *non-canoniques*, c'est-à-dire qu'ils ne sont ni Watson-Crick, ni *wobble* (cf. paragraphe 1.1.1). On peut aussi trouver des insertions ou des suppressions d'un côté ou l'autre de l'hélice. On a alors un renflement, puisqu'un nucléotide n'est pas apparié. Si l'on trouve plusieurs nucléotides contigus mésappariés, on préfèrera y voir une boucle intérieure.

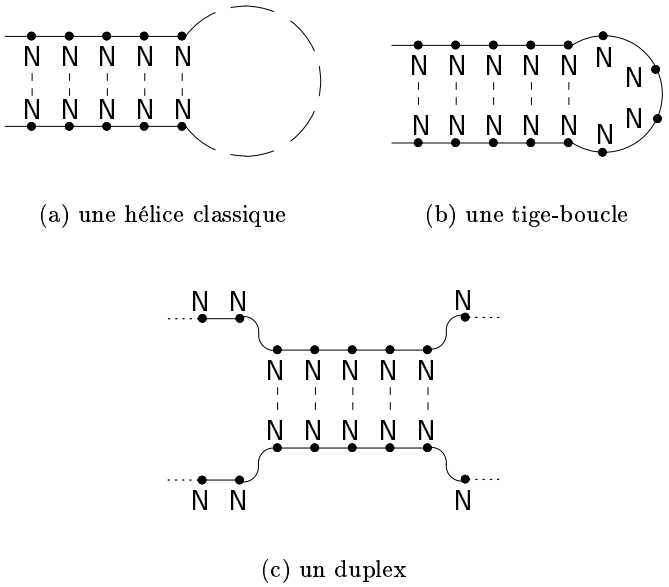
Hélices (contenant éventuellement des erreurs), renflements, et boucles intérieures sont autant d'éléments de structure qui peuvent composer une signature. Ces éléments sont en général accompagnés d'indications de longueur. Concernant les hélices, on peut donner leur taille (plus précisément, le nombre de nucléotides qui composent leurs brins) et la taille de leur boucle. Dans notre alignement, on remarque que toutes les séquences contiennent une hélice, dont le premier brin est compris entre les positions x_1 et x_2, et le second brin, entre les positions x_{15} et x_{16}. Pour être plus précis, cette hélice a une taille de sept à huit paires de bases, et que sa boucle peut contenir entre quarante et cinquante-huit nucléotides. Une insertion d'un nucléotide est éventuellement observée dans cette hélice, comme c'est le cas pour *Pf - H/ACA - 6*.

Il existe de plus quelques hélices particulières. La première est la tige-boucle 3.3(b). Il s'agit d'une hélice dont la boucle a une taille ne dépassant pas les quelques dizaines de nucléotides qui ne sont pas appariés entre eux. C'est un élément tellement fréquent qu'il est souvent utilisé pour décrire une signature. On peut aussi vouloir utiliser les duplex (cf. figure 3.3(c)), qui sont des hélices dont les brins sont situés sur deux séquences différentes. Cet élément de signature, qui n'est modélisable que dans MilPat (que nous présenterons dans le paragraphe 3.3.3), a pourtant une grande utilité. Il permet de modéliser des interactions entre deux séquences différentes. Ces interactions sont parfois au cœur de la fonction d'un ARNnc. C'est le cas des pARN à boîte H/ACA, dont la fonction est de guider un processus de transformation appelé *pseudo-uridilation* d'un nucléotide U appartenant à une séquence distincte du pARN. Dans la pratique, deux parties du pARN (situées entre les positions x_3 et x_4 d'une part, et entre x_{13} et x_{14} d'autre part) reconnaissent les séquences situées de part et d'autre du nucléotide U d'une autre séquence à transformer. Cette reconnaissance se fait par des interactions entre les deux brins. Une fois la région reconnue, un complexe protéique vient effectuer la transformation. Il est donc parfois important d'inclure les duplex dans les signatures.

3.2.3 Autres éléments de structure

Certains ARNnc ont des caractéristiques spéciales qui ne peuvent pas réellement se traduire en termes d'hélices et de mots. C'est par exemple le cas du (G+C)%, c'est-à-dire le pourcentage de nucléotides G et C dans une région donnée. Nous avons vu dans le paragraphe 1.1.4 que, pour des raisons de stabilité, on peut observer des zones qui contiennent un fort pourcentage de nucléotides G et C par rapport au pourcentage moyen du génome ou de la région considérée. Dans notre alignement, on peut remarquer que la proportion de ces nucléotides entre les positions x_1 et x_2, x_5 et x_6, et x_{11} et x_{12} est relativement élevée. Il est intéressant d'en tenir compte dans une signature.

Un autre élément de structure parfois important est l'interaction non-canonique. Nous avons vu dans le paragraphe 1.1.4 que les interactions Watson-Crick ou *wobble* ne sont pas les seules observées. Il peut exister un grand nombre d'autres interactions, faisant par exemple intervenir les côtés sucre ou Hoogsteen de chaque nucléotide. Ces interactions se retrouvent dans un élément appelé *k-turn* (cf. paragraphe 1.1.1), dont la forme est rappelée dans la figure 3.4. Dans notre alignement, il s'avère qu'en fait les mots GA et RUGA sont les deux brin d'un *k-turn*. Il est en pratique préférable d'utiliser



(a) une hélice classique (b) une tige-boucle

(c) un duplex

FIG. 3.3 – Trois exemples d'hélice.

des interactions non-canoniques plutôt que des mots, pour modéliser des *k-turn*. Les premières sont en effet plus à même d'expliquer les variations observées dans les mots conservés comme dans le cas du mot 2 de l'alignement où le nucléotide A comme le nucléotide G peut apparaître en première position.

Un dernier élément de structure utile est la répétition d'un mot. Comme l'hélice, la répétition peut contenir des erreurs qui sont les substitutions, les insertions ou les suppressions.

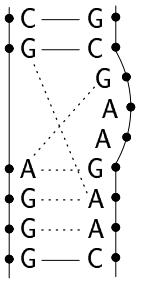


FIG. 3.4 – Un exemple de *k-turn* — les interactions canoniques sont dessinées en traits pleins, les interactions non-canoniques sont dessinées en pointillés.

3.2.4 L'espaceur

Dans la pratique, et on peut le vérifier sur l'alignement, les éléments de structure ne sont pas situés exactement l'un juste derrière l'autre. Il existe souvent des zones de taille variable, où l'on ne parvient pas trouver d'élément conservé. On utilise alors un espaceur, qui indique la distance relative entre deux autres éléments de structure. L'espaceur n'est pas en lui-même un élément de structure, mais il impose une relation entre deux éléments de structure. Il est crucial pour décrire une signature de façon précise.

Dans l'alignement, entre le mot GA et RUGA, c'est-à-dire entre les positions x_8 et x_9, il existe une séquence de taille variable, où l'on ne peut pas, de façon satisfaisante, trouver de mot ou hélice conservé. La seule information que l'on peut extraire de cet alignement est la taille qu'elle a. C'est cette caractéristique que l'on exploite dans l'élément de structure d'espacement. Ainsi, dans le cas étudié, il existe un espaceur entre les positions x_8 à x_9, dont la taille varie entre trois et vingt-et-un nucléotides.

3.2.5 Conclusion

Nous avons dressé ici une liste des éléments de signature. Peut-être en avons-nous oublié, mais il semble que la plupart des signatures de famille d'ARN puissent être décrites en termes de mots, hélices (de types divers), interactions, (G+C)% ou espaceurs. Il est toutefois possible que, dans le futur, nous trouvions de nouveaux éléments de structures dans de nouveaux ARN (eux-mêmes potentiellement découverts par des méthodes bio-informatiques). Nous verrons dans les prochains paragraphes comment le formalisme des réseaux de contraintes permettent d'ajouter simplement un nouvel élément de structure.

Nous avons néanmoins laissé plusieurs zones d'ombre. La première est la gestion des erreurs dans les éléments de structure. Nous pouvons tirer du chapitre 2 plusieurs manières de décrire les erreurs. La plus simple est de donner le nombre d'erreurs maximal que l'on accepte. Pour les mots et les hélices, le modèle pourrait être plus sensible et spécifique si l'on précisait séparément les insertions, suppressions et substitutions. En s'inspirant des matrices de poids position-spécifiques ou des chaînes de Markov cachées (paragraphe 2.2.2.1), il peut être intéressant d'autoriser des erreurs précises à des endroits donnés. La signature peut ainsi être complexifiée à loisir afin de la rendre la plus spécifique possible.

Mais deux éléments importants relativisent l'intérêt des signatures complexes. Tout d'abord, il est clair que la recherche de signatures d'ARN n'est intéressante que si tous les membres d'une même famille ne sont pas connus. De plus, on observe que les éléments d'une famille connaissent toujours des exceptions, des variabilités d'un génome à l'autre, ou même souvent au sein d'un même organisme. Partant de là, les exceptions sont susceptibles d'apparaître pratiquement partout, et donc toute signature ne peut pas être à la fois très discriminante (autrement dit, être très spécifique), et regrouper tous les membres d'une famille, c'est-à-dire être très sensible. Pour ces raisons, nous ne ferons pas le choix de spécifier des signatures utilisant un nombre important de

paramètres dans les signatures.

De plus, nous voulons que la signature soit facilement compréhensible et modifiable. L'accumulation des paramètres et des valeurs nuit bien sûr à la lisibilité de la signature, surtout s'il faut spécifier des probabilités d'insertion, de suppression, de mutation de chaque nucléotide.

3.3 Modélisation

Dans les paragraphes précédents, nous avons vu comment la recherche de signatures d'ARN pouvait se décomposer en recherche d'éléments de structures, composés par exemple de mots, hélices, (G+C)%, appariements, répétitions, espaceurs, etc. Nous allons ici montrer comment les réseaux de contraintes ont été utilisés pour effectuer cette tâche. Nous présenterons tout d'abord ce formalisme, puis nous étudierons l'approche développée par l'équipe de D. Gilbert, et enfin l'approche conçue par P. Thébault durant sa thèse.

3.3.1 Réseaux de contraintes

3.3.1.1 Définition

Définition 3.1 *Les* réseaux de contraintes *(RC) [Mon74] sont un triplet* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *où :*
- $\mathcal{X} = \{x_1, \ldots, x_n\}$ *est l'ensemble des* n variables *du problème ;*
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ *est l'ensemble des valeurs possibles pour chaque variable, appelé* domaines*, et par convention* $d = \max_{i \in [1..n]} |D_i|$ *;*
- $\mathcal{C}$ *est l'ensemble des* e contraintes *du problème.*

Chaque contrainte $c \in \mathcal{C}$ *porte sur un certain nombre de variables du problème* $var(c) = \{x_{i_1}, \ldots, x_{i_r}\}$*, où* r *est l'*arité *de la contrainte* c*. En d'autres termes,* c *est une relation, sous-ensemble de* $D_{i_1} \times \cdots \times D_{i_r}$*.* $var(c)$ *est appelée* portée *de* c*.*

L'*affectation* d'une variable x_i est une paire $\{(x_i, v_i) : v_i \in D_i\}$, habituellement notée $x_i \leftarrow v_i$. Une affectation de l'ensemble X est l'union de l'affectation des variables de X. Une affectation *totale* est l'affectation de toutes les variables du problème, tandis qu'une affection *partielle* ne touche pas nécessairement toutes les variables. L'ensemble des affectations possibles de l'ensemble des variables X est noté $\ell(X)$.

On définit la *projection* d'une affectation totale $a = \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}$ sur l'ensemble $X = \{x_{i_1}, \ldots, x_{i_m}\}$ par l'affectation partielle $\{x_{i_1} \leftarrow v_{i_1}, \ldots, x_{i_n} \leftarrow v_{i_n}\}$. Elle est notée $a[X]$.

On dit que l'affectation $a \in \ell(var(c))$ *satisfait* c si $a \in c$. Il est usuel de noter c_i la contrainte unaire (c'est-à-dire d'arité 1) ne portant que sur x_i, et c_{ij} la contrainte binaire (d'arité 2) ne portant que sur x_i et x_j. Nous nommerons enfin $\mathcal{C}_i$ l'ensemble des contraintes d'arité i, et $\mathcal{C}_+$ l'ensemble $\bigcup_{i>1} \mathcal{C}_i$.

On appelle *solution* du problème une affectation totale qui satisfait toutes les contraintes. Plus formellement, c'est une affectation $a \in \ell(\mathcal{X})$ telle que $\forall c \in \mathcal{C}, a[var(c)] \in c$. Un réseau qui n'a pas de solution est dit *incohérent*. Il est *cohérent* dans le cas contraire. La

requête usuelle sur un réseau de contraintes est de savoir s'il est cohérent. Ce problème est NP-complet.

Ces réseaux de contraintes sont très utilisés pour modéliser des problèmes proches de la recherche opérationnelle, allant de l'ordonnancement à l'établissement d'emploi du temps [Wal96]. Des outils commerciaux tels qu'ILOG Solver ou KAOLOG implantent des algorithmes de résolution efficaces.

Ces réseaux de contraintes ont été utilisés à plusieurs reprises pour modéliser des problèmes biologiques (voir [GBMS95] et [RBW06, chapitre 21] pour revue). Des problèmes de détermination de structure de protéines [KB02, PDF04], de visualisation de structure secondaire d'ARN [MGEW93] ou de détermination de structure tertiaire d'ARN [MTG+91, MGC93, LCCM98] ont par exemple été modélisés dans des réseaux de contraintes.

Afin d'illustrer les définitions précédentes, considérons l'exemple suivant, présenté dans [Bes92] et [ASBG02], ainsi que sa formalisation sous la forme d'un réseau de contraintes :

Exemple 3.1 *La firme Peunault va sortir un nouveau modèle de voiture fabriqué dans toute l'Europe :*

- *les* portières *et l'*intérieur *sont faits à Lille, où le constructeur ne dispose que de peintures* jaune, rouge *et* noire *;*
- *la* carrosserie *est faite à Hambourg, où l'on a de la peinture* blanche, jaune, rouge *et* noire *;*
- *les* pare-chocs, *faits à Palerme, sont toujours* blancs *;*
- *la bâche du* toit-ouvrant, *qui est faite à Madrid, ne peut être que* rouge *;*
- *les* enjoliveurs *sont faits à Athènes, où l'on a de la peinture* jaune *et de la peinture* rouge.

Le concepteur de la voiture impose quelques-uns de ses désirs quant à l'agencement des couleurs pour cette voiture :

- *la* carrosserie, *les* portières *et l'*intérieur *doivent être de la même couleur ;*
- *les* enjoliveurs, *les* pare-chocs *et le* toit-ouvrant *doivent être plus clairs que la* carrosserie.

Ce problème peut se modéliser sous la forme d'un réseau de contraintes binaires (toutes les contraintes portent sur au plus deux variables) dont les variables x_p, x_i, x_c, x_{pc}, x_{to} et x_e correspondent aux différents composants de la voiture, respectivement les portières, *l'*intérieur, *la* carrosserie, *le* pare-chocs, *le* toit-ouvrant *et les* enjoliveurs. *Les couleurs de peinture* blanc, jaune, rouge, noir *sont représentées par les symboles b, j, r et n. Les contraintes et leurs relations découlent simplement du problème. Le graphe du réseau de contraintes est donné figure 3.5. Chaque variable est représenté par un rectangle, et chaque valeur est représentée par un cercle. Les contraintes unaires qui portent sur chaque variable interdisent certaines valeurs. Celles-ci sont alors barrées (la valeur $(x_{pc} \leftarrow j)$ est par exemple interdite). Les contraintes binaires autorisent certaines paires de valeurs, qui sont alors reliées par un arc ($(x_{pc} \leftarrow b, x_c \leftarrow j)$ est par exemple autorisé).*

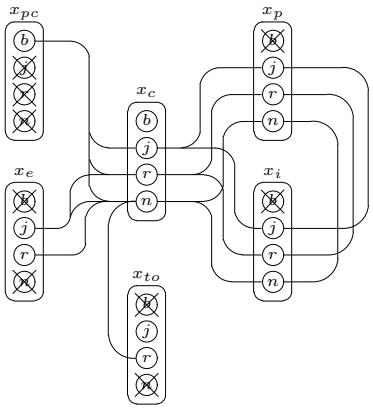


FIG. 3.5 – Un réseau de contraintes pour la conception de voitures Peunault.

Ce problème possède deux solutions :

$$S_1 = \{x_p \leftarrow n, x_i \leftarrow n, x_c \leftarrow n, x_{pc} \leftarrow b, x_{to} \leftarrow r, x_e \leftarrow j\}$$
$$S_2 = \{x_p \leftarrow n, x_i \leftarrow n, x_c \leftarrow n, x_{pc} \leftarrow b, x_{to} \leftarrow r, x_e \leftarrow r\}$$

3.3.1.2 Recherche de solutions

Une manière simple de résoudre le problème est d'explorer un arbre de retour-arrière (ou *backtrack*) en profondeur d'abord. La recherche choisit une variable, ainsi qu'une valeur de son domaine. Elle affecte la variable à cette valeur. Elle teste s'il cette affectation viole une contrainte. Si c'est le cas, alors on choisit une nouvelle valeur. Si ce n'est pas le cas, l'exploration de l'arbre continue en choisissant une nouvelle variable. Si toutes les variables sont affectées, et que l'on n'a pas détecté d'incohérence, alors on a une solution. Si toutes les valeurs d'une variable ont été testées sans trouver de solution, alors on procède à un retour-arrière, c'est-à-dire que l'on revient à la variable précédente et on lui affecte une autre valeur.

La procédure 5 (RechercheSolutions) donne les lignes directrices d'un algorithme de recherche de solutions dans les réseaux de contraintes. L'algorithme utilise des exceptions pour gérer les retours-arrières.

3.3.1.3 Algorithmes de filtrages

Les algorithmes de filtrage sont habituellement utilisés dans les solveurs de contraintes pour rechercher les solutions plus rapidement.

Un algorithme de filtrage transforme un problème en un problème équivalent, c'est-à-dire ayant le même ensemble de solution. Le nouveau problème doit être plus explicite dans le sens où une classe bien définie d'incohérences détectées dans de petits sous-problèmes sont rendus explicites. Si le problème obtenu est trivialement incohérent, alors on sait que le problème d'origine est incohérent. En particulier, la propriété de

Procédure 5 – RechercheSolutions($x_i \in \mathcal{X}$)

```
tant que (D_i ≠ ∅) faire
    v_i ← choix(D_i) ;
    essayer
        RechercheIncohérence() ;
        x_j ← ProchaineVariable() ;
        RechercheSolutions(x_j) ;
    D_i ← D_i \ {v_i} ;
```

cohérence locale la plus utilisée, appelée *cohérence d'arc*, cherche à détecter des valeurs qui ne permettent pas de satisfaire une contrainte.

La propriété de cohérence locale est souvent maintenue dans l'arbre de recherche. Les algorithmes de filtrage sont donc lancés avant et après chaque affectation de variable. Un algorithme de recherche de solutions générique utilisant une méthode de filtrage est décrit dans la figure 6 (RechercheSolutions2). Les lignes inchangées par rapport à la procédure 5 ont été grisées et les changements apparaissent en noir, afin de mettre en valeur les modifications.

Procédure 6 – RechercheSolutions2($x_i \in \mathcal{X}$)

```
tant que (D_i ≠ ∅) faire
    v_i ← choix(D_i) ;
    essayer
        Filtrage() ;
        x_j ← ProchaineVariable() ;
        RechercheSolutions(x_j) ;
    D_i ← D_i \ {v_i} ;
```

Une des propriétés de cohérence locale les plus simples que l'on puisse appliquer à un réseau de contraintes est la *cohérence de nœud* (CN), appelée aussi *node consistency* ou *NC* [Mac77] :

Définition 3.2 *Une variable x_i est nœud-cohérente si :*
- $\forall v_i \in D_i, v_i \in c_i$,
- $D_i \neq \varnothing$.

Un réseau est nœud-cohérent si toutes ses variables le sont.

La cohérence de nœud vérifie donc que toutes les valeurs de chaque variable sont autorisées par les contraintes unaires et que chaque domaine n'est pas vide. L'exemple suivant illustre cette propriété.

Exemple 3.2 *Considérons l'exemple de la figure 3.5. Les contraintes unaires du problème interdisent certaines valeurs du problème, comme par exemple la valeur $(x_{pc} \leftarrow j)$.*

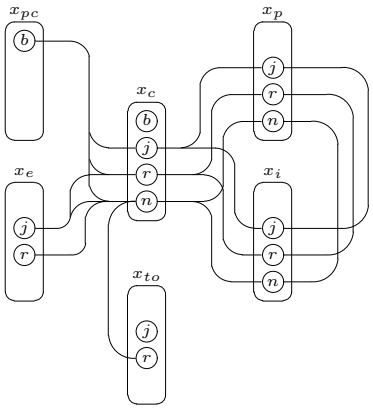


FIG. 3.6 – Le réseau de contraintes après établissement de la cohérence de nœud.

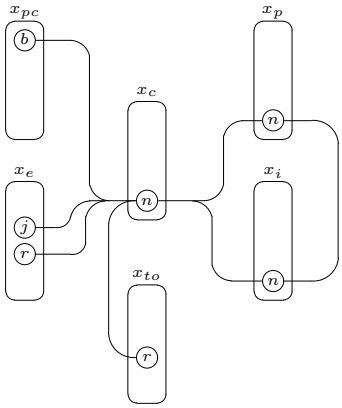


FIG. 3.7 – Le réseau de contraintes après établissement de la cohérence d'arc.

On peut donc retirer toutes ces valeurs du problème, puisqu'elles ne participeront à aucune solution. Après application de la cohérence de nœud au problème, on a le problème décrit dans la figure 3.6.

La cohérence d'arc (CA), appelée aussi *arc consistency* ou AC est la propriété de cohérence locale la plus employée dans les réseaux de contraintes. Elle se définit de la manière suivante [Mac77] :

Définition 3.3 *Une variable x_i est arc-cohérente si :*
- $\forall c \in \mathcal{C}, x_i \in var(c) \Rightarrow \forall v_i \in D_i, \exists a \in \ell(var(c) \setminus \{x_i\}), a \cup \{(x_i \leftarrow v_i)\} \in c$ *(a est alors appelé* support *de v_i) et*
- *x_i est nœud-cohérente.*

Un réseau est arc-cohérent si toutes ses variables le sont.

L'exemple suivant illustre la propriété de cohérence d'arc.

Exemple 3.3 *Considérons, dans l'exemple de la figure 3.6, la valeur r de la variable x_c et la contrainte liant x_c à x_{to}. On peut voir que, pour n'importe quelle valeur de x_{to}, la contrainte liant x_{to} à x_c n'accepte pas la valeur $(x_c \leftarrow r)$. Cette valeur ne participe donc à aucune solution du problème et l'on peut la supprimer sans changer l'ensemble des solutions du problème. Si l'on applique donc la cohérence d'arc au problème de la figure 3.6, on a le problème décrit dans la figure 3.7.*

Il existe une troisième propriété de cohérence locale qui est parfois utilisée, notamment lorsque les domaines des variables sont grands. Il s'agit de la *2B-cohérence* [Lho93] (appelée aussi *2B-consistency*). Cette propriété suppose que l'on a défini un ordre total sur les domaines. Chaque domaine D_i peut donc être défini comme un intervalle, I_i, dont la borne inférieure est nommée bi_i, et la borne supérieure, bs_i. La 2B-cohérence s'apparente à la cohérence d'arc, si ce n'est qu'elle n'établit les supports qu'aux bornes des domaines. Plus formellement :

Définition 3.4 *$x_i \in \mathcal{X}$, où $I_i = [bi_i..bs_i]$, est 2B-cohérent si*

$$\begin{aligned}\forall c \in \mathcal{C}, x_i \in var(c) \Rightarrow & (\exists a \in \ell(var(c) \setminus \{x_i\}), a \cup \{(x_i \leftarrow bi_i)\} \in c) \wedge \\ & (\exists a' \in \ell(var(c) \setminus \{x_i\}), a' \cup \{(x_i \leftarrow bs_i)\} \in c))\end{aligned}$$

Un réseau de contraintes est 2B-cohérent si toutes ses variables le sont.

Nous utiliserons cette propriété de cohérence locale par la suite.

3.3.2 Approche développée par l'équipe de D. Gilbert

L'équipe de David Gilbert utilise une modélisation basée sur les mots pour répondre à la question de la recherche de signatures. À partir de leur modélisation, deux implantations ont été proposées. La première s'appuie sur la programmation logique avec contraintes, la seconde, sur les réseaux de contraintes. Nous présentons ici la modélisation, les implantations ainsi que les résultats rendus publics par l'équipe.

3.3.2.1 Modélisation

L'équipe de D. Gilbert a écrit deux articles sur la recherche de signatures d'ARN : [BG95] et [EGJ+01]. Leur approche est la suivante. Toute signature peut se décomposer en mots, qui vérifient certaines propriétés. Par exemple, une hélice se décompose en deux mots, qui sont miroirs-compléments l'un de l'autre. Ainsi, rechercher une occurrence de signature revient à rechercher tous les mots qui satisfont cette signature. Une solution est alors un ensemble de mots qui respectent les propriétés énoncées dans la signature. La liste des propriétés exprimables est la suivante :

- la longueur d'un mot ;
- la distance entre deux mots ;
- le contenu d'un mot ;
- la position d'un mot par rapport au début de la séquence ;
- des corrélations entre mots, où l'on peut distinguer :

- le fait que deux mots soient semblables, c'est-à-dire égaux, moyennant quelques erreurs possibles ;
- le fait que deux mots soient miroirs-compléments l'un de l'autre.

Tentons d'appliquer ce formalisme à la recherche de pARN à boîte H/ACA. Un tel ARN peut se décomposer en une hélice de taille six, trois mots conservés (GA, RUGA, ACA), et des espaceurs. Un descripteur, c'est-à-dire une signature exprimée en termes propres à l'approche, nécessite donc cinq mots : deux pour l'hélice et un pour chaque mot conservé. On emploiera pour ces mots les lettres grecques, de α à ε. Les contraintes sont les suivantes :

- il existe une hélice entre α et δ, ce qui se note `miroir-complément`(α, δ) ;
- les deux brins qui composent cette hélice ont une taille variant entre sept et huit, ce qui se note `longueur`$(\alpha, [7..8])$ et `longueur`$(\delta, [7..8])$;
- la variable β est le mot GA, ce qui se note `identité`(β, GA) ;
- la variable γ est le mot RUGA, soit `identité`(γ, RUGA) ;
- la variable ε est le mot ACA, soit `identité`$(\varepsilon, \mathsf{ACA})$;
- on peut trouver de quinze à dix-sept nucléotides entre le début du mot α et le début du mot β, ce qui se note `distance`$(\alpha, \beta, [15..17])$;
- les autres contraintes de distance sont similaires.

Une fois le descripteur créé, on peut l'utiliser sur *Pa - H/ACA - 21*, le premier mot de notre alignement. La solution satisfait les propriétés suivantes :

$$\alpha = \mathsf{CCGCCCG}, \beta = \mathsf{GA}, \gamma = \mathsf{AUGA}, \delta = \mathsf{CGGGGCGG}, \varepsilon = \mathsf{ACA}$$

Il sera bien sûr nécessaire d'obtenir les positions où se trouvent ces mots dans la séquence.

3.3.2.2 Reformulation pour programmation logique avec contraintes

Les auteurs proposent deux implantations possibles. La première utilise la programmation logique avec contraintes, et s'appuie sur clp(FD) [DC93], une implantation de Prolog avec variables à domaine fini équipée d'un solveur de contraintes arithmétiques. Elle décompose chaque mot de la signature en couples nucléotide / position, notés (n_i, p_i), qui seront les nouvelles variables du problème. Par exemple la contrainte `identité`(β, GA) utilise deux couples nucléotide / position qui sont (n_1, p_1) et (n_2, p_2). La traduction de la contrainte d'identité sur ces couples donne alors le jeu de contraintes :

$$n_1 = \mathsf{G}, n_2 = \mathsf{A}, p_2 = p_1 + 1$$

Cette modélisation n'autorise pas de définir des mots de taille variable. Cela interdit donc de fait les hélices de taille variable, ainsi que les insertions et les suppressions dans les hélices. Mais exprimer une hélice de taille trois, dont la boucle est de longueur quatre est possible et revient à utiliser les quatre contraintes suivantes :

- `longueur`$(\alpha, 3)$,
- `longueur`$(\beta, 3)$,
- `distance`$(\alpha, \beta, 6)$ (la distance compte en effet le nombre de nucléotides séparant deux débuts de mot) et

– `miroir-complément`(α, β),

et donc trois couples nucléotide / position (n_1^α, p_1^α), (n_2^α, p_2^α) et (n_3^α, p_3^α) pour α, et autant pour β. La modélisation est alors la suivante :

`longueur`$(\alpha, 3)$	devient	$p_2^\alpha = p_1^\alpha + 1, p_3^\alpha = p_2^\alpha + 1$;
`longueur`$(\beta, 3)$	devient	$p_2^\beta = p_1^\beta + 1, p_3^\beta = p_2^\beta + 1$;
`distance`$(\alpha, \beta, 6)$	devient	$p_1^\beta = p_1^\alpha + 7$;
`miroir-complément`(α, β)	devient	$n_3^\beta =$ `complément`(n_1^α),
		$n_2^\beta =$ `complément`(n_2^α),
		$n_1^\beta =$ `complément`(n_3^α).

Après cette reformulation, les contraintes, codées sous forme numérique, sont données au solveur. Celui-ci utilise comme variables ces couples nucléotide / position, si bien que le domaine de chaque variable est l'ensemble des positions possibles sur la séquence d'entrée, multiplié par le nombre de nucléotides. Le solveur résout le système en exploitant le non-déterminisme de Prolog pour explorer les différentes alternatives et un mécanisme de coupure basé sur la propriété de contraintes arithmétiques. Il retourne chaque solution sous la forme d'un couple nucléotide / position pour chaque variable utilisée.

3.3.2.3 Reformulation pour les réseaux de contraintes

Une autre implémentation présentée dans [EGJ+01] utilise directement un solveur de réseaux de contraintes. Le but n'est pas ici de savoir si le réseau est cohérent ou non, mais plutôt de connaître toutes les solutions du problème.

Un solveur de contraintes a été développé par l'équipe de D. Gilbert pour résoudre le problème de la localisation de signatures d'ARN. Il traduit chaque mot α en un couple de positions de début et de fin du mot, notés (d_α, f_α). Chaque position est alors utilisée comme variable. Le domaine de chaque variable est donc l'ensemble des positions possibles dans la séquence d'entrée.

Les propriétés sur les mots sont traduites en contraintes impliquant en général un ou deux couples de variables. Ainsi, la contrainte `identité`(α, RUGA) devient la contrainte `identité`$(d_\alpha, f_\alpha, \mathsf{RUGA})$, qui est satisfaite lorsque $S[d_\alpha, f_\alpha] \in \{\mathsf{AUGA}, \mathsf{GUGA}\}$, où S est la séquence génomique d'entrée, et où $S[i..j]$ est la sous-séquence de S commençant à la position i, et se terminant à la position j.

Une fois la traduction faite, une passe de cohérence d'arc (présentée dans le paragraphe 3.3.1.3) est effectuée sur les contraintes de distance seulement, puis la recherche s'effectue par retour-arrière, sans utiliser d'autre filtrage. La solution d'un problème est ici la donnée des positions pour chaque début et fin de chaque mot.

3.3.2.4 Expérimentations

Deux types d'expérimentation ont été menés par les auteurs. Tout d'abord, deux petites signatures contenant deux à quatre variables de mots et une à trois contraintes ont été créées. Elles ont ensuite été lancées sur des séquences aléatoires ne dépassant pas les six cents nucléotides. L'approche par programmation logique avec contraintes donne

l'ensemble des solutions recherchées en moins d'une seconde. L'approche par réseaux de contraintes peut, elle, nécessiter près de six secondes.

Les deux approches sont ensuite utilisées pour retrouver un pseudo-nœud contenu dans le virus de la mosaïque du tabac [Ple94]. La séquence analysée compte cent quatre-vingt-huit nucléotides, et la recherche avec les deux approches reste en dessous de la seconde.

3.3.2.5 Conclusion

L'approche développée par l'équipe de D. Gilbert est probablement la première à avoir explicitement utilisé les réseaux de contraintes pour modéliser le problème de recherche d'occurrences de signatures d'ARNnc. Les perspectives offertes par ce formalisme, comme la possibilité de modéliser des pseudo-nœuds, sont déjà très exploitées. Mais ces travaux sont sans doute préliminaires. Les renflements dans les hélices ne sont pas acceptés, même dans l'approche par réseaux de contraintes. L'utilisation des coûts, évoquée dans les articles, n'est pas opérationnelle. Et surtout, les expérimentations sont peu convaincantes car elle ne présentent pas de problème de taille réaliste. De plus, l'approche par programmation logique avec contraintes utilise une variable pour chaque nucléotide de l'ARN recherché, dont la taille du domaine est quatre fois la taille de la séquence donnée en entrée. Cette méthode semble avoir peu de chance de donner des résultats compétitifs. Tel quel, l'outil n'est pas utilisable en pratique pour des recherches de signatures d'ARN réalistes. La modélisation utilisant des réseaux de contraintes semble aussi peu applicable à des cas concrets. La recherche de solutions dans le réseau de contraintes sans maintien de cohérence locale donne vraisemblablement des temps de résolution beaucoup trop importants sur des problèmes de taille réaliste.

3.3.3 Approche développée par P. Thébault

Nous allons ici décrire l'approche développée par Patricia Thébault dans le cadre de sa thèse, qui peut, en un sens, être vue comme une extension des travaux de D. Gilbert. Le développement d'un logiciel basé sur cette approche, nommé MilPat, permet de rechercher efficacement de nouveaux ARN non codants.

3.3.3.1 Modélisation

Partons par exemple de la modélisation de l'équipe de D. Gilbert, où toute signature est exprimée à partir de mots. Chaque mot peut être lui-même exprimé par un couple de positions, stockant l'endroit où commence le mot et où il se termine. Il est donc possible de raisonner exclusivement en termes de positions, plutôt que de mots. D'ailleurs, le problème étant d'identifier les occurrences d'une signature, il est naturel de raisonner en termes de positions. Ainsi, dans la modélisation proposée, les variables du réseau de contraintes correspondent aux positions sur la séquence génomique donnée en entrée, et les contraintes définissent les éléments de signature sur ces variables.

En d'autre termes, on suppose donnée une description de la famille incluant les éléments structurels caractéristiques pouvant être exprimés sous forme de mots, hélices,

répétitions, etc. Les étapes de la modélisation sont alors les suivantes.

- Les positions de début et de fin de ces éléments de structures sont notées. Il en faut par exemple deux pour un mot, et quatre pour une hélice (désignant le début et la fin de chaque brin de l'hélice).
- Ces positions seront les variables du problème.
- Les éléments de structure sont modélisés par des contraintes sur les variables.

Les solutions de ce réseau sont les valeurs des variables-positions qui respectent toutes les contraintes, c'est-à-dire qui sont des occurrences de la signature.

3.3.3.2 Variables

Dans cette modélisation, les variables représentent des positions dans la séquence génomique donnée en entrée. Ces positions délimitent les éléments de structure recherchés dans la séquence. Notons que le biologiste, s'il veut modéliser un duplex, doit donner deux séquences : la séquence principale et la séquence cible. Les variables-positions peuvent donc porter sur la séquence principale ou sur la séquence cible.

Chaque variable pouvant prendre n'importe quelle position sur la séquence où l'on recherche des solutions, la taille du domaine de la variable est donc la taille de la séquence. Ceci est notable car les séquences d'intérêt peuvent atteindre plusieurs centaines de millions de nucléotides.

3.3.3.3 Contraintes

Chaque contrainte posée modélise un élément de structure que l'on désire retrouver. Une contrainte porte sur une ou plusieurs variables, et n'est satisfaite que lorsque les variables sont positionnées sur des régions où l'on retrouve l'élément de structure recherché.

Dans sa thèse, P. Thébault met à disposition les contraintes suivantes :

- une contrainte de mot, portant sur une seule variable, qui est satisfaite si un mot donné commence à la position pointée par la variable ;
- une contrainte d'hélice, portant sur quatre variables, satisfaite si l'on trouve une hélice dont le premier brin est situé entre les deux premières variables, et l'autre brin est situé entre les deux dernières variables ;
- une contrainte de répétition, portant sur quatre variables, satisfaite si le mot trouvé entre les deux premières variables est sensiblement le même que celui situé entre les deux dernières variables ;
- une contrainte de duplex, portant sur quatre variables dont les deux premières pointent des positions sur la séquence principale, et les deux dernières, sur la séquence cible, satisfaite si l'on trouve un duplex sans erreur et sans liaison *wobble* entre la première paire de variables et la dernière ;
- une contrainte de distance, portant sur deux variables, satisfaite si une distance minimale entre les deux variables existe ;
- une contrainte de distance relative, portant sur quatre variables, satisfaite si la distance entre les deux premières variables est la même que la distance entre les

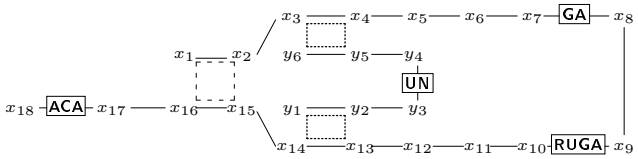


FIG. 3.8 – Ensemble des contraintes modélisant la structure d'un pARN — les lignes simples sont des contraintes de distance ; le rectangle en traits discontinus est une hélice ; les rectangles en pointillés sont des duplex ; les contraintes de mot sont réprésentées par des rectangles en traits continus, le mot lui-même étant écrit à l'intérieur.

deux dernières variables.

Pour les contraintes de mot, d'hélice ou de répétition, il est possible d'accepter des erreurs. Le nombre maximal d'erreurs doit alors être fourni pour chaque contrainte, ainsi que le type d'erreurs que l'on souhaite trouver : substitutions seulement, ou substitutions / insertions / suppressions.

Considérons l'exemple du pARN à boîte H/ACA, dont un alignement est donné dans la figure 3.1. On peut supposer que les éléments de structure conservés sont l'hélice, les quatre mots GA, RUGA, ACA et UN (ce dernier étant situé sur le brin cible), et les deux duplex. Afin de le modéliser dans MilPat, il faut créer une contrainte par élément de structure, ainsi qu'autant de variables que nécessaire : quatre pour chaque hélice et duplex, et une par mot. Cela nous donne un réseau, qui peut être représenté comme sur la figure 3.8.

3.3.3.4 Solutions

Une solution est une affectation totale, c'est-à-dire un positionnement de toutes les variables, qui respecte toutes les contraintes. Si toutes les contraintes sont satisfaites, c'est que tous les éléments de signature spécifiés apparaissent sur la séquence, et donc que l'on a bien une occurrence de cette signature.

Une fois les solutions données, chaque solution peut être représentée graphiquement comme sur la figure 3.9. Celle-ci reprend l'exemple du pARN à boîte H/ACA, où l'on a donné à MilPat le réseau de contraintes représenté sur la figure 3.8, avec le génome de *P. kodakarensis* comme séquence principale, et son ARN 23S comme séquence cible.

3.3.3.5 Implantation

P. Thébault a aussi développé un logiciel, nommé MilPat, implantant la modélisation précédemment décrite. Le solveur de contraintes utilise un algorithme de type retour-arrière avec branchement binaire maintenant la 2B-cohérence présentée dans le paragraphe 3.3.1.3. Les algorithmes de filtrage ont été conçus pour cette application et ont reçu une attention particulière afin d'accélérer la recherche.

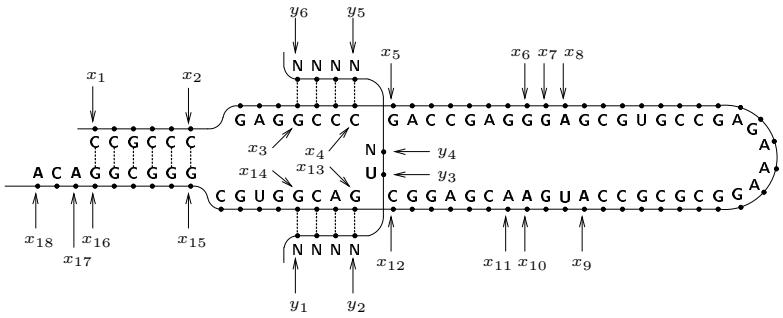


FIG. 3.9 – Solution dans le génome de *P. kodakarensis*.

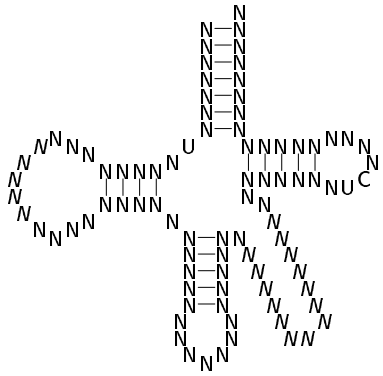


FIG. 3.10 – Un ARN de transfert pour de génomes d'eucaryotes — les nucléotides optionnels sont en italique.

3.3.3.6 Résultats

Afin de montrer l'intérêt du logiciel, P. Thébault a effectué deux jeux de test. Tout d'abord, MilPat est comparé aux autres outils existants, notamment RnaMot [GHC90, LGC94], PatScan [DLO97] et RNAMotif [MEG+01, MC01], que nous avons présentés dans le paragraphe 2.2.1.2. Une signature reconnaissant l'ARN de transfert proposée dans [LGC94] (décrit dans la figure 3.10) est traduite dans des langages compréhensibles pour chacun des logiciels. Ceux-ci sont ensuite utilisés pour retrouver les ARNt des génomes d'*Escherichia coli* et de *Saccharomyces cerevisiae*, la levure de bière. MilPat, quand il n'est pas le plus rapide, donne toujours des temps de réponse du même ordre de grandeur que le plus rapide des logiciels (pour un même nombre de solutions).

Une autre série de tests, développée dans [TdGSG06], a été effectuée pour rechercher d'autres ARN non-codants. De nouveaux ARN ont d'ailleurs été proposés, comme certains pARN à boîte H/ACA (présentés plus haut) dans le génome de *Methanococcus jannaschii*, une archée. L'intérêt de certaines fonctionnalités de MilPat a, par la même occasion, pu être testé. Par exemple, lors de la recherche de pARN à boîte H/ACA, l'ajout d'un duplex divise environ par dix le nombre de candidats proposés, et pour les pARN à boîte C/D (présenté dans la figure 1.5(b)), il permet de multiplier par au

moins deux la spécificité du descripteur. Afin de tester la robustesse du logiciel, les RNase P [KP06] ont été recherchés dans différents génomes d'archées (qui ont une longueur de quelques millions de nucléotides). Ces ARNnc, qui servent à la maturation des ARN de transfert, ont en effet une taille avoisinant quelques centaines de nucléotides. Le temps de recherche ne dépasse jamais les trente-cinq secondes.

3.3.3.7 Conclusion

Au vu des résultats, MilPat s'avère un outil de choix pour la recherche de signatures. L'utilisation des contraintes permet d'enrichir le nombre d'éléments de structure modélisables, comme des duplex, sans augmentation du temps de calcul. L'aspect modulaire du formalisme, qui permet d'ajouter simplement un nouveau type d'élément de structure en codant une nouvelle contrainte, est un autre avantage important de l'approche. Néanmoins, P. Thébault a relevé quelques améliorations restant à faire. Au chapitre algorithmique, elle note qu'un choix judicieux de l'ordonnancement des variables, même statique, permet de diviser par presque six le temps de résolution [TdGSG06]. Nous exposerons plus en détail les problèmes d'ordonnancement dans le chapitre suivant.

Certains problèmes de spécificité ont aussi été notés. En effet, pour avoir une signature la plus sensible possible, c'est-à-dire n'oubliant aucun candidat, il faut parfois accepter des éléments de structure dégénérés, que l'on ne retrouve que rarement. Le nombre de faux positifs augmente donc. Il semble alors intéressant de pénaliser les occurrences de ces structures dégénérées, afin d'émettre une préférence sur les structures « orthodoxes ». La pondération des contraintes semble alors un bon choix. De plus, pour un véritable ARN non codant, l'approche prédit parfois une dizaine de solutions, qui diffèrent entre elles d'un décalage sur la gauche ou sur la droite d'une variable. Ces solutions redondantes rendent ainsi la lecture des résultats difficile, et nous tenterons d'apporter une réponse à ce problème par l'utilisation de la dominance des solutions, évoquée dans le paragraphe 5.4.

D'autre part, quelques éléments de structures pourraient être ajoutées. La modélisation des pARN à boîte C/D ou H/ACA utilise des mots pour les signatures en *k-turn* (cf. figure 3.4) qu'ils contiennent. Une meilleure possibilité consisterait sans doute à modéliser les interactions non-canoniques qui les composent. De plus, nous avons vu que, notamment dans les pARN à boîte H/ACA (figure 3.1), il existe plusieurs régions où le (G+C)% est fort. Il est alors intéressant de pouvoir inscrire cette information dans une signature pour en améliorer sa spécificité.

3.4 Utilisation des réseaux de contraintes valuées

Nous allons ici présenter la modélisation que nous avons adoptée dans cette thèse. Celle-ci peut être vue comme l'ajout de la gestion des coûts par rapport aux travaux de P. Thébault. Nous avons aussi ajouté la possibilité d'accepter des erreurs dans les duplex, de modéliser des interactions non canoniques et de contrôler le (G+C)%.

3.4.1 Introduction

Nous avons énuméré dans le début de ce chapitre les éléments structurels que l'on pouvait rencontrer dans un ARN non-codant. Nous avons ensuite montré deux modélisations possibles dans les réseaux de contraintes classiques. Nous avons vu que ce formalisme présentait plusieurs avantages comme la faculté de représenter des pseudo-nœuds ou des duplex, des temps de recherche rapide, et la possibilité d'ajouter simplement de nouveaux éléments de structure suivant l'évolution des connaissances en biologie. Nous reprendrons donc ces aspects intéressants dans notre modélisation.

En revanche, un des inconvénients majeurs actuels de la recherche de signatures par réseaux de contraintes classiques est le fait que ceux-ci ne gèrent pas les coûts. Les outils non-probabilistes tels que Palingol [BKV96] ou RNAMotif [MEG+01, MC01] (cf. paragraphe 2.2.1.2), comme les outils probabilistes tels que la suite Infernal [ED94, LE97, GJBM+03, GJMM+05, KE03] et Erpin [GL01, LG03, LFL+04, LLG05, LLFG05] (cf. paragraphe 2.2.2.2) ont mis en avant l'utilité des scores. Ceux-ci servent notamment à construire des signatures plus précises et à quantifier la qualité d'une solution.

Dans les paragraphes suivants, nous définirons les réseaux de contraintes pondérées, puis nous détaillerons notre modélisation. Par rapport aux travaux précédents de P. Thébault, nous avons aussi proposé deux contraintes supplémentaires, l'une contrôlant le (G+C)%, et l'autre spécifiant des interactions non-canoniques. Nous avons aussi proposé quelques variantes de la contrainte d'hélice et revisité la contrainte de duplex afin d'y autoriser les erreurs. Nous présenterons ici les nouvelles modélisations de ces éléments de structure.

3.4.2 Réseaux de contraintes pondérées

3.4.2.1 Définition

De nombreux formalismes ont proposé d'étendre les réseaux de contraintes de manière à y inclure des préférences. Parmi eux, on peut par exemple citer les réseaux de contraintes floues [RHZ76], les réseaux de contraintes possibilistes [Sch92], les réseaux de contraintes probabilistes [FL93], les réseaux de contraintes basées sur un semi-anneau [BMR95, BMR97] et les réseaux de contraintes valuées [SFV95, BFM+99]. Ces derniers ajoutent aux réseaux de contraintes classiques une *structure de valuation*, permettant de définir une structure algébrique caractérisant les valuations ou *coûts* associés aux solutions du problème. Les *réseaux de contraintes pondérées* [Lar02] sont une spécialisation des réseaux de contraintes valuées, où la structure de valuation est la suivante.

Définition 3.5 *La structure de valuation $\mathcal{S}$ est un quadruplet $\langle E, \oplus, \ominus, \leq \rangle$ défini par :*
- *$E = [0..\top]$ est l'ensemble des* coûts *possibles, et $\top \in \mathbb{N}$, le plus grand coût, représente une* incohérence *;*
- *$\oplus$ est la* somme bornée *dans E :*

$$\forall (a, b) \in E^2, a \oplus b = \min\{\top, a + b\}$$

– $\ominus$ *est la* différence *dans* $[0..\top]$ *:*

$$\forall (a,b) \in E^2, a \ominus b = \begin{cases} \top & si\ a = \top \\ a - b & sinon \end{cases}$$

– $\leq$ *définit l'ordre usuel dans* $[0..\top]$.

Nous pouvons maintenant définir les réseaux de contraintes pondérées.

Définition 3.6 *Un* réseau de contraintes pondérées *(RCP) est un quadruplet* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{S} \rangle$ *où :*

– $\mathcal{X}$ *est l'ensemble des variables ;*
– $\mathcal{D}$ *est l'ensemble des domaines ;*
– $\mathcal{S}$ *est la structure de valuation du réseau ;*
– $\mathcal{C}$ *est l'ensemble des* contraintes pondérées.

La différence majeure avec les réseaux de contraintes classiques est que chaque contrainte pondérée c *est une* fonction de coût*, c'est-à-dire une fonction de* $var(c)$ *dans* E*. On définit alors le* coût $\mathcal{V}(a)$ *d'une affectation totale* $a \in \ell(\mathcal{X})$ *par :*

$$\mathcal{V}(a) = \bigoplus_{c \in \mathcal{C}} c(a[var(c)])$$

Une affectation totale dont le coût est inférieur à $\top$ *est appelée une* solution *du problème. Une requête classique que l'on effectue sur un réseau de contraintes pondérées est de trouver le coût minimal des solutions du problème, si le problème a des solutions.*

Lorsqu'elle n'existe pas, une fonction de coût d'arité nulle appelée $c_\varnothing$ est créée et prend la valeur nulle. Comme cette fonction ne porte sur aucune variable, son coût doit être payé pour toute affectation. Puisque le coût des fonctions s'additionne et que les coûts sont positifs ou nuls, la valeur de $c_\varnothing$ constitue un minorant du coût de n'importe quelle affectation des variables, et donc un minorant des solutions du problème.

Il est à noter que si $\top = 1$, alors le réseau de contraintes pondérées se réduit à un réseau de contraintes classiques, un coût de 0 signifiant la satisfaction, et 1, la violation. De plus, le problème de décision associé au problème d'optimisation d'un RCP est NP-complet.

Exemple 3.4 *L'exemple de la figure 3.11 décrit un petit RCP. Il contient deux variables (*x_1 *et* x_2*), chaque variable contenant deux valeurs (*a *et* b*). Sur chaque variable, on a ajouté une fonction de coût unaire, dont les coûts sont écrits dans les cercles. Par exemple, l'affectation de* x_1 *en* a *donnera un coût de trois. Il existe aussi une fonction de coût binaire, reliant chaque paire de variables. Les coûts sont écrits sur les arcs joignant les paires de valeurs. Par exemple, l'affectation de* x_1 *et* x_2 *en* a *donnera un coût de deux. S'il n'existe pas d'arc entre deux valeurs, le coût est par défaut* 0*. La valeur* $\top$ *a été arbitrairement fixée à* 4*.*

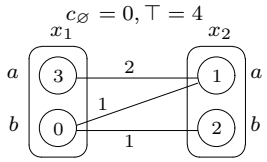


FIG. 3.11 – Un réseau de contraintes pondérées.

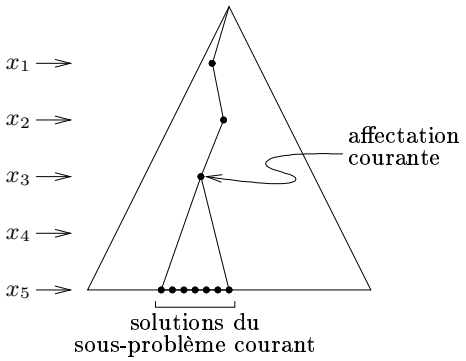


FIG. 3.12 – Arbre d'évaluation et de séparation recherchant les solutions d'un RCP.

3.4.2.2 Recherche de solutions

La recherche de solutions explore un arbre d'évaluation et de séparation en profondeur d'abord, comme sur la figure 3.12. À chaque nœud de l'arbre, on calcule un minorant du coût des solutions contenues dans le sous-arbre courant ainsi qu'un majorant du coût des solutions. Si l'on recherche le coût optimum des solutions du problème, ce majorant est donné par le plus petit coût des solutions trouvées jusqu'alors. Si le minorant est supérieur ou égal au majorant, alors on sait que le sous-problème ne contient pas de solution. Il est donc inutile de continuer la recherche dans le sous-arbre.

Afin de trouver un minorant aussi haut que possible du coût des solutions contenues dans le sous-arbre courant, on fait appel à des méthodes de filtrage.

Un algorithme de filtrage transforme un problème en un problème équivalent, c'est-à-dire ayant le même ensemble de solution et ayant une même distribution des coûts pour toutes les affectations totales du problème. Le nouveau problème doit être plus explicite dans le sens où des coûts obligatoirement payés dans de petits sous-problèmes sont rendus explicites. En particulier, le problème obtenu peut avoir une contrainte $c_\varnothing$ plus grand que le problème originel, fournissant ainsi un minorant du coût des solutions non naïf.

Nous présenterons dans le chapitre suivant plusieurs méthodes de filtrage.

3.4.3 Modélisation

Les étapes de la modélisation dans le réseau de contraintes pondérées sont essentiellement les mêmes que celles développées par P. Thébault. Les variables représentent

toujours des positions sur la séquence principale, et éventuellement sur la séquence cible si l'on a des duplex. Les tailles des domaines sont donc aussi les tailles des séquences.

Les coûts des réseaux de contraintes pondérées ont dans notre problème plusieurs significations. Ils traduisent tout d'abord l'adéquation du candidat avec la signature. Une solution au coût élevé indique ainsi qu'elle est peu conforme à la spécification. L'utilisation de coûts est aussi motivée par le fait qu'il existe une variabilité inhérente aux ARN non-codants. Cette variabilité doit être tolérée, dans une certaine mesure, car elle amène sinon à accepter trop de candidats. Ces coûts représentent également une préférence. Cette préférence peut être exploitée lors du choix de solutions. Supposons en effet que le solveur trouve deux solutions se chevauchant largement (c'est par exemple le cas si la seule différence entre les deux solutions est le fait qu'une variable est décalée d'un nucléotide sur la gauche ou la droite). Il est peu probable que l'on ait alors deux ARNnc différents, et il est plus probable qu'une solution soit un artefact. Il faut décider quelle solution éliminer. Naturellement, c'est la solution la moins préférée, donc celle ayant le coût le plus élevé, qui sera supprimée. Le mécanisme de coût pourra donc aider à sélectionner des solutions. Nous détaillerons ce mécanisme de sélection de solutions localement optimales dans le paragraphe 5.4.

Enfin, le caractère additif du coût des solutions autorise des phénomènes de compensation. Supposons par exemple que l'on recherche un petit ARN à boîte H/ACA. On peut décider d'accepter les solutions présentant des erreurs dans les mots conservés mais pas d'erreur dans l'hélice modélisée, ou bien des erreurs dans l'hélice mais pas dans les mots. Un juste réglage de la valeur $\top$ permettra de ne pas accepter les solutions dont les mots et l'hélice contiennent des erreurs. Plus généralement, un réseau de contraintes pondérées peut s'interpréter de façon probabiliste, comme l'ensemble des potentiels d'un champ de Markov [GG84, DJ89].

3.4.4 Fonctions de coût

Dans notre modélisation, les fonctions de coût permettent de spécifier que l'on désire trouver un élément de structure dans la région pointée par les variables de la fonction. La finesse de la signature dépend ainsi largement de l'offre en fonctions de coût proposée par notre approche. Nous avons tenté de modéliser tous les éléments de structure que nous connaissions, et nous avons aussi proposé quelques alternatives pour la fonction de coût sans doute la plus cruciale : l'hélice. De plus, chaque fonction renvoie un coût, caractérisant la préférence que l'on a pour les valeurs prises par les variables de la fonction. Ce score est d'autant plus élevé que la préférence est faible. Ces fonctions dépendent parfois du nombre d'erreurs que l'on peut trouver dans un mot ou une hélice. Pour des raisons de simplicité, nous supposerons que toutes les erreurs donnent un coût de un. Mais il est possible de changer ce système de score, en faisant payer, par exemple, moins cher les substitutions que les insertions ou les suppressions.

De plus, pour chaque élément de structure, nous allons proposer trois fonctions de coût possibles, chacune modélisant d'une façon particulière l'élément de structure. Pour cela, nous utiliserons un paramètre spécial, nommé $\top_{\text{loc}}$, que toutes les fonctions de coût utilisent. Ce paramètre désigne le coût maximum qui peut être renvoyé par la

fonction. Par exemple, si l'on cherche un mot avec deux erreurs au plus, alors $\top_{\text{loc}}$ vaudra deux. Cette valeur $\top_{\text{loc}}$ sera, pour chaque fonction de coût, interprétée de deux manières possibles : soit tout coût supérieur à $\top_{\text{loc}}$ est ramené à $\top_{\text{loc}}$, soit tout coût supérieur à $\top_{\text{loc}}$ est augmenté à $\top$, le plus grand coût du réseau de contraintes pondérées, exprimant une incohérence. Dans ce premier cas, et si ce coût de $\top_{\text{loc}}$ est strictement inférieur à $\top$, alors la fonction peut être vue comme une contrainte optionnelle ou comme une préférence pure car, ne donnant jamais un coût de $\top$, elle ne pourra pas éliminer seule une solution potentielle. Dans un deuxième cas, chaque élément de structure peut être modélisé par une préférence avec contrainte ou par préférence souple (si la valeur de $\top_{\text{loc}}$ est interprétée comme une incohérence). Nous ajoutons à cela une troisième modélisation : si le coût donné par la fonction est inférieur à $\top_{\text{loc}}$, alors on renvoie zéro, sinon, on renvoie $\top$. Cette modélisation peut être interprétée comme une contrainte pure, dans la mesure où les coûts qu'elle renvoie sont soit zéro (satisfaction totale), soit $\top$ (incohérence). Ces types de fonction de coût permettent notamment de spécifier des règles inviolables.

Plus formellement, supposons que l'on ait une fonction f et un score maximum $\top_{\text{loc}}$ (comme représenté sur la figure 3.13(a)), alors on peut transformer f de trois façons différentes : en une fonction *optionnelle* (voir figure 3.13(b)), en fonction *dure* (voir figure 3.13(c)), ou bien fonction *souple* (voir figure 3.13(d)). Nous garderons ces termes optionnel, dur ou souple tout au long de ce document. Les trois transformateurs sont appelés T_{option}, T_{souple} et T_{dur}. Ils sont définis, pour une fonction de coût f et tout point a du domaine de f de la manière suivante :

$$T_{\mathsf{optionnel}}(f(a)) = \min\{f(a), \top_{\text{loc}}\},$$

$$T_{\mathsf{dur}}(f(a)) = \begin{cases} 0 & \text{si } f(a) \leq \top_{\text{loc}}, \\ \top & \text{sinon,} \end{cases}$$

$$T_{\mathsf{souple}}(f(a)) = T_{\mathsf{option}}(f(a)) \oplus T_{\mathsf{dur}}(f(a)).$$

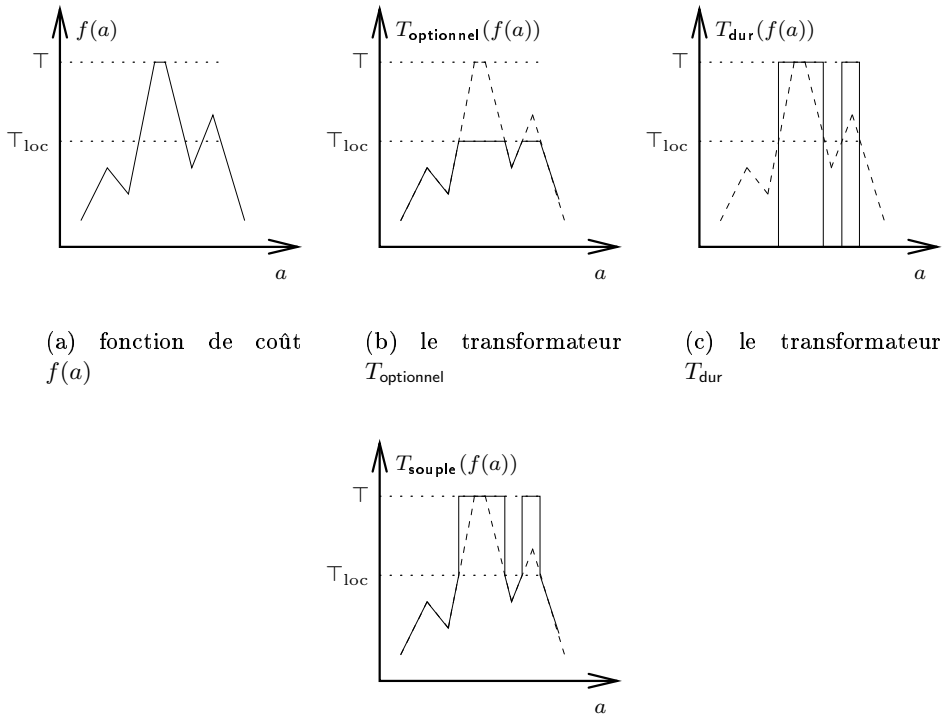
La possibilité de modéliser chaque élément de structure sous trois manières différentes offre un niveau d'expression supplémentaire laissé à l'utilisateur. Ceci permet de réaliser des signatures plus fines.

Les paragraphes suivants détaillent chaque fonction de coût utilisable. Chaque paragraphe présente le nom de la fonction, les paramètres qu'elle reçoit, les variables qu'elle utilise, suivi d'une description du sens de la fonction.

3.4.4.1 Le mot

$\mathtt{mot}[\mathsf{mot}, \top_{\text{loc}}, \mathsf{transformateur}](x_i, x_j)$

La fonction de coût de mot évalue à quel point le mot mot est conservé entre les positions x_i et x_j. Pour les nucléotides connus de façon imprécise, il est possible d'utiliser des nucléotides ambigus, selon la norme IUPAC, présentée dans le paragraphe 1.1.1. Ce mot peut être à rechercher dans la séquence principale, ou dans la séquence cible. Il est possible de spécifier que la fonction accepte des erreurs ($\top_{\text{loc}}$ étant le nombre maximum d'erreurs autorisées), qui peuvent être des insertions, des suppressions et des



(a) fonction de coût $f(a)$

(b) le transformateur $T_{\mathsf{optionnel}}$

(c) le transformateur T_{dur}

(d) le transformateur T_{souple}

FIG. 3.13 – Les différents transformateurs de fonctions.

substitutions. Le coût donné par la fonction est le nombre d'erreurs trouvées entre le mot contenu entre les deux variables de la fonction et le mot donné en paramètre.

Afin de mieux modéliser cet élément de structure et de s'assurer que le mot commence et finit où l'on souhaite, nous avons interdit les insertions et les suppressions au début et à la fin du mot.

Exemple 3.5 *Dans l'exemple du petit ARN à boîte H/ACA, nous pouvons par exemple créer une fonction de coût entre les positions x_9 et x_{10}. Le mot à rechercher pourrait être RUGA, et aucune erreur ne serait acceptée. Dans ce cas, on préfèrerait une modélisation sous forme dure.*

3.4.4.2 Les interactions

Les fonctions de coût suivantes modélisent toutes des interactions, mais chacun à sa spécificité. La fonction `hélice` modélise une hélice classique. La fonction `hélice_alt` privilégie les grandes hélices en attribuant un score plus élevé aux hélices courtes. Le `repliement` évalue le repliement d'une séquence sur elle-même. Le `duplex` modélise des interactions inter-moléculaire. La `paire` modélise une seule interaction éventuellement non-canonique.

hélice `hélice`[taille_min, taille_max, boucle_min, boucle_max, indels, wobble, $\top_{\text{loc}}$, transformateur]$(x_i, x_j, x_k, x_l$
La fonction d'hélice évalue la présence éventuelle d'une hélice contenue entre les quatre variables données en paramètres, dont les deux premières délimitent le début et la fin du premier brin, et les deux dernières délimitent le début et la fin du second brin. Les tailles minimale et maximale de l'hélice doivent aussi être spécifiées en paramètres, ainsi que les tailles minimale et maximale de la boucle. Il est possible d'accepter les liaisons *wobble*, ou de les considérer comme des mésappariements. La fonction accepte également des erreurs, qui peuvent être soit des mésappariements / insertions / suppressions, soit simplement des mésappariements, selon la valeur de indels. Le coût donné par cette fonction correspond au nombre d'erreurs trouvées.

Exemple 3.6 *Cette fonction de coût peut s'utiliser dans l'exemple du pARN. Il existe en effet une hélice entre les positions x_1 et x_2 d'une part, et x_{15} et x_{16} d'autre part. En regardant de plus près les séquences, on voit que les hélices de l'exemple peuvent contenir une insertion et des liaisons wobble. On peut donc caractériser une telle hélice, dont la taille varie entre sept et huit paires de bases, et dont la boucle contient quarante à cinquante-huit nucléotides. Cette fonction de coût pourrait être modélisée sous forme de souple, car si l'on accepte les hélices avec une erreur (et moyennant une pénalité), on n'est pas prêt à accepter d'hélice avec un plus grand nombre d'erreurs.*

Hélice alternative `hélice_alt`[taille_max, wobble, $\top_{\text{loc}}$, transformateur](x_i, x_j, x_k, x_l)
Une autre fonction de coût d'hélice a été créée, afin d'élargir le choix des fonctions de coût. Cette hélice utilise aussi quatre variables, qui délimitent les deux brins de cette hélice. Il est aussi possible d'accepter ou non les liaisons *wobble* et il faut spécifier la

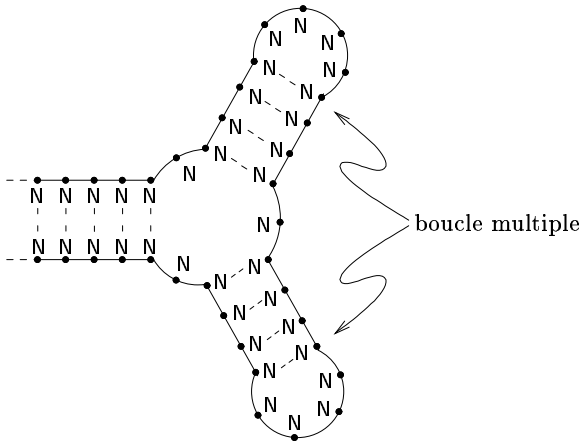


FIG. 3.14 – Un exemple de branchement.

longueur maximale de l'hélice. La spécificité de cette contrainte est que si la taille de l'hélice est inférieure à la longueur maximale donnée, alors une pénalité est donnée à l'hélice : cette pénalité est nulle si l'hélice a la longueur maximale, de un si l'hélice est plus courte d'une paire de base, de deux si l'hélice est plus courte de deux paires de bases, etc. Comme pour les deux fonctions de coût précédentes, la fonction renvoie le nombre d'erreurs trouvées.

L'intérêt de cette fonction de coût est qu'elle favorise les hélices les plus longues en attribuant un coût plus élevé aux hélices courtes. C'est en général ce qui est préféré par l'utilisateur, car les hélices longues sont en général plus stables.

Le repliement `repliement`$[\mathsf{taille\_min}, \mathsf{taille\_max}, \mathsf{wobble}, \top_{\mathrm{loc}}, \mathsf{transformateur}](x_i, x_j)$
Une troisième alternative pour la modélisation d'une hélice est la fonction de repliement, qui évalue le meilleur repliement possible entre les deux variables données en paramètres. Les paramètres donnés en entrée sont les longueurs minimale et maximale du repliement, le statut des liaisons *wobble* et le nombre d'erreurs maximal. Ces erreurs sont de type mésappariements / insertions / suppressions. Comme précédemment, la fonction renvoie le nombre d'erreurs trouvées.

Une des particularités de cette fonction de coût est de permettre la caractérisation de structures complexes, comme les branchements (cf. 3.14), et donc les boucles intérieures.

Le duplex `duplex`$[\mathsf{wobble}, \top_{\mathrm{loc}}, \mathsf{transformateur}](x_i, x_j, y_k, y_l)$
La fonction de coût de duplex évalue l'éventuel duplex situé entre les quatre variables données en paramètre. Les deux premières bornent le duplex sur la séquence principale et les deux dernières bornent ce duplex sur la séquence cible. Le nombre d'erreurs, ainsi que le statut des liaisons *wobble* doit être donné en paramètre. Ici, les erreurs sont de type mésappariements / insertions / suppressions. Comme précédemment, la fonction renvoie le nombre d'erreurs trouvées.

Exemple 3.7 *On peut par exemple modéliser une interaction extra-moléculaire de notre petit ARN à boîte H/ACA grâce à cette fonction. On crée pour cela une fonction de coût entre les positions x_3, x_4, y_5 et y_6, où les liaison wobble sont acceptées, et le nombre maximum d'erreurs est de un. Le choix de modélisation de type souple pourrait être le plus adapté ici.*

Les interactions non-canoniques `paire`[côté1, côté2, orientation, famille, $\top_{\text{loc}}$, transformateur](x_i, x_j)
La fonction de coût d'interactions non-canoniques évalue la possibilité d'apparier deux nucléotides selon des côtés, une orientation et une famille donnés. Ces deux nucléotides sont positionnés par les deux variables sur lesquelles porte la fonction de coût. Il faut alors spécifier le côté de chaque nucléotide interagissant (le lecteur pourra revenir dans le paragraphe 1.1.1 pour se refamiliariser avec la stéréochimie du nucléotide), ainsi que le sens d'interaction. Il faut aussi donner la famille d'isostéricité que l'on souhaite voir apparaître (cf. paragraphe 1.1.1), mais il est possible d'accepter toutes les familles. Le coût donné par la fonction est nul si l'appariement est bien trouvé. Une pénalité est donnée si l'on observe une interaction qui appartient à une autre famille.

Si l'on souhaite modéliser une interaction sous la forme dure, l'interaction observée n'est acceptée que si elle appartient à la famille donnée.

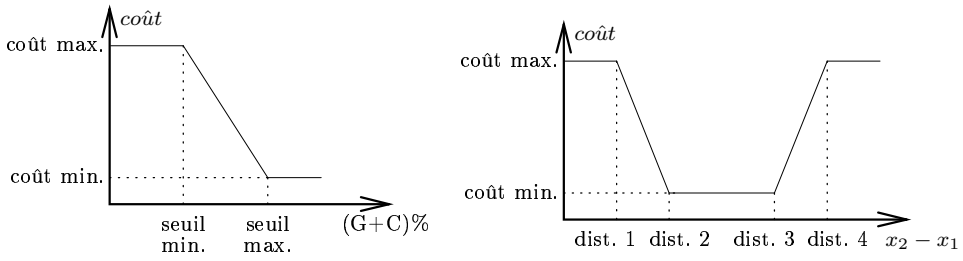
Exemple 3.8 *Dans l'exemple du pARN, si l'on considère les mots conservés entre x_7 et x_8 d'une part, et x_9 et x_{10} d'autre part, on peut voir qu'il s'agit d'une partie de structure appelée k-turn (cf. 3.4). Cette dernière comporte plusieurs appariements non-canoniques, dont par exemple de type Sucre–Hoogsteen en trans, de la famille 2, entre les nucléotides positionnés en x_7 et x_{10}. Cette interaction peut être modélisée par cette fonction de coût, sous la forme dure.*

3.4.4.3 La répétition

`répétition`[taille_min, taille_max, dist_min, dist_max, indels, $\top_{\text{loc}}$, transformateur](x_i, x_j, x_k, x_l)
La fonction de coût de répétition évalue la similarité de deux mots. Cette fonction utilise quatre variables, dont les deux premières délimitent le premier mot, et les deux dernières, le second. Les paramètres de la fonction incluent les tailles minimale et maximale de la longueur de mot, ainsi que les distances minimale et maximale entre les deux occurrences du mot. Il faut aussi donner le type d'erreurs (à choisir entre substitutions uniquement, ou bien substitutions / insertions / suppressions). La fonction renvoie le nombre d'erreurs trouvées.

3.4.4.4 La composition

`composition`[nucléotides, relation, seuil1, seuil2, coût_min, $\top_{\text{loc}}$, transformateur](x_i, x_j)
La fonction de coût de composition évalue à quel point la proportion de certains nucléotides est supérieure ou inférieure à des seuils donnés. Dans la pratique, la fonction de coût attend un opérateur de comparaison ($\leq$ ou $\geq$), deux seuils (un seuil minimum et un seuil maximum) et deux coûts (un coût minimum et un coût maximum). Si l'opérateur



(a) fonction de coût de composition

(b) fonction de coût d'espaceur

FIG. 3.15 – Deux fonctions de coût.

de comparaison est $\geq$, alors le coût maximal sera donné si la proportion de nucléotides descend en dessous du seuil minimal. Elle donne le coût minimal si la proportion passe au-dessus du seuil maximal. Entre les seuils, la fonction est prolongée par continuité, de façon à garder une fonction affine par morceaux (cf. figure 3.15(a)). Les nucléotides donnés en paramètres servent le plus souvent à définir le (G+C)% ou le (A+T)% et sont donc souvent CG ou AT.

Si l'on souhaite modéliser la composition sous la forme dure, c'est-à-dire si l'on veut que le score renvoyé soit zéro ou $\top$, alors un seul seuil est donné. Toute proportion des nucléotides en deçà du seuil sera interprétée comme une violation.

Exemple 3.9 *La densité de nucléotides G et C entre les positions x_1 et x_2 peut être prise en compte par cette fonction de coût. On peut par exemple imposer une proportion minimale de 75 % et une proportion maximale de 100 %, les coûts possibles allant de zéro à deux. On modélise alors cet élément de structure sous la forme souple.*

3.4.4.5 L'espaceur

espaceur[dist1, dist2, dist3, dist4, seuil1, seuil2, coût_min, $\top_{\text{loc}}$, transformateur](x_i, x_j)
La fonction d'espaceur évalue la distance relative entre deux variables-positions. Cette fonction utilise comme paramètres quatre distances et deux coûts (un coût minimal et un maximal). Si les deux variables sont trop proches (c'est-à-dire si la distance entre les deux variables est inférieure à dist1), ou si les deux variables sont trop éloignées (si la distance entre les deux variables est supérieure à dist4), alors le coût donné est le coût maximal. Si la distance entre ces variable est la distance désirée (c'est-à-dire, si elle se situe entre dist2 et dist3), alors le coût minimal est attribué. Pour les autres cas, on prolonge la fonction par continuité, de manière à garder une fonction affine par morceaux (cf. figure 3.15(b)).

Si l'on souhaite modéliser l'espaceur sous la forme dure, seules dist2 et dist3 sont utilisées. Elles donnent les distances minimale et maximale que l'on peut avoir entre les deux variables données en paramètres.

Exemple 3.10 *Dans notre exemple du pARN à boîte H/ACA, on peut aussi utiliser quelques contraintes dures d'espacement. La contrainte d'espacement permet de contraindre la distance entre les positions x_8 et x_9. Les distances autorisées vont alors de quatre à vingt-deux. Il ne semble en revanche pas exister de distance préférencielle entre x_8 et x_9. Dans ce cas, nous n'utilisons pas de fonction de coût supplémentaire.*

3.4.4.6 Traduction nombre d'erreurs / scores

Afin de permettre encore plus de souplesse sur les scores des fonctions de coûts, nous avons donné à l'utilisateur la possibilité de définir pour chaque fonction de coût une fonction qui, à un nombre d'erreurs donné, associe un score. L'utilisateur peut alors spécifier que si le nombre d'erreurs est n, alors le score retourné sera $f_{\text{trad}}(n)$, où f_{trad} est une fonction donnée en extension par l'utilisateur.

Cette traduction est utilisable sur toutes les fonctions de coût où les scores renvoyés sont basés sur un nombre d'erreurs : mots, hélices de tout type, duplex, répétitions. Il existe toutefois une restriction importante à ce mécanisme : f_{trad} doit absolument, pour des raisons algorithmiques, être une fonction croissante. Néanmoins, si la signature est correctement définie, il est logique de préférer les solutions ayant un nombre minimal d'erreurs.

3.5 Conclusion

Nous avons ici présenté les modélisations proposées par David Gilbert et Patricia Thébault du problème de la localisation d'ARN. Tous deux utilisent les réseaux de contraintes. Nous avons vu que ces approches, si elles donnaient de bons résultats, manquaient d'expressivité par le fait qu'elles ne pouvaient pas gérer de mécanisme de coût. Nous avons donc utilisé un formalisme, appelé réseau de contraintes pondérées, qui étend celui des réseaux de contraintes et qui gère les scores. Nous avons donc repris et étendu les modélisations proposées, et nous avons ajouté quelques fonctions de coût supplémentaires, contrôlant le (G+C)% ou les appariements non-canoniques. Nous avons vu que l'utilisation des coûts permettait notamment d'exprimer des préférences, ce qui représente un pouvoir d'expressivité important pour le modélisateur.

Deux questions restent encore en suspens. La première est de savoir si l'on n'a pas oublié d'élément de structure à modéliser. À ce jour, nous avons par exemple pensé à une variante de la fonction de coût de duplex, qui prendrait en entrée non pas une seule, mais plusieurs séquences cibles. Cela permettrait de traiter plus simplement les cas où un ARN peut s'apparier avec un des ARN de transfert (il y en a au moins une vingtaine dans chaque organisme), ce qui est le cas des petits ARN à boîte C/D [BCH02].

Il serait également intéressant de complexifier les fonctions de coût. Prenons par exemple le cas du mot. Une nouvelle fonction de coût pourrait prendre en entrée non pas un mot-consensus, mais une matrice de poids position-spécifique (présentée dans le paragraphe 2.2.2.1). Le coût donné par la fonction dépendrait du score donné par la

matrice. Le choix des chaînes de Markov cachées (présentées dans le même paragraphe) pourrait aussi être intéressant pour autoriser les insertions et les suppressions.

Une seconde question est le choix du formalisme. Utiliser un cadre d'optimisation combinatoire dont la question de décision associée est NP-complet semble être plus à même de répondre à problèmes réalistes, notamment par rapport aux formalismes polynomiaux. Mais est-ce suffisant ? On ne peut par exemple pas modéliser simplement de disjonction : on recherche soit un mot, soit une hélice. D'autres recherches complémentaires pourraient s'orienter sur les mécanismes supportant ce type de spécification comme [HSD95] ou [Lho03].

Chapitre 4

Algorithmes de résolution de réseaux de contraintes pondérées

Nous avons détaillé dans le chapitre précédent la modélisation de notre problème dans le cadre des réseaux de contraintes pondérées. L'apport d'un tel formalisme est multiple. Il offre tout d'abord un cadre mathématique rigoureux, il permet de modéliser des éléments de structure par une fonction de coût, et donc l'ajout d'un élément de structure se fait simplement par l'implantation d'une nouvelle fonction de coût. De plus, la recherche en intelligence artificielle a proposé, notamment durant la dernière décennie, de nombreuses méthodes de résolution efficaces des réseaux de contraintes pondérées. Enfin, par rapport aux réseaux de contraintes classiques, la possibilité de donner un coût aux solutions prédites permet de rendre encore plus pertinent le modèle.

Dans la pratique, les réseaux de contraintes, pondérées ou non, utilisent des techniques dites de *filtrage par cohérence locale* pour accélerer la recherche. Nous avons présenté dans le paragraphe 3.3.1.3 quelques méthodes de filtrages par cohérence locale applicables aux réseaux de contraintes classiques. Nous avons, dans le paragraphe 3.4.2.2, mentionné qu'il en existait aussi dans les réseaux de contraintes pondérées, sans en avoir détaillé aucune. Nous les présenterons ici.

Ce chapitre s'articule de la manière suivante. Nous présenterons le principe du filtrage par cohérence locale dans le cadre des réseaux de contraintes pondérées. Nous présenterons dans un premier temps les filtrages existants. Nous présenterons ensuite deux filtrages que nous avons conçu dans le cadre de nos recherches : le filtrage par *cohérence existentielle*, qui, même s'il ne peut s'appliquer à la recherche d'ARNnc, donne de bons résultats dans d'autres types de problèmes et le filtrage par *cohérence d'arc aux bornes*, que nous avons développé pour la recherche d'ARNnc. Nous comparerons ce dernier filtrage avec une autre modélisation se basant sur les réseaux de contraintes classiques.

Les algorithmes que nous allons présenter sont souvent nommés par un acronyme. Il est parfois d'usage d'utiliser en français les acronymes anglais. Nous avons toutefois préféré, à chaque fois que c'était possible, les traduire en français, afin de privilégier une certaine cohérence linguistique.

4.1 Propriétés de cohérence locale

Nous détaillons ici les principales méthodes de filtrage applicables aux réseaux de contraintes pondérées.

4.1.1 Propriétés classiques

4.1.1.1 Cohérence de nœud (CN)

Une des propriétés les plus simples que l'on puisse établir est la *cohérence de nœud* (CN) [Lar02]. Elle est appelée *node consistency* ou *NC* en anglais.

Définition 4.1 *Une variable x_i est nœud-cohérente si :*
- $\exists v_i \in D_i, c_i(v_i) = 0$ *(v_i est alors appelé* support unaire *de x_i) et*
- $\forall v_i \in D_i, c_\varnothing \oplus c_i(v_i) < \top$.

Un réseau est nœud-cohérent si toutes ses variables le sont.

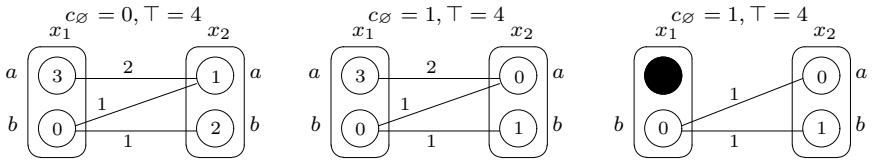
Une valeur non valide *est une valeur qui ne respecte pas la seconde assertion de CN.*

Dans la définition précédente, c_i correspond à la fonction de coût unaire dont la portée est exactement x_i, et $c_\varnothing$ est la fonction de coût d'arité nulle. L'exemple suivant illustre cette propriété, ainsi que le rôle particulièrement important que joue cette dernière fonction de coût.

Exemple 4.1 *Nous reprenons ici l'exemple 3.2. On peut remarquer que la variable x_2 ne respecte pas la première règle de la définition : tous les coûts unaires sont strictement positifs. Étant donné que x_2 devra être affectée, cela signifie que toute solution aura un coût minimum de un. Ce coût représente donc un minorant du coût de toutes les solutions, nous voulons donc augmenter $c_\varnothing$ de un. Afin de conserver l'équivalence du problème d'origine et du problème de départ, il nous faut donc retrancher ce coût de un à tous les coûts unaires de x_2. Cette opération s'appelle la* projection unaire, *implantée dans la fonction 7 (*ProjectionUnaire*). Le résultat de son exécution est donné figure 4.1(b).*

*On peut maintenant voir que la valeur a de la variable x_1 ne respecte pas la seconde propriété de la définition de CN. De fait, le coût de l'affectation x_1 à a vaut $c_\varnothing + c_1(a) = 1 + 3 = 4 = \top$. L'affectation $a \leftarrow x_1$ ne peut donc participer à aucune solution du problème ; la valeur peut donc être effacée, ce que fait la fonction 8 (*SuppressionValeurs*). Le résultat est donné dans la figure 4.1(c), qui est CN.*

La fonction 7 (ProjectionUnaire) établit un support unaire et donc la première propriété de la définition de CN. Sa valeur de retour, pour l'instant inutilisée, s'avèrera importante par la suite. La fonction 8 (SuppressionValeurs) enlève les valeurs dont le coût unaire est trop élevé. L'intérêt de la valeur de *drapeau* sera aussi révélé par la suite. Les preuves de correction et la recherche des complexités théoriques en temps et en mémoire ont été données dans [Lar02]. Nous les rappellerons ici car elles seront utilisées dans les algorithmes suivants.



(a) Problème originel.

(b) Établissement de la première propriété de CN.

(c) Établissement de la seconde propriété de CN.

FIG. 4.1 – Établissement de CN

Propriété 4.1 *L'algorithme de la procédure 9 (*CN*) est correct. Les complexités temporelle et spatiale de l'algorithme établissant CN sont $\mathcal{O}(nd)$, où n est le nombre de variables du problème, et d la taille maximale des domaines.*

Fonction 7 – ProjectionUnaire($x_i \in \mathcal{X}$) : booléen

$min \leftarrow \min_{v_i \in D_i}\{c_i(v_i)\}$;
pour chaque $v_i \in D_i$ **faire**
  $c_i(v_i) \leftarrow c_i(v_i) \ominus min$;
1 $c_\varnothing \leftarrow c_\varnothing \oplus min$;
2 **si** $(c_\varnothing = \top)$ **alors**
  lever incohérence ;
retourner $(min > 0)$;

Fonction 8 – SuppressionValeurs($x_i \in \mathcal{X}$) : booléen

$drapeau \leftarrow$ faux ;
pour chaque $v_i \in D_i$ **faire**
  si $(c_\varnothing \oplus c_i(v_i) = \top)$ **alors**
    $drapeau \leftarrow$ vrai ;
    $D_i \leftarrow D_i \setminus \{v_i\}$;
si $(D_i = \varnothing)$ **alors**
  lever incohérence ;
retourner $drapeau$;

Preuve 4.1 ProjectionUnaire(x_i) établit la première propriété de la définition de CN sur la variable x_i, et SuppressionValeurs(x_i) établit bien la seconde propriété. La question qui reste à se poser est de savoir si la seconde fonction n'efface pas un support unaire. Il faudrait dans ce cas rétablir le support unaire de la variable. Ce n'est pas le cas, car SuppressionValeurs n'enlève une valeur v_i de x_i que si $c_\varnothing \oplus c_i(v_i) = \top$. Si v_i est un

Procédure 9 – CN

pour chaque $x_i \in \mathcal{X}$ **faire**
  ProjectionUnaire(x_i) ;
pour chaque $x_i \in \mathcal{X}$ **faire**
  SuppressionValeurs(x_i) ;

support unaire, alors $c(v_i) = 0$, et donc $c_\varnothing = \top$. Cela signifierait que notre problème est incohérent. Si c'était le cas, alors la ligne **2** aurait levé une exception, puisque la seule ligne qui fait augmenter $c_\varnothing$ est la ligne **1**. SuppressionValeurs ne détruit donc pas de support unaire.

Concernant la complexité temporelle, SuppressionValeurs et ProjectionUnaire s'établissent en temps $\mathcal{O}(d)$, où d est la taille du plus grand domaine. La complexité temporelle est donc de $\mathcal{O}(nd)$. Étant donné que l'on peut enlever de chaque domaine toute valeur, la complexité spatiale est elle aussi de $\mathcal{O}(nd)$. □

4.1.1.2 Cohérence d'arc (CA)

Il existe aussi une autre propriété de cohérence locale, plus forte que la précédente, dont l'équivalent dans les réseaux de contraintes classiques est très utilisé : la *cohérence d'arc* (CA), appelée *arc consistency* ou *AC* en anglais [Sch00, Lar02]. Cette propriété fait intervenir le *voisinage* d'une variable.

Définition 4.2 *Le voisinage d'une variable x_i est l'ensemble des variables défini par :*

$$V(x_i) = \{x_j \in \mathcal{X} \setminus \{x_i\} | \exists c \in \mathcal{C}, \{x_i, x_j\} \subseteq var(c)\}$$

Définition 4.3 *Une variable x_i est arc-cohérente si :*
- $\forall c \in \mathcal{C}, x_i \in var(c) \Rightarrow \forall v_i \in D_i, \exists a \in \ell(var(c) \setminus \{x_i\}), c(a \cup \{(x_i \leftarrow v_i)\}) = 0$ *(a est alors appelé* support *de v_i) et*
- *x_i est nœud-cohérente.*

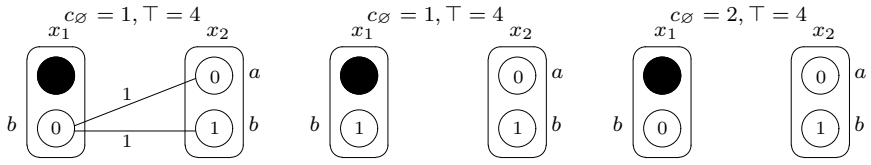
Un réseau est arc-cohérent si toutes ses variables le sont.

Remarque 4.1 *Si le réseau est* binaire*, c'est-à-dire si toutes les fonctions de coût ont une arité d'au plus deux, alors la première propriété de la définition de CA se réécrit en :*

$$\forall x_j \in V(x_i), \forall v_i \in D_i, \exists v_j \in D_j, c_{ij}(v_i, v_j) = 0$$

L'exemple suivant illustre cette propriété.

Exemple 4.2 *Nous reprenons le petit problème de la figure précédente, dessiné en 4.2(a). Il est à noter que tous les coûts de $c_{1,2}$ partant de la valeur b de x_1 sont strictement positifs. Cela ne respecte pas la première propriété de la définition de CA. De fait, toute solution où l'on affecte x_1 à b donnera au minimum un coût de 1, puisqu'il faudra aussi affecter x_2 à une valeur. Afin de « simplifier » le problème, on peut augmenter le coût*



(a) Problème originel.

(b) Établissement de la première propriété de CA.

(c) Établissement de CN

FIG. 4.2 – Établissement de CA

unaire de (x_1, b) *de* 1. *C'est ce que l'on fait, en prenant soin de retrancher ce coût de la fonction binaire. Cette opération s'appelle la* projection, *et elle est établie par la fonction 10. Le nouveau problème, est dessiné en 4.2(b). Après application de CN (voir figure 4.2(c)), le problème est CA.*

Fonction 10 – Projection($x_i \in \mathcal{X}, x_j \in V(x_i), v_i \in D_i$) : booléen

```
drapeau ← faux ;
min ← min_{v_j ∈ D_j} {c_ij(v_i, v_j)} ;
si (min > 0) alors
    drapeau ← (c_i(v_i) = 0) ;
    pour chaque v_j ∈ D_j faire
        c_ij(v_i, v_j) ← c_ij(v_i, v_j) ⊖ min ;
    c_i(v_i) ← c_i(v_i) ⊕ min ;
retourner drapeau ;
```

L'algorithme 12 (CA) établit la cohérence d'arc et nous allons prouver sa correction, initialement démontrée dans [Lar02]. Cette propriété sera elle aussi réutilisée par la suite.

Fonction 11 – ÉtablissementSupport($x_i \in \mathcal{X}, x_j \in V(x_i)$) : booléen

```
drapeau ← faux ;
pour chaque v_i ∈ D_i faire
    drapeau ← drapeau ∨ Projection(x_i, x_j, v_i) ;
retourner drapeau ;
```

Propriété 4.2 *L'algorithme* CA *décrit dans la procédure 12 est correct.*

Preuve 4.2 La cohérence d'arc utilise les nouvelles fonctions Projection(x_i, x_j, v_i) et ÉtablissementSupp
La première fonction établit le support de la valeur v_i de x_i par rapport à la fonction de

Procédure 12 – CA

```
3  CN ;
   Q ← 𝒳 ;
4  tant que (Q ≠ ∅) faire
       x_j ← Extraire(Q) ;
       pour chaque x_i ∈ V(x_j) faire
5          si (ÉtablissementSupport(x_i, x_j)) alors
6              ProjectionUnaire(x_i) ;
7          si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
       pour chaque x_i ∈ 𝒳 faire
8          si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
```

coût c_{ij}. Elle renvoie le booléen vrai si la projection a fait augmenter le coût unaire de v_i de zéro à une valeur strictement positive. La seconde fonction, ÉtablissementSupport, établit un support pour toutes les valeurs de x_i par rapport à c_{ij}, en utilisant la première fonction. Elle renvoie vrai si l'une des projections a fait augmenter un coût unaire de zéro à une valeur strictement positive. L'intérêt de cette valeur de retour apparaitra sous peu.

L'ensemble Q regroupe toutes les variables telles qu'une variable voisine pourrait avoir perdu un support par rapport à elle. En d'autres termes, à la ligne **4** :

$$x_j \notin Q \Rightarrow \forall x_i \in V(x_j), \forall v_i \in D_i, \exists v_j \in D_j, c_{ij}(v_i, v_j) = 0$$

Vérifions que ceci est vrai. Au début, toutes les variables sont dans Q, et la propriété est donc vraie. Lorsqu'une variable est dans Q, si l'algorithme termine, cette variable sera dépilée et ÉtablissementSupport vérifiera tous les supports de ses voisins par rapport à elle. L'algorithme établit donc bien CA sur les voisins de x_i.

Au cours de l'algorithme, des propriétés peuvent être brisées. Recherchons les cas où la valeur d'une variable perd son support. Les fonctions de coût binaires ne sont modifiées que par la fonction Projection, et donc les coûts binaires décroissent lors de l'application de CA. Il n'existe alors qu'un seul moyen pour une variable de perdre un support : ce support a été supprimé par la fonction SuppressionValeurs. Lorsqu'une valeur est supprimée, la fonction SuppressionValeurs retourne vrai, et la variable contenant cette valeur est empilée dans Q, car une voisine pourrait avoir perdu un support. Donc quand l'algorithme termine, Q est vide et toutes les valeurs ont un support par rapport à toutes les variables voisines.

Vérifions maintenant que toutes les variables sont aussi CN. Elles le sont au début de l'algorithme, grâce à l'appel à CN de la ligne **3**. La première propriété de CN, forçant l'existence d'un support unaire, peut être détruite lorsqu'un coût unaire augmente. Ce n'est le cas que lors d'une projection. Si le support unaire d'une variable a disparu,

cela implique que l'on a fait augmenter un coût unaire de zéro à une valeur strictement positive, et donc ÉtablissementSupport a donné une valeur de retour valant vrai. Dans ce cas, la fonction ProjectionUnaire est appelée à la ligne **6**.

Il est de plus possible de rendre un coût unaire non valide, c'est à dire de briser la seconde propriété de CN soit en faisant augmenter $c_\varnothing$, soit en faisant augmenter le coût unaire d'une variable. Seule ÉtablissementSupport peut augmenter un coût unaire, et seule ProjectionUnaire peut augmenter $c_\varnothing$. Au cas où ÉtablissementSupport aurait rendu une valeur non valide, on appelle SuppressionValeurs à la ligne **7** sur la variable dont un coût unaire pourrait avoir augmenté. Et au cas où ProjectionUnaire aurait fait augmenter $c_\varnothing$, on applique SuppressionValeurs à la ligne **8** sur toutes les valeurs de toutes les variables. À la fin de l'algorithme, chaque variable est donc CN. Ceci prouve la propriété. □

Nous donnons ici la complexité temporelle de l'algorithme.

Propriété 4.3 *L'algorithme précédent termine en temps $\mathcal{O}(ed^3 + n^2d^2)$, où e est le nombre de fonctions de coût du problème, n, le nombre de variables, et d la taille maximale des domaines.*

Preuve 4.3 Tout d'abord, les fonctions ProjectionUnaire, Projection et SuppressionValeurs ont une complexité temporelle de $\mathcal{O}(d)$, et donc ÉtablissementSupport a une complexité temporelle de $\mathcal{O}(d^2)$.

Tentons maintenant de trouver combien de fois la ligne **5** est appelée. Considérons une fonction de coût binaire c_{ij}. Dans la fonction ÉtablissementSupport, l'algorithme peut être amené à trouver des supports de x_i ou x_j, selon que l'une ou l'autre des variables a été dépilée de Q. Or une variable n'est insérée dans Q qu'au début, ou bien lorsqu'une de ses valeurs a été supprimée (lignes **7** et **8**). Donc, chaque fonction de coût est examinée au maximum $2(d+1)$ fois pour l'établissement des supports, et donc la ligne **5** est appelée au maximum $2e(d+1)$ fois.

Étant donné que chaque variable est empilée au plus $d+1$ fois dans Q, la boucle de la ligne **4** est appelée au plus $n(d+1)$ fois, et la ligne **7**, $n^2(d+1)$ fois. La complexité de l'algorithme est donc $\mathcal{O}(ed^3 + n^2d^2)$. □

Une légère amélioration apportée à l'algorithme établissant CA consiste à remarquer qu'il ne faut revérifier tous les coûts unaires que lorsque $c_\varnothing$ augmente. La procédure 13 (CA2) décrit ce nouvel algorithme. Les lignes inchangées par rapport à la procédure 12 (CA) ont été grisées. La complexité temporelle passe de $\mathcal{O}(ed^3 + n^2d^2)$ à $\mathcal{O}(ed^3 + \min\{\top, nd\} \times nd)$.

Preuve 4.4 Nous avons vu dans la preuve précédente que, lorsque $c_\varnothing$ augmente, il faut revérifier la validité de tous les coûts unaires. À la différence de l'algorithme CA, CA2 utilise un booléen *drapeau* qui est vrai lorsqu'au moins un appel ProjectionUnaire renvoie vrai. C'est le cas lorsque $c_\varnothing$ a augmenté.

La condition de la ligne **9** ne peut pas être vraie plus de $\top$ fois (car sinon $c_\varnothing > \top$), ou nd fois (les domaines sont sinon tous vides). La complexité de la ligne **10** passe donc à $\mathcal{O}(\min\{\top, nd\} \times nd)$, et celle de l'algorithme complet, à $\mathcal{O}(ed^3 + \min\{\top, nd\} \times nd)$. □

Procédure 13 – CA2

```
   Q ← 𝒳 ;
   CN ;
   tant que (Q ≠ ∅) faire
       x_j ← Extraire(Q) ;
       drapeau ← faux ;
       pour chaque x_i ∈ V(x_j) faire
           si (ÉtablissementSupport(x_i, x_j)) alors
               drapeau ← drapeau ∨ ProjectionUnaire(x_i) ;
           si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
9      si (drapeau) alors
           pour chaque x_i ∈ 𝒳 faire
               si (SuppressionValeurs(x_i)) alors
10                 Q ← Q ∪ {x_i} ;
```

Cette optimisation, que nous avons mise en évidence dans le cadre de nos recherche, n'apporte d'amélioration que du point de vue théorique. En effet, le solveur que nous utilisons, TOULBAR[1] [LS03, HLdGZ05], implante déjà un tel mécanisme.

Il reste encore à déterminer la complexité spatiale des algorithmes. La complexité des algorithmes précédemment décrits est de $\mathcal{O}(ed^2)$, puisque les coûts binaires sont modifiés. [CS04] décrit comment réduire cette complexité. Dans la pratique, on ne modifie pas la valeur des coûts dans Projection, mais on utilise de nouvelles structures de données $\Delta(x_i, x_j, v_i)$, stockant le coût qui a été projeté des fonctions de coût binaires c_{ij} sur la valeur $v_i \in D_i$. Le code réellement utilisé est décrit dans la fonction 14 (Projection2).

Les matrices de coûts données en entrée de l'algorithme ne sont donc pas modifiées, et nous nommerons par exemple $\mathbf{c_{ij}}(v_i, v_j)$ le coût originel de l'affectation de $(x_i \leftarrow v_i, x_j \leftarrow v_j)$ donné par la fonction de coût c_{ij}. $c_{ij}(v_i, v_j)$ désignera donc quant à lui le coût courant de cette affectation, qui vaut $\mathbf{c_{ij}}(v_i, v_j) \ominus (\Delta(x_i, x_j, v_i) \oplus \Delta(x_i, x_j, v_j))$. Ne sont donc modifiés dans tout l'algorithme que les Δ, ainsi que les coûts unaires et $c_\varnothing$. La complexité spatiale devient ainsi $\mathcal{O}(ed)$.

Si $\top$ vaut un, ou si tous les coûts sont soit $\top$, soit zéro, appliquer la cohérence d'arc dans le RCP revient à appliquer la cohérence d'arc dans le réseau de contraintes classiques équivalent. Il existe toutefois une différence de magnitude d'ordre d, lorsque l'on compare les complexités temporelles des deux algorithmes : dans le cas classique, CA peut s'établir en temps $\mathcal{O}(ed^2)$. L'optimalité de la cohérence d'arc dans les réseaux de contraintes pondérées n'a pas été prouvée, mais la différence de complexité des algorithmes peut s'expliquer ainsi. Dans le cas classique, en utilisant des algorithmes de type AC2001/3.1 [BRYZ05], le temps passé à trouver le support d'une valeur par rapport à une contrainte binaire se fait en temps cumulé linéaire par rapport à la taille du

[1] Accessible sur http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro.

Fonction 14 – Projection2($x_i \in \mathcal{X}, x_j \in V(x_i), v_i \in D_i$) : booléen

```
drapeau ← faux ;
min ← min_{v_j ∈ D_j} {c_ij(v_i, v_j) : v_j ∈ D_i} ;
si (min > 0) alors
    drapeau ← (c_i(v_i) = 0) ;
    Δ(x_i, x_j, v_i) ← Δ(x_i, x_j, v_i) ⊕ min ;
    c_i(v_i) ← c_i(v_i) ⊕ min ;
retourner drapeau ;
```

plus grand domaine. Pour arriver à cette complexité, AC2001/3.1 utilise le fait qu'une valeur est ou n'est pas le support d'une valeur d'une autre variable. Dans les réseaux de contraintes pondérées, le problème est légèrement différent : une valeur, qui initialement n'est pas un support peut *devenir* un support (suite à une projection, notamment). Ceci peut expliquer le surcoût en d de la complexité dans le cas pondéré.

Une autre différence entre les deux formalismes est que, dans le cas classique, il n'existe qu'une seule fermeture arc-cohérente, ce qui n'est pas vrai dans le cas pondéré (cf. [Sch00, CS04]). En d'autres termes, l'établissement de la cohérence d'arc dans le cas pondéré ne conflue pas et l'ordre des opérations établissant CA a donc une importance.

4.1.1.3 Cohérence d'arc complète (CAC)

Si l'on veut maintenant chercher à définir une propriété de cohérence locale plus forte, on peut tenter de définir la *cohérence d'arc complète* dans les réseaux de contraintes pondérées binaires. Cette propriété est appelée *full arc consistency* ou *FAC* en anglais.

Définition 4.4 *Une variable x_i est complètement arc-cohérente (CAC) si :*
- $\forall v_i \in D_i, \forall x_j \in V(x_i), \exists v_j \in D_j, c_{ij}(v_i, v_j) \oplus c_j(v_j) = 0$ *(v_j est appelé* support complet *de v_i par rapport c_{ij}) et*
- *x_i est nœud-cohérente.*

Un réseau est complètement arc-cohérent si toutes ses variables le sont.

Malheureusement, il existe des réseaux qui n'ont pas de fermeture CAC. Le théorème suivant le prouve.

Propriété 4.4 *Le problème de la figure 4.3 ne contient pas de fermeture CAC.*

Preuve 4.5 Toute propriété de cohérence locale doit laisser inchangée la distribution des coûts sur l'ensemble des affectations. La propriété n'ajoute pas de variable au problème, ni de valeur. On a donc les égalités suivantes :

$$c_\varnothing \oplus c_1(a) \oplus c_{12}(a,a) \oplus c_2(a) = 1, \tag{4.1}$$
$$c_\varnothing \oplus c_1(a) \oplus c_{12}(a,b) \oplus c_2(b) = 1, \tag{4.2}$$
$$c_\varnothing \oplus c_1(b) \oplus c_{12}(b,a) \oplus c_2(a) = 0, \tag{4.3}$$
$$c_\varnothing \oplus c_1(b) \oplus c_{12}(b,b) \oplus c_2(b) = 1. \tag{4.4}$$

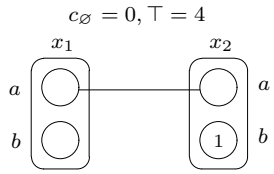


FIG. 4.3 – Un réseau n'admettant pas de fermeture CAC.

Étant donné que tous les coûts sont positifs ou nuls, alors la formule 4.3 implique :

$$c_\varnothing = c_1(b) = c_{12}(b,a) = c_2(a) = 0 \tag{4.5}$$

Avec les égalités 4.5 et en observant que tous les coûts ne peuvent pas valoir plus de un, les formules 4.1, 4.2 et 4.4 peuvent se réécrire de la manière suivante :

$$c_1(a) + c_{12}(a,a) = 1, \tag{4.6}$$
$$c_1(a) + c_{12}(a,b) + c_2(b) = 1, \tag{4.7}$$
$$c_{12}(b,b) + c_2(b) = 1. \tag{4.8}$$

Traduisons maintenant le fait que $(x_1 \leftarrow a)$ et $(x_2 \leftarrow b)$ ont un support complet :

$$\exists v_j \in D_j, c_{12}(a,v_j) \oplus c_2(v_j) = 0$$
$$\exists v_i \in D_i, c_{12}(v_i,b) \oplus c_1(v_i) = 0$$

Comme chaque variable n'a que deux valeurs possibles, et en exploitant la forumule 4.5, il est facile de donner tous les cas possibles des deux formules précédentes :

$$c_{12}(a,a) = 0 \quad \vee \quad c_{12}(a,b) + c_2(b) = 0 \tag{4.9}$$
$$c_{12}(a,b) + c_1(a) = 0 \quad \vee \quad c_{12}(b,b) = 0 \tag{4.10}$$

Notons que si $c_{12}(a,a) = 0$, alors par l'égalité 4.6, $c_1(a) = 1$. De même, si $c_{12}(a,b) + c_2(b) = 0$, alors par l'égalité 4.7, $c_1(a) = 1$. Donc, par la formule 4.9, $c_1(a) = 1$. Dans ce cas :

$$\begin{aligned} c_{12}(b,b) &= 0 && \text{en appliquant la formule 4.10} \\ c_2(b) &= 1 && \text{par 4.8, et donc} \\ c_1(a) + c_{12}(a,b) + c_2(b) &\geq 2 \end{aligned}$$

Ce qui est en contradiction avec 4.7. Il n'existe donc pas de fermeture CAC pour l'exemple donné. □

Cette propriété de cohérence locale semble donc inutile en pratique. On peut toutefois garder l'idée de support complet et tenter de l'utiliser dans une autre cohérence locale. C'est ce que nous ferons dans les deux prochains paragraphes.

4.1.1.4 Cohérence d'arc directionnelle (CAD)

Il est possible d'établir une propriété de cohérence locale plus faible que la cohérence d'arc complète mais qui utilise aussi l'idée de supports complets : la *cohérence d'arc directionnelle* (CAD) [Coo03]. Celle-ci n'établit la cohérence d'arc complète que dans un sens. Cette propriété est appelée *directional arc consistency* ou *DAC* en anglais.

Définition 4.5 *On ordonne ici toutes les variables du problème selon un ordre arbitraire. Par convention, nous dirons ici que $x_i \prec x_j$ si $i < j$. On définit de plus le voisinage inférieur $V^-(x_i)$ de la variable x_i par l'ensemble :*

$$V^-(x_i) = V(x_i) \cap \{x_j \in \mathcal{X} | x_j \prec x_i\}$$

On définit le voisinage supérieur par symétrie.

Une variable x_i est directionnellement arc-cohérente si :

- $\forall x_j \in V^+(x_i), \forall v_i \in D_i, \exists v_j \in D_j, c_{ij}(v_i, v_j) \oplus c_j(v_j) = 0$ *(v_j est alors appelé le* support complet *de v_i) et*
- *x_i est nœud-cohérente.*

Un réseau est directionnellement arc-cohérent si toutes ses variables le sont.

L'exemple suivant illustre cette propriété.

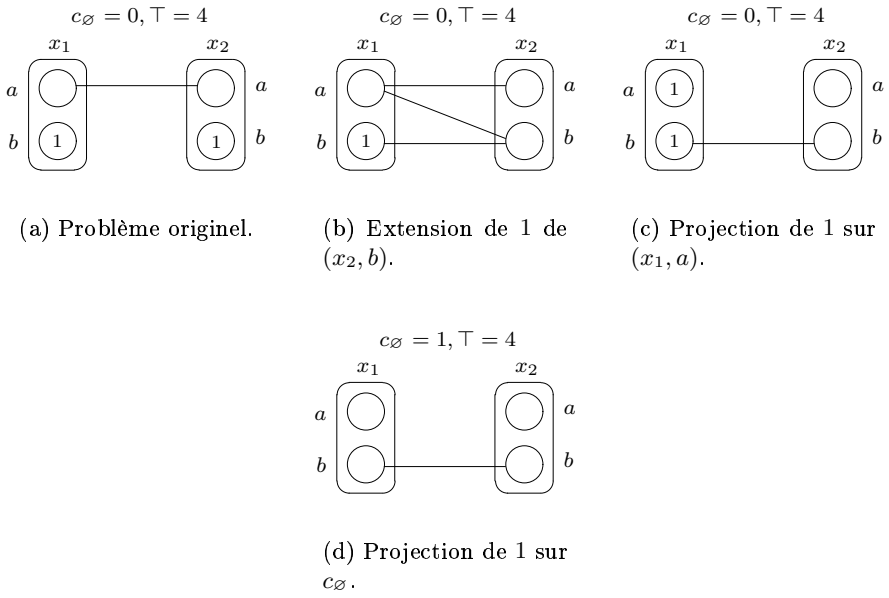
Exemple 4.3 *Considérons le problème 4.4(a). Nous pouvons noter que la valeur a de x_1 n'a pas de support complet par rapport à x_2 : si x_2 prend la valeur a, alors la fonction de coût binaire donne un, si x_2 prend la valeur b, alors la fonction de coût unaire donne un. Pour n'importe quelle valeur de x_2, si x_1 vaut a, alors on aura un coût de un. Nous voulons rendre ceci plus explicite en* étendant *un coût de un de la valeur b de x_2 sur la fonction de coût binaire (cf. figure 4.4(b)). L'extension est l'opération duale de la projection. On peut ensuite projeter un coût de un sur la valeur a de x_1. C'est ce que l'on fait dans la figure 4.4(c). La variable x_1 n'a plus de support unaire, on peut donc projeter un coût de un sur $c_\varnothing$. Le réseau de la figure 4.4(d) est CAD.*

Cette propriété, qui tend à accumuler les coûts vers les variables de plus petit indice, peut s'établir en $\mathcal{O}(ed^2)$, avec un espace de $\mathcal{O}(ed)$. Dans la pratique, il est possible d'utiliser CA et CAD simultanément, dans une propriété de cohérence locale appelée *cohérence d'arc complète directionnelle* (CACD) [Coo03, LS03]. Elle est appelée *full directional arc consistency* (*FDAC*) en anglais.

Définition 4.6 *Une variable est complètement et directionnellement arc-cohérente si elle est arc-cohérente et directionnellement arc-cohérente.*

Un réseau est complètement et directionnellement arc-cohérent si toutes ses variables le sont.

Propriété 4.5 *La procédure 19 (*CACD*) rend un problème CACD.*



(a) Problème originel.

(b) Extension de 1 de (x_2, b).

(c) Projection de 1 sur (x_1, a).

(d) Projection de 1 sur $c_\varnothing$.

FIG. 4.4 – Établissement de la cohérence d'arc directionnelle.

Procédure 15 – Extension($x_i \in \mathcal{X}, x_j \in V(x_i), v_j \in D_j, e \in E$)

$c_j(v_j) \leftarrow c_j(v_j) \ominus e$;
$\Delta(x_j, x_i, v_j) \leftarrow \Delta(x_j, x_i, v_j) \ominus e$;

Fonction 16 – ÉtablissementSupportComplet($x_i \in \mathcal{X}, x_j \in V(x_i)$) : booléen

$drapeau \leftarrow$ faux ;
pour chaque $v_i \in D_i$ **faire**
  $min \leftarrow \min_{v_j \in D_j} \{c_{ij}(v_i, v_j) \oplus c_j(v_j)\}$;
  si $(min > 0)$ **alors**
    pour chaque $v_j \in D_j$ **faire**
      si $(c_{ij}(v_i, v_j) \neq \top)$ **alors**
        Extension$(x_i, x_j, v_j, c_{ij}(v_i, v_j) \ominus min)$
    $drapeau \leftarrow drapeau \vee$ Projection(x_i, x_j, v_i) ;
retourner $drapeau$;

Procédure 17 – CAD

```
   drapeau ← faux ;
11 tant que (R ≠ ∅) faire
       x_j ← max_{x_k ∈ R} x_k ;
       R ← R \ {x_j} ;
       pour chaque x_i ∈ V⁻(x_j) faire
12         si (ÉtablissementSupportComplet(x_i, x_j)) alors
13             R ← R ∪ {x_i} ;
14             drapeau ← drapeau ∨ ProjectionUnaire(x_i) ;
15         si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
16 si (drapeau) alors
       pour chaque x_i ∈ 𝒳 faire
           si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
```

Procédure 18 – CA3

```
17 tant que (Q ≠ ∅) faire
       x_j ← Extraire(Q) ;
       drapeau ← faux ;
       pour chaque x_i ∈ V⁺(x_j) faire
18         si (ÉtablissementSupport(x_i, x_j)) alors
19             R ← R ∪ {x_i} ;
               drapeau ← drapeau ∨ ProjectionUnaire(x_i) ;
           si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
20     si (drapeau) alors
           pour chaque x_i ∈ 𝒳 faire
               si (SuppressionValeurs(x_i)) alors
                   Q ← Q ∪ {x_i} ;
```

Procédure 19 – CACD

```
Q ← 𝒳 ;
R ← 𝒳 ;
CN ;
tant que ((Q ≠ ∅) ∧ (R ≠ ∅)) faire
    CA3 ;
    CAD ;
```

Preuve 4.6 La cohérence d'arc complète et directionnelle utilise deux procédures principales : la procédure 18 (CA3), qui établit la cohérence d'arc et présente peu de différences avec la procédure 13 (CA2), et la procédure 17 (CAD) qui établit la cohérence d'arc directionnelle. Celle-ci utilise notamment la fonction 16 (ÉtablissementSupportComplet), qui établit le support complet d'une variable, et renvoie vrai si elle fait passer un coût unaire de zéro à une valeur strictement positive. Cette fonction à son tour utilise la procédure 15 (Extension), qui étend un coût unaire.

Prouvons d'abord que CAD rend le problème CAD. Pour cela, nous utiliserons un ensemble R contenant toutes les variables dont un plus petit voisin pourrait ne pas avoir de support complet par rapport à elles. Nous prouverons que l'assertion suivante est vraie à la ligne **11** durant l'exécution du programme :

$$x_j \notin R \Rightarrow \forall x_i \in V^-(x_j), \forall v_i \in D_i, \exists v_j \in D_j, c_{ij}(v_i, v_j) \oplus c_j(v_j) = 0$$

Initialement, R contient toutes les variables et donc l'invariant est vrai. Ensuite, la fonction ÉtablissementSupportComplet(x_i, x_j, v_i), appelé à la ligne **12**, établit effectivement un support complet aux valeurs v_i des plus petits voisins x_i des variables x_j dépilées de R.

Recherchons maintenant les actions pouvant mener à la perte du support complet de v_i de x_i, par rapport à c_{ij}, où $x_i \prec x_j$. v_i peut perdre un support complet si un coût binaire augmente, si le support est supprimé, ou si un coût unaire augmente. Un coût binaire ne peut augmenter que par extension, lors de l'établissement de CAD. Or, par définition, il ne peut y avoir d'extension de x_i sur c_{ij}, puisque $x_i \prec x_j$. Le support v_j ne peut être supprimé que parce que le coût unaire dépasse $\top - c_\varnothing$. Mais puisque v_j est support complet de v_i, alors $c_j(v_j) = 0$, et donc $c_\varnothing = \top$, ce qui n'est pas possible, car l'incohérence du réseau aurait été détectée avant. Il reste un dernier cas : le coût unaire de v_j a augmenté, ce qui arrive lorsqu'un coût est projeté sur cette valeur. C'est pour cela qu'à chaque fois qu'une projection fait augmenter le coût unaire de la valeur d'une variable de zéro à une valeur strictement positive, la variable est empilée dans R (ligne **18**). Si l'algorithme termine, il établit donc bien CAD.

Il reste maintenant à voir dans quelle mesure l'établissement de CAD détruit la propriété CA. CA peut être détruite lorsqu'un support est perdu. Ce support peut être perdu lors de la suppression de cette valeur. Dans ce cas, la variable est empilée dans Q. Le support peut être perdu car CAD a étendu un coût binaire, par exemple de x_j vers c_{ij}. Dans ce cas, si v_i de x_i est le support de v_j de x_j, on a :

- avant l'extension de ÉtablissementSupportComplet, $c_{ij}(v_i, v_j) = 0$,
- après l'extension, $c_{ij}(v_i, v_j) = min$,
- après la projection, $c_{ij}(v_i, v_j) = 0$, et donc le support n'est pas perdu.

La première propriété de CN, forçant l'existance d'un coût unaire, peut être brisée par CAD, après une projection. Si c'est le cas, alors un coût unaire a dû passer de zéro à une valeur strictement positive, et alors ÉtablissementSupportComplet (ligne **12**) a renvoyé vrai. Si c'est le cas, alors on appelle ÉtablissementSupportUnaire à la ligne **14**.

La seconde propriété de CN peut être détruite lorsque le coût unaire d'une valeur augmente, ce qui est le cas après une projection, et donc après ÉtablissementSupportComplet. C'est pourquoi on appelle SuppressionValeurs (ligne **15**) pour enlever les valeurs

non valides. La seconde propriété de CN peut aussi être brisée lors de l'augmentation de $c_\varnothing$, dans la fonction ProjectionUnaire. Si c'est le cas, alors le booléen *drapeau* prend la valeur vrai, et le code sous la ligne **16** passe alors toutes les variables en revue pour éliminer les valeurs non valides.

Étant donné que CA3 est très similaire à CA2 (une seule ligne, écrite en noir dans l'algorithme diffère entre eux), les interactions entre CA et CN, et notamment les disparitions de supports, sont traités de la même manière que dans l'algorithme précédent. Donc, si l'algorithme termine, CACD est bien établi.

Une dernière remarque est qu'il est maintenant inutile d'établir les supports de CA dans les voisinages supérieurs. Cette tâche est en effet effectuée par CAD, qui établit des supports complets et donc *a fortiori* des supports simples. □

Propriété 4.6 CACD *s'applique en temps* $\mathcal{O}(end^3)$ *en utilisant un espace en mémoire de* $\mathcal{O}(ed)$.

Preuve 4.7 On peut déjà noter que la complexité totale de CA3 est la même que celle de celle de CA2. En effet, les deux algorithmes ne diffèrent que d'une seule ligne, et le nombre de fois qu'une variable est empilée dans Q reste borné par d, ce qui signifie que le code sous la boucle **17** garde sa complexité de $\mathcal{O}(ed^3)$. De plus, $c_\varnothing$ ne pouvant augmenter que $\top$ fois au maximum, la complexité sous la condition **20** reste aussi de $\mathcal{O}(\min\{nd, \top\} \times nd)$, ce qui nous donne une complexité cumulée inchangée à $\mathcal{O}(ed^3 + \min\{nd, \top\} \times nd)$.

Recherchons maintenant la complexité temporelle de CAD. La fonction ÉtablissementSupportComplet s'établit en temps $\mathcal{O}(d^2)$. On peut ensuite remarquer que, lorsqu'une variable x_j est dépilée de R, on n'empile à la ligne **13** que des variables x_i plus grandes que x_j. De plus, la variable qui est dépilée de R est systématiquement la plus petite de l'ensemble. En conséquence, dans une exécution de CAD, chaque variable est au plus dépilée une seule fois de R. En remarquant que le code sous la condition de la ligne **16** a une complexité de $\mathcal{O}(nd)$, on conclut que la complexité d'une exécution de CAD est donc de $\mathcal{O}(ed^2)$.

Cette procédure peut être itérée à chaque fois que l'on applique CA3, puisque la ligne **19** ajoute un élément à R. Cette ligne est appelée au maximum $\mathcal{O}(nd)$ fois, donc la complexité cumulée de CAD est $\mathcal{O}(nd \times ed^2)$. En revanche, puisque CAD ajoute un élément dans Q au maximum $\mathcal{O}(\min\{\top, nd\})$ fois, la complexité totale de CA3 ne change pas. CACD est donc en $\mathcal{O}(end^3 + \min\{nd, \top\} \times nd) = \mathcal{O}(end^3)$.

CAD n'utilise pas de structure de données supplémentaire, la complexité spatiale de CACD reste donc en $\mathcal{O}(ed)$. □

Si, dans un réseau de contraintes pondérées, $\top$ vaut un, ou tous les coûts sont nuls ou valent $\top$, établir CAD revient à établir la cohérence d'arc directionnelle dans le réseau de contraintes classiques. Établir CACD revient à établir la cohérence d'arc.

4.1.2 Cohérence d'arc existentielle (CAE)

4.1.2.1 Algorithmes

Dans le cadre de la thèse, nous avons conçu une nouvelle propriété de cohérence locale plus forte que les précédentes, appelée *cohérence d'arc existentielle* (CAE) [HLdGZ05], tout d'abord définie dans les réseaux de contraintes binaires. Cette propriété est aussi connue sous le nom *existential arc consistency* ou *EAC*.

L'intuition derrière cette cohérence locale est la suivante. Si, pour une valeur v_i d'une variable x_i donnée, il existe une fonction de coût c_{ij} telle que v_i n'a pas de support complet par rapport à c_{ij}, alors il est possible d'augmenter le coût unaire de v_i par une série d'extensions et de projections. Donc, si, pour toute valeur v_i de x_i de coût unaire nul, il existe une fonction de coût c_{ij} telle que v_i n'a pas de support complet par rapport à c_{ij}, alors on peut faire passer tous les coûts unaires de x_i au dessus de zéro, et donc faire augmenter $c_\varnothing$. Le but de cette cohérence locale est de trouver dans une variable x_i une valeur v_i de coût unaire nul qui possède un support complet par rapport à toutes les fonctions de coût binaires pesant sur x_i. S'il n'existe pas de telle valeur, alors on peut faire augmenter $c_\varnothing$.

La définition suivante décrit plus formellement la propriété de cohérence locale.

Définition 4.7 *Une variable x_i est existentiellement arc-cohérente si :*
- $\exists v_i \in D_i, c_i(v_i) = 0 \wedge \forall x_j \in V(x_i), \exists v_j \in D_j, c_{ij}(v_i, v_j) \oplus c_j(v_j) = 0$ *(v_i est alors appelé* support existentiel *de x_i) et*
- *x_i est nœud-cohérente.*

Un réseau est existentiellement arc-cohérent si toutes ses variables le sont.

L'exemple d'exécution dans le paragraphe suivant montre que si une variable n'est pas CAE, alors on peut faire augmenter le $c_\varnothing$.

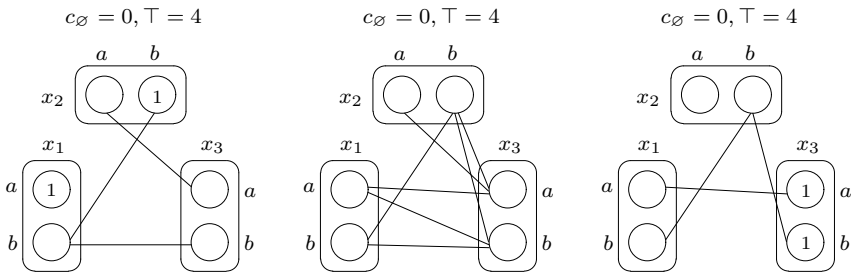
Exemple 4.4 *On peut voir sur le problème de la figure 4.5(a) que la variable x_3 ne respecte pas la première propriété de la définition de CAE. On étend donc les coûts unaire des variables x_1 et x_2, comme décrit dans la figure 4.5(b), puis on projette les coûts binaires sur x_3 (voir sur la figure 4.5(c)). On peut enfin faire une projection unaire pour faire augmenter le $c_\varnothing$ (cf.* FIG. *4.5(d)).*

En pratique, la CAE est établie conjointement avec la CACD.

Définition 4.8 *Un réseau est existentiellement et directionnellement arc-cohérent (CACE) s'il est existentiellement arc-cohérent et directionnellement arc-cohérent.*

Cette propriété s'appelle *existential and directional arc consistency*, ou *EDAC* en anglais.

Pour appliquer la CACE, on utilise en pratique trois files de propagation principales, Q, R et P. Comme dans CACD, Q stocke les variables dont un plus grand voisin pourrait ne pas avoir de support par rapport à elles. R stocke les variables dont un plus petit voisin pourrait ne pas avoir de support complet par rapport à elles. P stocke les



(a) Problème originel.

(b) Extension des coûts unaires de x_1 et x_2.

(c) Projection sur x_3.

(d) Projection unaire.

FIG. 4.5 – Établissement de la cohérence d'arc existentielle.

variables x_i dont le support existentiel a peut-être disparu. La file auxiliaire S sert à remplir efficacement P.

Propriété 4.7 *L'algorithme* CACE *est correct.*

Procédure 20 – ÉtablissementSupportExistentiel(x_i)

```
21 e ← min_{v_i∈D_i}{c_i(v_i) ⊕_{x_j∈V⁻(x_i)} min_{v_j∈D_j}{c_ij(v_i, v_j) ⊕ c_j(v_j)}};
   si (e > 0) alors
       pour chaque x_j ∈ V⁻(x_i) faire
22         ÉtablissementSupportComplet(x_i, x_j) ;
```

Procédure 21 – CA4

```
   tant que (Q ≠ ∅) faire
       x_j ← Extraire(Q) ;
       drapeau ← faux ;
       pour chaque x_i ∈ V(x_j) faire
           si (ÉtablissementSupport(x_i, x_j)) alors
               R ← R ∪ {x_i} ;
23             S ← S ∪ {x_i} ;
               drapeau ← drapeau ∨ ProjectionUnaire(x_i) ;
           si (SuppressionValeurs(x_i)) alors
               Q ← Q ∪ {x_i} ;
       si (drapeau) alors
           pour chaque x_i ∈ 𝒳 faire
               si (SuppressionValeurs(x_i)) alors
                   Q ← Q ∪ {x_i} ;
```

Preuve 4.8 La procédure principale 24 (CACE) utilise trois grandes procédures : les procédures 21 (CA4) et 22 (CAD2) d'une part, qui établissent respectivement CA et CAD, et présentent peu de différences avec les procédures 18 (CA3) et 17 (CAD), et la procédure 23 (CAE) qui établit la cohérence d'arc existentielle uniquement dans un sens. En effet, CAD établit un support complet à toutes les variables par rapport aux plus grands voisins. Il n'est donc pas besoin, lorsque l'on utilise CACE, de vérifier les supports existentiels dans les plus grands voisins. CAE utilise la procédure 20, qui établit un support existentiel par rapport aux plus petits voisins.

Vérifions tout d'abord que ÉtablissementSupportExistentiel est correct. Si l'on détecte que x_i n'a pas de support existentiel (ligne **21**), le plus simple est de projeter la somme des coûts binaires et unaires de tous les voisins de x_i sur cette variable (ligne **22**). De

Procédure 22 – CAD2

```
drapeau ← faux ;
tant que (R ≠ ∅) faire
    x_j ← max_{x_k ∈ R} x_k ;
    R ← R \ {x_j} ;
    pour chaque x_i ∈ V⁻(x_j) faire
        si (ÉtablissementSupportComplet(x_i, x_j)) alors
            R ← R ∪ {x_i} ;
24          S ← S ∪ {x_i} ;
            drapeau ← drapeau ∨ ProjectionUnaire(x_i) ;
        si (SuppressionValeurs(x_i)) alors
            Q ← Q ∪ {x_i} ;
si (drapeau) alors
    pour chaque x_i ∈ 𝒳 faire
        si (SuppressionValeurs(x_i)) alors
            Q ← Q ∪ {x_i} ;
```

Procédure 23 – CAE

```
drapeau ← faux ;
tant que (P ≠ ∅) faire
    x_i ← min_{x_k ∈ P} x_k ;
    P ← P \ {x_i} ;
    si (ÉtablissementSupportExistentiel(x_i)) alors
25      R ← R ∪ {x_i} ;
        pour chaque x_j ∈ V⁺(x_i) faire
26          P ← P ∪ {x_j} ;
    drapeau ← drapeau ∨ ProjectionUnaire(x_i) ;
    si (SuppressionValeurs(x_i)) alors
        Q ← Q ∪ {x_i} ;
27 si (drapeau) alors
    pour chaque x_i ∈ 𝒳 faire
        si (SuppressionValeurs(x_i)) alors
            Q ← Q ∪ {x_i} ;
```

Procédure 24 – CACE

28 **tant que** $((Q \neq \varnothing) \vee (R \neq \varnothing) \vee (S \neq \varnothing))$ **faire**
29 $\quad P \leftarrow \left(\bigcup_{x_i \in S} V^+(x_i)\right) \cup S$;
$\quad S \leftarrow \varnothing$;
$\quad$ CAE ;
$\quad$ CAD2 ;
$\quad$ CA4 ;

cette manière, on est assuré que chaque valeur a un support complet par rapport à tous les voisins, ce qui entraîne que la variable a un support existentiel. Il est à noter que cette projection complète ne se fait que des plus petites variables sur les plus grandes ; CAD se charge de l'autre sens.

Regardons ensuite quelles propriétés l'établissement de CAE peut avoir brisées. CAE peut faire augmenter les coûts unaires de x_i, et donc tous les plus grands voisins de x_i peuvent avoir perdu leur support existentiel. On empile donc toutes ces variables dans P à la ligne **26**. Puisque les coûts unaires de x_i ont pu augmenter, il faudra donc vérifier les supports complets des plus grands voisins de x_i par rapport à x_i. Ce sera fait en empilant x_i dans R à la ligne **25**.

CAE peut à son tour être brisée par l'établissement d'autres propriétés de cohérence locale. Toute projection qui fait augmenter le coût unaire d'une variable x_i peut par exemple rendre les plus grands voisins existentiellement incohérents. Il faut donc veiller, dans CA4 et CAD2, à empiler des variables dans S après chaque projection (lignes **23** et **24**). Tous les plus grands voisins des variables empilées dans S sont ensuite empilés dans P (ligne **29**) pour un ré-examen de leur cohérence existentielle.

L'algorithme est donc correct. □

Propriété 4.8 *La complexité de* CACE *est de* $\mathcal{O}(ed^2 \max\{nd, \top\})$ *en temps et* $\mathcal{O}(ed)$ *en espace.*

Preuve 4.9 Tentons tout d'abord de trouver la complexité temporelle d'une exécution de CAE. Cette procédure itère sur le nombre d'éléments de P. À l'inverse de CAD, CAE empile dans P les variables par indice croissant, et donc, au sein d'une itération de CAE, chaque variable n'est empilée qu'une fois. Donc ÉtablissementSupportExistentiel n'est appelée qu'une fois par variable, et donc la complexité cumulée de cette fonction sur une exécution de CAE est de $\mathcal{O}(ed^2)$. La ligne **26** a une complexité cumulée de $\mathcal{O}(e)$ et le code sous la condition de la ligne **27**, une complexité de $\mathcal{O}(nd)$. La complexité d'une exécution de CAE est donc de $\mathcal{O}(nd + ed^2) = \mathcal{O}(ed^2)$.

En s'aidant des propriétés précédentes, on note aussi qu'une exécution de CAD2 se fait en temps $\mathcal{O}(ed^2)$ et que la complexité cumulée de CA4 se fait toujours en $\mathcal{O}(ed^3 + \min\{nd, \top\}nd)$.

Trouvons maintenant la complexité totale de l'algorithme. La ligne **28** itère tant que Q, R ou S n'est pas vide. Q est rempli à la suppression d'une valeur. Cela ne peut se faire qu'au maximum $\mathcal{O}(nd)$ fois. Si R n'est pas vide, cela signifie que CA y

a inséré un élément, soit au maximum $\mathcal{O}(nd)$ fois. Si S n'est pas vide, cela signifie que CAD2 y a inséré un élément. Cette fonction a été appelée parce que CA ou CAE avait inséré un élément dans R. Le premier cas ne peut pas se produire plus de $\mathcal{O}(nd)$ fois, le second, plus de $\mathcal{O}(\top)$ fois car à chaque fois que CAE insère un élément dans R, il faut aussi augmenter $c_\varnothing$. La complexité de l'algorithme dans le pire cas est donc $\mathcal{O}(ed^2 \max\{nd, \top\} + ed^3 + \min\{nd, \top\}nd) = \mathcal{O}(ed^2 \max\{nd, \top\})$.

En ce qui concerne la complexité spatiale, aucune structure de donnée n'a été ajoutée. Elle reste donc à $\mathcal{O}(ed)$. □

4.1.2.2 Expérimentations

Cette cohérence locale a été comparée avec CACD sur divers problèmes. Dans la pratique, le solveur a *maintenu* la propriété CACD et CACE durant l'exécution du problème, c'est-à-dire que la propriété est établie à chaque nœud de l'arbre de recherche. La cohérence locale CACE n'étant alors définie que pour les fonctions de coût binaires, le solveur n'effectue aucune propagation sur les fonctions de coût d'arité supérieure. Il attend simplement que toutes les variables, sauf deux, soient affectées.

Les heuristiques utilisées affectent en priorité la variable possédant le plus petit ratio « taille du domaine » sur « nombre de fonctions de coût pesant sur elle ». La valeur sélectionnée pour affectation est choisie parmi celles qui ont le plus petit coût unaire. Toutes les variables étant numérotées $x_1, \ldots, x_n$, l'ordre utilisé par la cohérence d'arc directionnelle est l'ordre naturel de leur indice. La valeur de $\top$ est fixée à l'optimum.

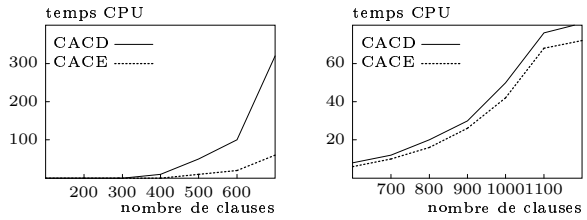
Les instances Max-SAT ont constitué le premier jeu de tests. Il s'agit de clauses en formes normales, où chaque variable n'apparaît au maximum qu'une fois par clause. Ces clauses ont été générées aléatoirement par *Cnfgen*[2]. Ce logiciel génère des clauses éventuellement identiques, ce qui rend les problèmes largement plus complexes que lorsque les clauses répétées ne sont pas autorisées. Le but du problème est alors de trouver une affectation de toutes les variables telle qu'un nombre minimal de clauses soient violées. Nous avons généré des instances de clauses contenant deux littéraux (Max-2-SAT, cf. figure 4.6(a)) avec 80 variables, et des instances de clauses contenant trois littéraux (Max-3-SAT, cf. figure 4.6(b)) avec 40 variables. Chaque point de la figure rapporte le temps CPU passé en moyenne sur 10 instances à résoudre un problème dont le nombre de clauses varie sur l'axe des abscisses.

Pour les instances de Max-2-SAT, CACE résout en moyenne les instances $7,4$ fois plus rapidement que CACD. La différence de rapidité est toujours à l'avantage de CACE dans les instances Max-3-SAT, même si celle-ci est largement moindre. Ceci est dû au fait que CACE n'est établie que lorsque les clauses sont devenues binaires ; l'algorithme attend donc l'affectation d'une variable dans une clause avant de s'exécuter.

Un deuxième jeu de tests a été constitué d'instances de Max-CSP aléatoires, c'est-à-dire de CSP générés aléatoirement, où il faut minimiser le nombre de contraintes violées. Les instances générées sont d'une part divisées en :

– « peu dense » si $e = 2{,}5 \times n$,

[2]Le logiciel est téléchargeable sur ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances.



(a) Test sur des instances de Max-2-SAT aléatoires.

(b) Test sur des instances de Max-3-SAT aléatoires.

FIG. 4.6 – Évaluation de CACE sur plusieurs problèmes Max-SAT.

– « denses » si $e = \frac{n(n-1)}{8}$ et
– « complètes » si $e = \frac{n(n-1)}{2}$,

et d'autre part :

– « peu contraintes » si $t = t^0$,
– « contraintes » si $t = d^2 - \frac{t^0}{4}$,

où t représente la proportion de tuples interdits par fonction de coût, et t^0 est la proportion minimale de tuples interdits par fonction de coût pour que le réseau soit incohérent. Nous avons donc six classes d'instances et chaque point des figures 4.7 donne le temps CPU moyen passé à résoudre cinquante de ces instances, le nombre de variable augmentant. L'ordinateur utilisé pour ce jeu de tests, ainsi que pour le précédent, était un Pentium 4 à 2,8 GHz.

On peut noter qu'ici encore CACE est plus rapide. Il est seulement 2,2 fois plus rapide en moyenne pour les instances « complet / peu contraint », mais 12,8 fois plus rapide pour les instances « peu dense / contraint ».

Un dernier jeu de test a été le problème du « placement d'entrepôts » (*uncapacitated warehouse location problem*), dans lequel une entreprise propose d'ouvrir des entrepôts sur au maximum l sites donnés pour fournir s magasins existants. L'objectif est alors de déterminer quels entrepôts ouvrir, et à quel entrepôt assigner un magasin pour minimiser le coût de la maintenance et les coûts d'approvisionnement, sachant que chaque magasin est approvisionné par un unique entrepôt. Le problème est modélisé par l variables booléennes correspondant à l'ouverture ou non d'un entrepôt sur un site, s variables entières (de domaines l) modélisant l'emplacement d'un entrepôt. $l+s$ fonctions de coût unaires modélisent le coût d'ouverture d'un site et $l \times s$ autres fonctions modélisent le coût d'approvisionnement d'un magasin par un entrepôt.

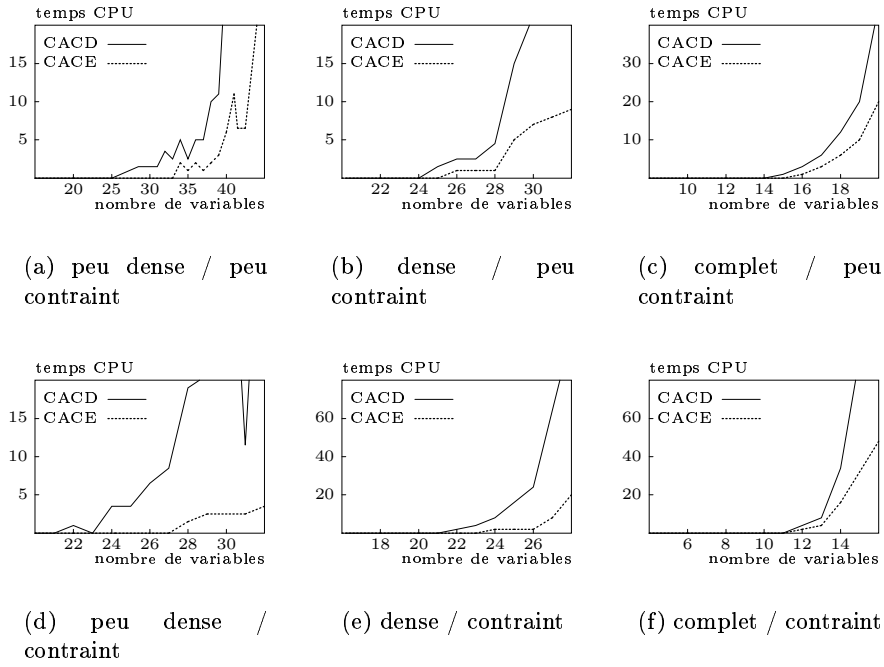
Les instances résolues sont celles des problèmes cap71-134[3] des bancs de tests de la librairie standard OR, ainsi que les instances MO*-MP* proposées par [KTFL01].

Les résultats des tests, consignés dans le tableau 4.1, ont été obtenus sur un Pen-

[3] J. E. Beasley http://www.dcc.unicamp.br/~aprox/facility/instancias/uncapacitated/uncapinfo.html.gz.

Problème	Taille	CACD	CACE	CPLEX
cap71	16 × 50	0,01	0,00	0,02
cap72	16 × 50	0,01	0,01	0,00
cap73	16 × 50	0,02	0,01	0,00
cap74	16 × 50	0,02	0,01	0,02
cap101	25 × 50	0,03	0,02	0,02
cap102	25 × 50	0,13	0,03	0,01
cap103	25 × 50	0,35	0,03	0,01
cap104	25 × 50	0,24	0,04	0,01
cap131	50 × 50	65,02	0,11	0,05
cap132	50 × 50	155,08	0,12	0,04
cap133	50 × 50	194,36	0,14	0,05
cap134	50 × 50	44,75	0,12	0,05
MO1	100 × 100	–	216,57	202,48
MO2	100 × 100	11775,60	44,45	36,67
MO3	100 × 100	15123,70	74,60	199,84
MO4	100 × 100	3525,61	31,74	43,26
MO5	100 × 100	2354,95	33,98	42,96
MP1	200 × 200	–	3403,37	2296,65
MP2	200 × 200	–	1443,93	917,92
MP3	200 × 200	–	981,28	1209,15
MP4	200 × 200	–	1768,51	1936,57
MP5	200 × 200	–	1099,88	697,51

TAB. 4.1 – Temps en secondes pour trouver l'instanciation optimale pour le problème du placement d'entrepôts. La taille du problème est $l \times s$, où l est le nombre de sites et s le nombre de magasins. Le signe « – » signifie que l'instance n'a pas été résolue en moins de 5 heures.



(a) peu dense / peu contraint

(b) dense / peu contraint

(c) complet / peu contraint

(d) peu dense / contraint

(e) dense / contraint

(f) complet / contraint

FIG. 4.7 – Évaluation de CACE sur diverses problèmes Max-CSP.

tium 4 à 2 GHz, avec 512 Mo de mémoire vive. Ils montrent que CACE est de plusieurs ordres meilleur que CACD pour des problèmes de taille supérieure ou égale à $l \times 50$. La borne inférieure calculée par CACE est donc meilleure et dépend probablement moins de l'ordre des variables utilisé par la cohérence d'arc directionnelle. Bien que le problème du placement d'entrepôts soit rapidement résolu par des modélisations dédiées ([Erl78, Kör89]) ou par des heuristiques *ad hoc* ([KTFL01, MH04]), on peut noter que notre algorithme générique est capable de résoudre le problème pour des instances de taille raisonnable dans un temps relativement court (ici moins d'une heure). À titre de comparaison, nous avons utilisé le solveur Ilog CPLEX 9.0 avec les paramètres par défaut et sans borne supérieure initiale (alors que dans les instances résolues par CACE, la valeur de $\top$ est le coût optimum), sur une formulation directe du problème. CACE et CPLEX résolvent les instances en des temps comparables.

4.1.3 Cohérence d'arc aux bornes (CAB)

La modélisation en RCP du problème de la localisation de motifs, que nous avons détaillée dans le précédent chapitre, utilise des domaines très grands, de l'ordre de plusieurs centaines de millions de valeurs possibles. Les complexités temporelle et surtout spatiale font que les propriétés de cohérence locale précédemment évoquées ne peuvent pas être appliquées sur ce type d'instance. Nous avons donc conçu, dans le cadre de nos recherches, une nouvelle propriété de cohérence locale pour les réseaux de

contraintes pondérées, appelée *cohérence d'arc aux bornes*. Cette propriété est inspirée de la 2B-cohérence [Lho93] (voir la définition 3.4) des réseaux de contraintes classiques, qui s'appuie sur une représentation des domaines par des intervalles.

4.1.3.1 Algorithmes

Définition 4.9 *Nous supposerons maintenant que les domaines peuvent se représenter par des intervalles : $\mathcal{D} = \mathcal{I} = \{I_1, \ldots, I_n\}$. Chaque domaine est ordonné par un ordre total arbitraire $\preceq$. Les bornes inférieure et supérieure d'un intervalle I_i seront respectivement notées bi_i et bs_i, et pour tout intervalle I_i, nous noterons $\mathring{I}_i$ l'intérieur de I_i, c'est-à-dire l'ensemble potentiellement vide $I_i \setminus \{bi_i, bs_i\}$.*

Nous utiliserons aussi les fonctions successeur (succ) et prédécesseur (pred) naturelles pour l'ordre $\preceq$, et pour cela nous introduisons deux nouvelles valeurs, $-\infty$ et $+\infty$, telles que :

- $prec(-\infty) = succ(-\infty) = -\infty$,
- $prec(+\infty) = succ(+\infty) = +\infty$,
- $-\infty \prec +\infty$,
- $\forall x_i \in \mathcal{X}, prec(bi_i) = -\infty$,
- $\forall x_i \in \mathcal{X}, succ(bs_i) = +\infty$.

Le fait d'utiliser la représentation par intervalles change radicalement l'expressivité de notre formalisme ; il n'est *a priori* pas possible de supprimer une valeur se trouvant à l'intérieur d'un intervalle. En revanche, cette représentation nous permet de décrire un domaine D_i uniquement par deux valeurs : bi_i et bs_i.

Puisqu'il est maintenant impossible de supprimer une valeur à l'intérieur d'un domaine, la cohérence de nœud n'est ici plus utilisable. On définit alors la *cohérence de nœud aux bornes*, l'équivalent de la cohérence de nœud adapté à ce type de réseaux de contraintes pondérés. Cette propriété ne permet que les suppressions aux bornes des domaines.

Définition 4.10 *Une variable x_i est nœud-cohérent aux bornes (CNB) si :*

- $(c_\varnothing \oplus c_i(bi_i) < \top) \wedge (c_\varnothing \oplus c_i(bs_i) < \top)$ *et*
- $\exists v_i \in I_i, c_i(v_i) = 0$.

Un RCP est nœud-cohérent aux bornes si toutes ses variables le sont.

Les algorithmes ProjectionUnaireBornes, SuppressionBorneInférieure, SuppressionBorneSupérieure (non décrit ici, mais pouvant être déduit de la fonction précédente par symétrie) et CNB nous permettent d'établir cette propriété.

Afin de réduire la complexité spatiale de CN, nous ne devons pas modifier les coûts unaires. En effet, la modification des coûts unaires implique l'utilisation d'une structure de données de taille proportionnelle en la taille des domaines, ce que nous ne voulons pas. Nous utilisons donc de nouvelles structures de données $\Delta(x_i)$, représentant le coût qui a été projeté de x_i sur $c_\varnothing$. $c_i(v_i)$ est donc un raccourci pour $\mathbf{c_i}(v_i) \ominus \Delta(x_i)$.

Propriété 4.9 *L'algorithme* CNB *est correct.*

Fonction 25 – ProjectionUnaireBornes($x_i \in \mathcal{X}$) : booléen

$min \leftarrow \min_{v_i \in D_i}\{c_i(v_i)\}$;
$\Delta(x_i) \leftarrow \Delta(x_i) \oplus min$;
$c_\varnothing \leftarrow c_\varnothing \oplus min$;
retourner $(min > 0)$;

Fonction 26 – SuppressionBorneInférieure($x_i \in \mathcal{X}$) : booléen

$drapeau \leftarrow$ faux ;
tant que $(c_\varnothing \oplus c_i(bi_i) = \top)$ **faire**
  $bi_i \leftarrow succ(bi_i)$;
  si $(bi_i \succ bs_i)$ **alors**
    lever incohérence ;
  $drapeau \leftarrow$ vrai ;
retourner $drapeau$;

Procédure 27 – CNB

pour chaque $x_i \in \mathcal{X}$ **faire**
  ProjectionUnaireBornes(x_i) ;
pour chaque $x_i \in \mathcal{X}$ **faire**
  SuppressionBorneInférieure(x_i) ;
  SuppressionBorneSupérieure(x_i) ;

Preuve 4.10 L'algorithme utilise principalement les procédures 25 (ProjectionUnaireBornes) et 26 (SuppressionBorneInférieure), dont les valeurs de retour ne seront utiles que par la suite. Les appels successifs de ProjectionUnaireBornes permettent de trouver le support unaire de chaque variable. Les appels de SuppressionBorneInférieure et SuppressionBorneSupérieure permettent d'établir la première propriété de CNB sur une variable. Ces fonctions ne détruisent pas les supports unaires, puisque si une valeur v_i de x_i est enlevée, cela signifie que $c_\varnothing \oplus c_i(v_i) = \top$. Or, si v_i est un support unaire, cela implique que $c_i(v_i) = 0$ et donc que $c_\varnothing = \top$. Si c'est le cas, toutes les valeurs sont effacées par SuppressionBorneInférieure, et donc l'instance est CNB. L'algorithme CNB est donc correct.

Propriété 4.10 *La CNB peut s'établir en temps $\mathcal{O}(nd)$ et en espace $\mathcal{O}(n)$.*

Preuve 4.11 La complexité temporelle de ProjectionUnaireBornes est de $\mathcal{O}(d)$, et la complexité cumulée de SuppressionBorneSupérieure est de $\mathcal{O}(nd)$. Il est donc clair que CNB peut s'établir en temps $\mathcal{O}(nd)$.

La complexité spatiale est la place occupée pour stocker les bornes inférieure et supérieure de chaque domaine, ainsi que les Δ : $\mathcal{O}(n)$. □

À l'instar de la cohérence de nœud, la cohérence de nœud aux bornes peut être avantageusement complétée par une autre propriété de cohérence locale : *la cohérence d'arc aux bornes*. Celle-ci permet d'établir des supports aux valeurs des variables, par rapport aux variables voisines. Pour maintenir une complexité spatiale basse — en l'occurrence, pour avoir une complexité spatiale indépendante de la taille des domaines —, nous n'autoriserons que les projections sur les bornes des domaines.

Définition 4.11 *Une variable x est arc-cohérente aux bornes (CAB) si :*

- *elle est nœud-cohérente aux bornes, et*
- $\forall c \in \mathcal{C}, x \in var(c) \Rightarrow (\exists (a, a') \in \ell^2(var(c) \setminus \{x\}), c(a \cup \{(x \leftarrow bi)\}) = c(a' \cup \{(x \leftarrow bs)\}) = 0)$.

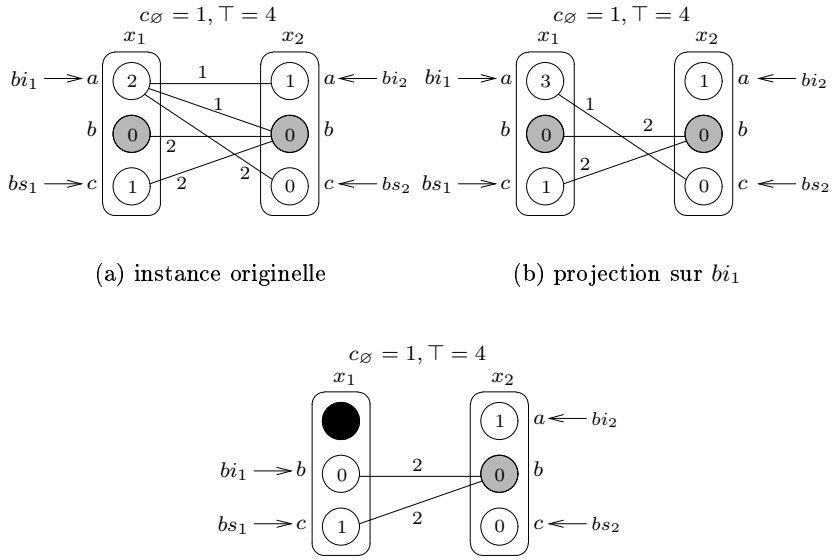
Un RCP est arc-cohérent aux bornes si toutes ses variables le sont.

Dans le cas où les fonctions de coût sont binaires, alors la seconde propriété de la définition peut se réécrire en :

- $\forall x_j \in V(x_i), \exists (v_j, v'_j) \in I^2(x_j), c_{ij}(bi_i, v_j) = c_{ij}(bs_i, v'_j) = 0$.

L'exemple suivant décrit l'établissement de CAB.

Exemple 4.5 *La figure 4.8(a) représente un RCP où les valeurs sont classées par ordre alphabétique ($a \prec b \prec c$), et donc par exemple $bi_1 = a$ et $bs_1 = c$. Les valeurs des intervalles intérieurs sont grisés. $c_\varnothing$ est initialement à un. CAB détecte tout d'abord que la valeur bi_1 de x_1 n'a pas de support. On projette donc un coût de un sur cette valeur (cf. figure 4.8(b)). Le coût unaire de bi_1 passe à $\top$, on supprime donc la valeur. La borne inférieur de l'intervalle de x_1 est ensuite mise à jour (cf. figure 4.8(c)). Cette instance est CAB, mais pas CA, car la valeur b de x_2 n'a pas de support. Ceci montre que CAB peut être plus faible que CA.*



(a) instance originelle

(b) projection sur bi_1

(c) mise à jour de bi_1

FIG. 4.8 – Étapes de l'établissement de CAB.

Les algorithmes ProjectionBorneInférieure, ProjectionBorneSupérieure (non décrit ici car pouvant être obtenu à partir de la fonction précédente par symétrie), SuppressionBorneInférieure, SuppressionBorneSupérieure (non décrit) et CAB permettent d'établir CAB.

Ces différentes procédures utilisent les structures de données $\Delta(x_i, x_j, v_i)$ dont le fonctionnement est sensiblement le même que les structures utilisées dans la cohérence d'arc. Toutefois, une différence majeure est qu'ici v_i ne peut valoir que bi_i ou bs_i, ce qui limite la complexité spatiale des algorithmes. De plus, nous utiliserons une nouvelle structure de données, $\Delta(x_i, v_i)$, où v_i ne peut être encore une fois que bi_i ou bs_i, qui représente la somme des coûts binaires projetés sur bi_i. En d'autres termes, $\Delta(x_i, v_i) = \bigoplus_{x_j \in V(x_i)} \Delta(x_i, x_j, v_i)$. Pour résumer, $\forall x_i \in \mathcal{X}, \forall x_j \in V(x_i)$:

$$c_i(v_i) = \left(\mathbf{c_i}(v_i) \oplus \begin{cases} 0 & \text{si } v_i \in \mathring{I}(x_i), \\ \Delta(x_i, v_i) & \text{sinon.} \end{cases} \right) \ominus \Delta(x_i)$$

$$c_{ij}(v_i, v_j) = \mathbf{c_{ij}}(v_i, v_j) \ominus \left(\begin{cases} 0 & \text{si } v_i \in \mathring{I}(x_i), \\ \Delta(x_i, x_j, v_i) & \text{sinon.} \end{cases} \oplus \begin{cases} 0 & \text{si } v_j \in \mathring{I}(x_j), \\ \Delta(x_j, x_j, v_j) & \text{sinon.} \end{cases} \right)$$

Propriété 4.11 *L'algorithme* CAB *est correct.*

Preuve 4.12 Nous devons tout d'abord prouver que les soustractions sont possibles, c'est-à-dire que tous les membres de gauche de l'opérateur $\ominus$ sont inférieurs ou égaux

Fonction 28 – ProjectionBorneInférieure($x_i \in \mathcal{X}, x_j \in V(x_i)$) : booléen

$min \leftarrow \min_{v_j \in I_j}\{c_{ij}(bi_i, v_j)\}$;
si $(min = 0)$ **alors**
  retourner faux ;
$\Delta(x_i, x_j, bi_i) \leftarrow \Delta(x_i, x_j, bi_i) \oplus min$;
$\Delta(x_i, bi_i) \leftarrow \Delta(x_i, bi_i) \oplus min$;
retourner vrai ;

Fonction 29 – SuppressionBorneInférieure2($x_i \in \mathcal{X}$) : booléen

drapeau $\leftarrow$ vrai ;
30 **tant que** $(c_i(bi_i) \oplus c_\varnothing = \top)$ **faire**
  $bi_i \leftarrow succ(bi_i)$;
  si $(bi_i \succ bs_i)$ **alors**
    lever incohérence ;
31 $\Delta(x_i, bi_i) \leftarrow 0$;
32 **pour chaque** $x_j \in V(x_i)$ **faire**
33 $\Delta(x_i, x_j, bi_i) \leftarrow 0$;
  drapeau $\leftarrow$ vrai ;
retourner *drapeau* ;

aux membres de droite. Il faut donc pour cela prouver que les assertions suivantes sont vraies :

- $\forall x_i \in \mathcal{X}, \forall x_j \in V(x_i), \Delta(x_i, x_j, bi_i) \leq \mathbf{c_{ij}}(bi_i, bi_j) \wedge \Delta(x_i, x_j, bi_i) \leq \mathbf{c_{ij}}(bi_i, bs_j)$,
- $\forall x_i \in \mathcal{X}, \forall x_j \in V(x_i), \Delta(x_i, x_j, bi_i) \leq \min \left\{ \begin{array}{l} \mathbf{c_{ij}}(bi_i, bi_j) \ominus \Delta(x_j, x_i, bi_j), \\ \min_{v_j \in \mathring{I}_j}\{\mathbf{c_{ij}}(bi_i, v_j)\}, \\ \mathbf{c_{ij}}(bi_i, bs_j) \ominus \Delta(x_j, x_i, bs_j) \end{array} \right.$
- les assertions sur les bornes supérieures sont aussi vraies,
- $\forall x_i \in \mathcal{X}, \Delta(x_i) \leq \min\{\mathbf{c_i}(bi_i) \oplus \Delta(x_i, bi_i), \min_{v_i \in \mathring{I}_i}\{\mathbf{c_i}(v_j)\}, \mathbf{c_i}(bs_i) \oplus \Delta(x_i, bs_i)\}$.

Étant donné que les Δ sont nuls au début de l'algorithme, les assertions précédentes sont vraies à ce moment. Les transformations de ces structures de données par projection, projection unaire et effacement de valeurs conservent ces assertions.

On peut aussi remarquer que les assertions suivantes sont vraies durant l'exécution de l'algorithme :

- $\forall x_i \in \mathcal{X}, \Delta(x_i, bi_i) = \bigoplus_{x_j \in V(x_i)} \Delta(x_i, x_j, bi_i)$,
- de même pour les bornes supérieures, et
- $c_\varnothing = \bigoplus_{x_i \in \mathcal{X}} \Delta(x_i)$.

Pour prouver que l'algorithme est correct, nous utiliserons les invariants suivants :

1. ligne **34**, toutes les variables sont CNB,
2. ligne **34**, si x_i n'est pas dans Q et que x_j, voisin de x_i, n'est pas non plus dans Q, alors les bornes inférieure et supérieure de l'intervalle de x_i ont un support par rapport à x_j, et les bornes inférieure et supérieure de l'intervalle de x_j ont un

Procédure 30 – CAB

```
   pour chaque x_i ∈ 𝒳 faire
   |  Δ(x_i) ← 0 ;
   |  Δ(x_i, bi_i) ← 0 ;
   |  Δ(x_i, bs_i) ← 0 ;
   |  pour chaque x_j ∈ V(x_i) faire
   |  |  Δ(x_i, x_j, bi_i) ← 0 ;
   |  |  Δ(x_i, x_j, bs_i) ← 0 ;
   CNB ;
   Q ← 𝒳 ;
34 tant que (Q ≠ ∅) faire
   |  x_j ← Extraire(Q) ;
   |  drapeau ← faux ;
35 |  pour chaque x_i ∈ V(x_j) faire
36 |  |  si (ProjectionBorneInférieure(x_i, x_j)) alors
37 |  |  |  si (SuppressionBorneInférieure2(x_i)) alors
38 |  |  |  |  Q ← Q ∪ {x_i} ;
   |  |  [ Même chose pour la borne supérieure du domaine de x_i ]
39 |  |  drapeau ← drapeau ∨ ProjectionUnaireBornes(x_i) ;
40 |  |  si (ProjectionBorneInférieure(x_j, x_i)) alors
41 |  |  |  si (SuppressionBorneInférieure2(x_j)) alors
42 |  |  |  |  Q ← Q ∪ {x_j} ;
   |  |  [ Même chose pour la borne supérieure du domaine de x_j ]
43 |  drapeau ← drapeau ∨ ProjectionUnaireBornes(x_j) ;
44 |  si (drapeau) alors
   |  |  pour chaque x_i ∈ 𝒳 faire
45 |  |  |  si (SuppressionBorneInférieure2(x_i)) alors
46 |  |  |  |  Q ← Q ∪ {x_i} ;
   |  |  |  [ Même chose pour la borne supérieure du domaine de x_i ]
```

support par rapport à x_i.

L'algorithme utilise principalement les fonctions suivantes : la procédure 30, CAB, établit la cohérence d'arc aux bornes ; la fonction 28, ProjectionBorneInférieure, établit le support d'une borne inférieure d'un domaine et retourne vrai si le coût projeté est strictement positif ; la fonction 29, SuppressionBorneInférieure2, met à jour la borne inférieure d'un domaine en enlevant des valeurs non valides et retourne vrai si la borne inférieure a effectivement été modifiée ; la fonction 25, ProjectionUnaireBornes, établit un support unaire pour une variable et retourne vrai si elle a fait augmenter $c_\varnothing$.

La première fois que l'algorithme atteint la ligne **34**, tout le problème est bien CNB, car la procédure CNB a été appelée auparavant. Il faut maintenant vérifier que les valeurs gardent cette propriété. Le support unaire ne peut être perdu qu'à la suite d'une projection (lignes **36** et **40**). Chaque projection est donc suivie d'une procédure de rétablissement du support unaire (lignes **39** et **43**).

Une valeur peut être rendue invalide pour deux raisons : soit son coût unaire augmente, soit $c_\varnothing$ augmente. Le coût unaire peut augmenter à la suite d'une projection. Si une projection fait augmenter un coût unaire, alors les fonctions ProjectionBorneInférieure ou ProjectionBorneSupérieure retourne vrai. À chaque fois que c'est le cas, on appelle la fonction SuppressionBorneInférieure ou SuppressionBorneSupérieure pour mettre à jour la borne modifiée (lignes **37** et **41**). $c_\varnothing$ ne peut augmenter qu'à la suite d'une projection unaire, à condition que ProjectionUnaireBornes retourne vrai. Si c'est le cas, alors le booléen *drapeau* prend la valeur vrai et le code sous la ligne **44** vérifie que toutes les valeurs de chaque variable sont valides.

L'invariant 1 est donc bien respecté. Examinons maintenant le second invariant. Au début de l'algorithme, toutes les variables sont empilées dans Q, ce qui rend cet invariant vrai. Lorsqu'une variable x_i est dépilée, les projections des lignes **36** et **40** permettent aux bornes de x_i d'établir un support avec ses voisins, ainsi qu'aux bornes des intervalles des voisins de x_i d'établir un support avec elle.

Ce support ne peut être brisé que par l'augmentation d'un coût binaire ou la suppression d'une valeur. Le premier cas ne se produit jamais. Dans le second cas, il faut empiler la variable dont la valeur a été supprimée (lignes **38**, **42** et **46**). Bien évidemment, à chaque fois qu'une borne est mise à jour, il faut effacer les Δ retraçant les coûts qui ont été projetés sur la valeur effacée (lignes **31** et **33**). Le second invariant est donc respecté ; l'algorithme est correct. □

Propriété 4.12 *L'algorithme a une complexité temporelle de* $\mathcal{O}(ed^2 + \min\{\top, nd\} \times n)$ *et une complexité spatiale de* $\mathcal{O}(e)$.

Preuve 4.13 Comme pour CA, la boucle de la ligne **34** itère au maximum $n(d+1)$ fois et la boucle ligne **35** itère aussi au maximum $2e(d+1)$ fois. Les complexités cumulées des lignes **36** et **40** sont donc de $\mathcal{O}(ed^2)$.

Calculons maintenant la complexité temporelle cumulée de SuppressionBorneInférieure2. La condition de la boucle **tant que** (ligne **30**) est vraie au maximum nd fois, et donc la complexité de la boucle de la ligne **32** est $\mathcal{O}(ed)$. La complexité cumulée de SuppressionBorneInférieure2 à l'intérieur du **tant que** est $\mathcal{O}(ed)$. Cette fonc-

tion est appelée lignes **37** et **41**, qui peuvent respectivement être appelées au maximum nd et $\min\{\top, nd\} \times n$ fois. La complexité cumulée de la procédure est donc de $\mathcal{O}(ed + \min\{\top, nd\} \times n)$. La complexité temporelle est donc de $\mathcal{O}(ed^2 + \min\{\top, nd\} \times n)$.

Concernant la complexité spatiale, elle est donnée par le nombre de Δ que nous utilisons : trois par variable, et quatre par fonction de coût binaire. La complexité spatiale est donc de $\mathcal{O}(e)$. □

Il est important de noter que la complexité temporelle est divisée par d par rapport à CA. De plus, la complexité spatiale ne dépend pas de d. Ce constat est conforme à nos souhaits, dans la mesure où cette cohérence locale est applicable à des problèmes à très grands domaines.

4.1.3.2 La cohérence ∅-inverse (C∅I)

Pour certaines fonctions de coût, CAB peut sembler trop faible par rapport à CA, d'autant plus qu'il semble parfois possible de faire augmenter $c_\varnothing$ relativement facilement. Considérons par exemple la fonction de coût suivante :

$$c_{12} : \left\{ \begin{array}{lcl} I_1 \times I_2 & \rightarrow & E \\ (v_1, v_2) & \mapsto & v_1 + v_2 \end{array} \right. \qquad I_1 = I_2 = [1..10]$$

CA peut ici augmenter le $c_\varnothing$, en projetant un coût de deux de c_{12} sur chaque valeur de x_1, puis en projetant ce coût sur $c_\varnothing$. CAB ne peut en revanche pas faire augmenter le $c_\varnothing$ puisqu'il lui est impossible de projeter un coût sur une valeur intérieure à x_1 ou x_2.

Il existe cependant un moyen de faire augmenter ce $c_\varnothing$, tout en gardant une complexité spatiale indépendante de d. C'est la *cohérence ∅-inverse*. Celle-ci projette des coûts directement de la fonction binaire sur $c_\varnothing$.

Définition 4.12 *Une fonction de coût c est ∅-inverse cohérente (C∅I) si :*

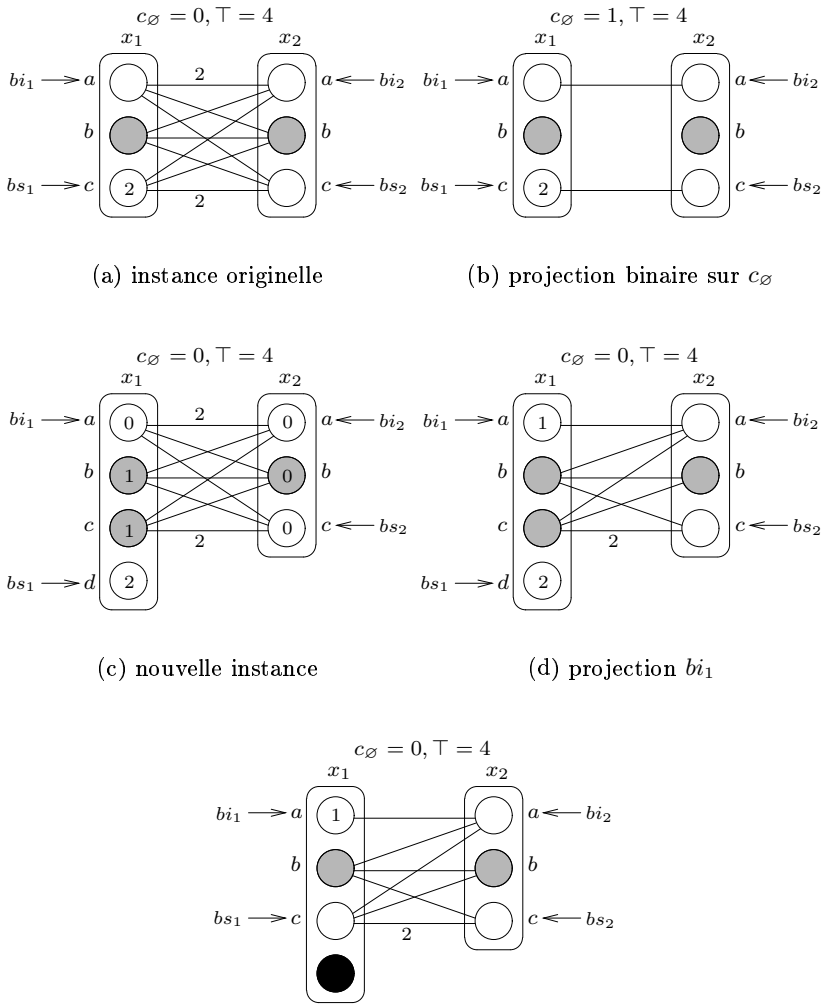
$$\exists a \in \ell(var(c)), c(a) = 0$$

(a est alors appelée support r-aire de c, où r est l'arité de c). Un RCP est ∅-inverse cohérent si chaque fonction de coût l'est.

L'exemple suivant illustre une application de C∅I sur un RCP.

Exemple 4.6 *Considérons par exemple la figure 4.9(a). CAB peut projeter un coût binaire sur bi_1 et bs_1 (ou bi_2 et bs_2), mais elle ne peut pas faire augmenter $c_\varnothing$. Malgré tout, pour n'importe quelle affectation de x_1 et x_2, la fonction binaire donnera au moins un coût de 1. Un tel coût peut donc être projeté sur $c_\varnothing$, comme décrit dans la figure 4.9(b). La fonction de coût est donc maintenant ∅-inverse cohérente.*

En revanche, on peut constater que l'établissement de CAB ou CA établit automatiquement C∅I. Supposons par exemple que $v_j \in I_j$ soit le support de $bi_i \in I_i$ par rapport à c_{ij}, alors (bi_i, v_j) est un support binaire de c_{ij}, et donc cette fonction de coût



(a) instance originelle

(b) projection binaire sur $c_\varnothing$

(c) nouvelle instance

(d) projection bi_1

(e) mise à jour de bs_1

FIG. 4.9 – Un établissement de C∅I problématique.

est C∅I. C'est pourquoi il est important d'établir C∅I avant CAB. Le fait que l'ordre d'établissement des cohérences locales influe sur les valeurs de l'instance d'arrivée — en d'autres termes, le fait que les cohérences locales ne confluent pas — est classique dans les RCP (cf. [Sch00, CS04]).

Malgré tout, établir C∅I avant CAB n'est pas encore totalement satisfaisant. L'exemple suivant illustre cela.

Exemple 4.7 *Considérons le RCP de la figure 4.9(c). Il est déjà C∅I, puisque la fonction binaire présente un coût nul lorsque x_1 vaut d et x_2 vaut c. CAB projette un coût de un sur bi_1 (cf.* FIG. *4.9(d)). Ensuite, pour une raison quelconque, le solveur élimine la valeur d de x_1 (cf.* FIG. *4.9(e)). Le réseau est toujours C∅I et CAB peut projeter un coût de un sur bs_1.*

Mais on peut faire mieux. On pourrait par exemple préférer annuler la projection sur bi_1 et appliquer C∅I. Dans ce cas, $c_\varnothing$ augmenterait, comme montré sur la figure 4.10(g).

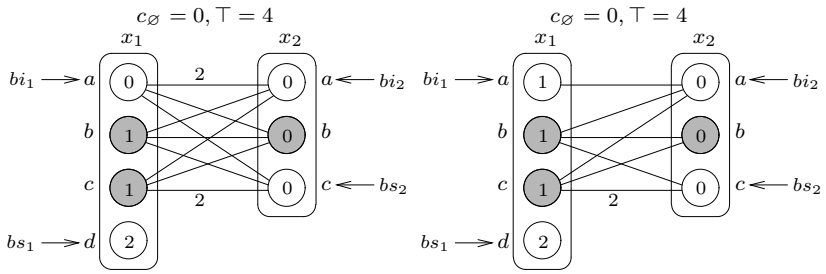
Il est donc particulièrement intéressant de pouvoir être en mesure d'« annuler » une projection, si l'on peut faire augmenter $c_\varnothing$ par l'application de C∅I. Il existe en revanche une légère difficulté : le coût que l'on avait projeté peut déjà avoir servi à faire augmenter $c_\varnothing$, par CNB. L'exemple suivant illustre ce cas de figure.

Exemple 4.8 *Considérons la figure 4.10(a), qui est C∅I. Par CAB, on projette un coût de un sur bi_1 (cf.* FIG. *4.10(b)), puis par CNB, on projette un coût de un sur $c_\varnothing$ (cf.* FIG. *4.10(c)). Supposons encore une fois que la valeur d de x_1 soit supprimée par le solveur, pour une raison arbitraire (cf. 4.10(d)). Le réseau est toujours C∅I, mais on voudrait annuler la projection unaire sur $c_\varnothing$ (cf.* FIG. *4.10(e)), puis la projection sur bi_1 (cf.* FIG. *4.10(f)), pour pouvoir appliquer $c_\varnothing$. Le résultat est donné sur la figure 4.10(g). Même si la valeur de $c_\varnothing$ n'a pas augmenté, les coûts ont été transférés de la fonction de coût binaire sur la fonction unaire, ce qui constitue un avantage.*

La fonction $\mathsf{ProjectionBinaire}(x_i, x_j)$ établit C∅I sur c_{ij}. Elle utilise une nouvelle structure de données, $\Delta(x_i, x_j)$, qui stocke le coût projeté directement de c_{ij} sur $c_\varnothing$ par C∅I. On a donc :

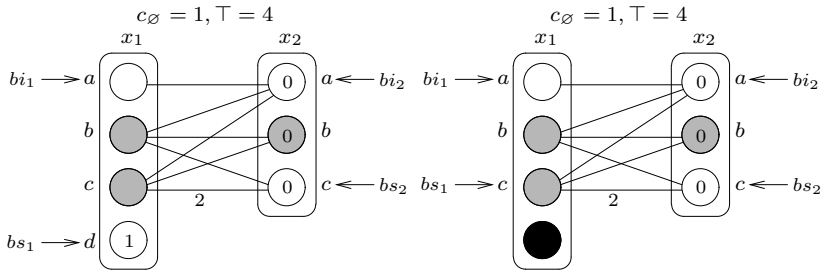
$$\forall c_{ij} \in \mathcal{C}, \forall (v_i, v_j) \in I_i \times I_j,$$
$$c_{ij}(v_i, v_j) = \mathbf{c_{ij}}(v_i, v_j) \ominus \left(\begin{cases} \Delta(x_i, x_j, v_i) & \text{si } v_i \in \{bi_i, bs_i\} \\ 0 & \text{sinon} \end{cases} \right.$$
$$\left. \oplus \begin{cases} \Delta(x_j, x_i, v_j) & \text{si } v_j \in \{bi_j, bs_j\} \\ 0 & \text{sinon} \end{cases} \oplus \Delta(x_i, x_j) \right)$$

La fonction procède ainsi : si l'on se rend compte que $\min_{\substack{v_i \in I_i \\ w_j \in I_j}} \{\mathbf{c_{ij}}(v_i, w_j)\} \ominus \Delta(x_i, x_j)$ n'est pas nul, alors on pourra faire augmenter $c_\varnothing$. Il faudra pour cela éventuellement annuler des projections et des projections unaires (voir exemple 4.8). Cela fait, on projette un coût non nul de la fonction de coût binaire sur $c_\varnothing$.



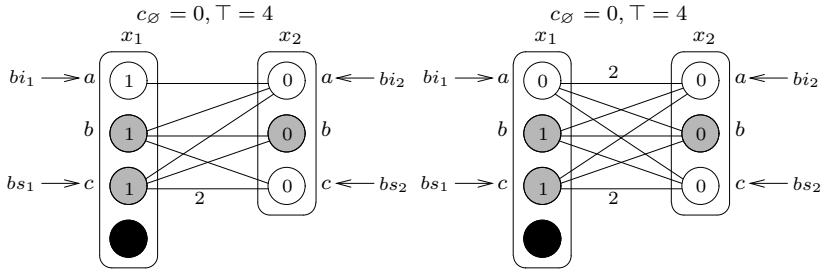
(a) instance originelle

(b) projection sur bi_1



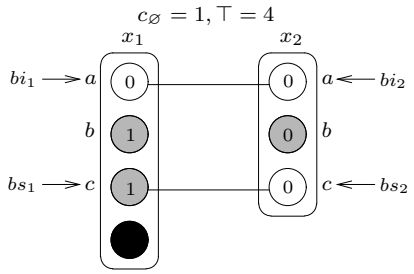
(c) projection unaire de x_1

(d) suppression de la valeur d de x_1



(e) annulation de la projection unaire

(f) annulation de la projection



(g) projection de c_{12} sur $c_\varnothing$

FIG. 4.10 – Comportement de C∅I.

La propriété suivante prouve la correction de l'algorithme 32 (CAB2) et donne ses complexités temporelle et spatiale.

Fonction 31 – ProjectionBinaire($x_i \in \mathcal{X}, x_j \in V(x_i)$) : booléen

47 $min \leftarrow \min_{\substack{v_i \in I_i \\ v_j \in I_j}} \{\mathbf{c_{ij}}(v_i, v_j)\} \ominus \Delta(x_i, x_j)$;
si $(min = 0)$ **alors**
  retourner faux ;
$c_\varnothing^0 \leftarrow c_\varnothing$;
$\Delta(x_i, bi_i) \leftarrow \Delta(x_i, bi_i) \ominus \Delta(x_i, x_j, bi_i)$;
$\Delta(x_i, bs_i) \leftarrow \Delta(x_i, bs_i) \ominus \Delta(x_i, x_j, bs_i)$;
$\Delta(x_i, bi_j) \leftarrow \Delta(x_i, bi_j) \ominus \Delta(x_j, x_i, bi_j)$;
$\Delta(x_i, bs_j) \leftarrow \Delta(x_i, bs_j) \ominus \Delta(x_j, x_i, bs_j)$;
$\Delta(x_i, x_j, bi_i) \leftarrow 0$;
$\Delta(x_i, x_j, bs_i) \leftarrow 0$;
$\Delta(x_j, x_i, bi_j) \leftarrow 0$;
$\Delta(x_j, x_i, bs_j) \leftarrow 0$;
$\Delta(x_i) \leftarrow 0$;
$\Delta(x_j) \leftarrow 0$;
$c_\varnothing \leftarrow (c_\varnothing \ominus (\Delta(x_i) \oplus \Delta(x_j))) \oplus min$;
si $(c_\varnothing = \top)$ **alors**
  lever incohérence ;
ProjectionBorneInférieure(x_i, x_j) ;
ProjectionBorneSupérieure(x_i, x_j) ;
ProjectionBorneInférieure(x_j, x_i) ;
ProjectionBorneSupérieure(x_j, x_i) ;
ProjectionUnaireBornes(x_i) ;
ProjectionUnaireBornes(x_j) ;
retourner $(c_\varnothing > c_\varnothing^0)$;

Propriété 4.13 *L'algorithme établissant CAB avec C∅I est correct. Il a une complexité temporelle et spatiale de* $\mathcal{O}(ed^3 + \min\{\top, nd\} \times n)$ *et* $\mathcal{O}(e)$ *respectivement.*

Preuve 4.14 La procédure CAB2 ne présente que très peu de différences avec CAB. Il s'agit de l'ajout de deux lignes, écrites en noir alors que les lignes inchangées sont grisées. La première initialise les Δ des projections binaires. La seconde appelle la fonction 31 (ProjectionBinaire). Cette fonction établit un support binaire et retourne vrai si elle a fait augmenter $c_\varnothing$.

Il nous faut tout d'abord vérifier que les soustractions sont correctes.

- $\forall x_i \in \mathcal{X}, \forall x_j \in V(x_i), \forall v_i \in I_i, \forall v_j \in I_j, \Delta(x_i, x_j) \leq \mathbf{c_{ij}}(v_i, v_j)$,
- $\forall x_i \in \mathcal{X}, \forall x_j \in V(x_i), \Delta(x_i, x_j, bi_i) \leq \mathbf{c_{ij}}(bi_i, bi_j) \ominus \Delta(x_i, x_j) \wedge \Delta(x_i, x_j, bi_i) \leq \mathbf{c_{ij}}(bi_i, bs_j) \ominus \Delta(x_i, x_j)$,

Procédure 32 – CAB2

```
    pour chaque x_i ∈ 𝒳 faire
        Δ(x_i) ← 0 ;
        Δ(x_i, bi_i) ← 0 ;
        Δ(x_i, bs_i) ← 0 ;
        pour chaque x_j ∈ V(x_i) faire
            Δ(x_i, x_j, bi_i) ← 0 ;
            Δ(x_i, x_j, bs_i) ← 0 ;
            Δ(x_i, x_j) ← 0 ;
    CNB ;
    Q ← 𝒳 ;
48  tant que (Q ≠ ∅) faire
        x_j ← Extraire(Q) ;
        drapeau ← faux ;
49      pour chaque x_i ∈ V(x_j) faire
            drapeau ← drapeau ∨ ProjectionBinaire(x_i, x_j) ;
            si (ProjectionBorneInférieure(x_i, x_j)) alors
                si (SuppressionBorneInférieure2(x_i)) alors
                    Q ← Q ∪ {x_i} ;
            [ Même chose pour la borne supérieure du domaine de x_i ]
            drapeau ← drapeau ∨ ProjectionUnaireBornes(x_i) ;
            si (ProjectionBorneInférieure(x_j, x_i)) alors
                si (SuppressionBorneInférieure2(x_j)) alors
                    Q ← Q ∪ {x_j} ;
            [ Même chose pour la borne supérieure du domaine de x_j ]
        drapeau ← drapeau ∨ ProjectionUnaireBornes(x_j) ;
50      si (drapeau) alors
            pour chaque x_i ∈ 𝒳 faire
                si (SuppressionBorneInférieure2(x_i)) alors
                    Q ← Q ∪ {x_i} ;
                [ Même chose pour la borne supérieure du domaine de x_i ]
```

- $\forall x_i \in \mathcal{X}, \forall x_j \in V(x_i), \Delta(x_i, x_j, bi_i) \leq \min \left\{ \begin{array}{l} \mathbf{c_{ij}}(bi_i, bi_j) \ominus (\Delta(x_i, x_j) \oplus \Delta(x_j, x_i, bi_j)), \\ \min_{v_j \in \mathring{I}_j} \{\mathbf{c_{ij}}(bi_i, v_j)\} \ominus \Delta(x_i, x_j), \\ \mathbf{c_{ij}}(bi_i, bs_j) \ominus (\Delta(x_i, x_j) \oplus \Delta(x_j, x_i, bs_j)) \end{array} \right.$
- les assertions symétriques des trois précédentes appliquées aux bornes supérieures sont aussi vraies,
- les autres assertions (telles que décrites dans la preuve de la proposition 4.11) restent inchangées.

On peut facilement vérifier que ces assertions sont vraies durant l'exécution du programme.

Par rapport à CAB, nous ajoutons un troisième invariant pour prouver la correction de l'algorithme. Les trois propriétés de l'invariant sont donc maintenant :

1. ligne **48**, toutes les variables sont CNB,
2. ligne **48**, si x_i n'est pas dans Q et que x_j, voisin de x_i, n'est pas non plus dans Q, alors les bornes inférieure et supérieure de l'intervalle de x_i ont un support par rapport à x_j, et les bornes inférieure et supérieure de l'intervalle de x_j ont un support par rapport à x_i.
3. ligne **48**, si x_i et x_j ne sont pas dans Q, alors c_{ij} a un support binaire.

Vérifions que ProjectionBinaire établit bien C∅I. La ligne **47** trouve le minimum de la fonction de coût binaire étudiée. Si celui-ci n'est pas nul, alors il faut le projeter sur $c_\varnothing$. Pour que l'opération soit correcte, il faut annuler toutes les projections sur les bornes, puis les projections des fonctions de coût unaires sur $c_\varnothing$. Pour ensuite laisser les deux variables CAB, il faut éventuellement re-projeter des coûts sur les bornes des deux variables, puis re-projeter des coûts des variables sur $c_\varnothing$. ProjectionBinaire trouve donc bien un support binaire, sans détruire les supports des bornes des variables de la fonction de coût, ni leur support unaire.

Vérifions maintenant que le troisième invariant est vrai. Au début de l'algorithme, toutes les variables sont dans Q, donc les invariants sont respectés. Un support binaire de c_{ij} ne peut être brisé que par l'augmentation d'un coût binaire, ou la suppression d'une valeur dans x_i ou x_j. Le premier cas ne se produit jamais. À chaque fois qu'une variable x_i perd une valeur, x_i est empilée dans Q, et le support sera vérifié ultérieurement. L'invariant est donc bien respecté.

Concernant la complexité temporelle de l'algorithme, ProjectionBinaire s'exécute en temps $\mathcal{O}(d^2)$. La complexité temporelle de l'algorithme est donc $\mathcal{O}(ed^2 + \min\{\top, nd\} \times n + ed^3) = \mathcal{O}(ed^3 + \min\{\top, nd\} \times n)$. La complexité spatiale est, elle, inchangée : elle reste à $\mathcal{O}(e)$. □

Il pourrait être possible de réduire la complexité temporelle de l'algorithme précédent d'un facteur de d en utilisant une structure ordonnant les coûts binaires. Mais cela augmenterait la complexité spatiale de l'algorithme d'un facteur d^2, ce qui est inacceptable. Une autre possibilité consiste, à chaque fois que c'est possible, à exploiter la sémantique d'une fonction de coût binaire, pour réduire la complexité temporelle. On pourra alors utiliser une des trois propriétés suivantes.

Propriété 4.14 *Si le minimum de toutes les fonctions de coût binaires, lorsqu'une des variables a une valeur fixée, peut être trouvé en temps constant, alors la complexité temporelle de CAB devient* $\mathcal{O}(ed + \min\{\top, nd\} \times n)$*, tandis que la complexité spatiale reste inchangée.*

Preuve 4.15 Si, pour toute fonction de coût c_{ij}, et en tout point v_i de I_i, $\min_{v_j \in I_j}\{c_{ij}(v_i, v_j)\}$ peut être calculé en temps constant, alors les lignes **36** et **40** prennent un temps cumulé de $\mathcal{O}(ed)$, et donc CAB peut être établi $\mathcal{O}(ed + \min\{\top, nd\} \times n)$. □

Propriété 4.15 *Si le minimum de toutes les fonctions de coût binaires peut être trouvé en temps* $\mathcal{O}(d)$*, alors la complexité temporelle de CAB avec C∅I devient* $\mathcal{O}(ed^2 + \min\{\top, nd\} \times n)$*, tandis que la complexité spatiale reste inchangée.*

Preuve 4.16 Si le minimum de toutes les fonctions de coût binaires peut être trouvé en temps $\mathcal{O}(d)$, alors la ligne **47** prend un temps proportionnel à d, ce qui ramène la complexité de tout l'algorithme à $\mathcal{O}(ed^2 + \min\{\top, nd\} \times n)$. □

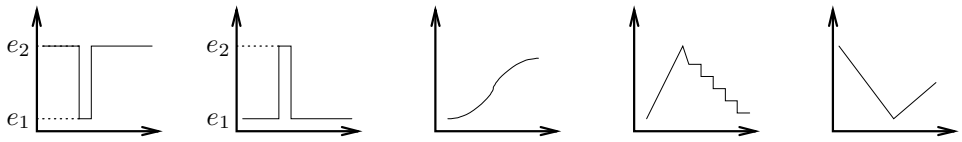
Propriété 4.16 *Si le minimum de chaque fonction de coût, binaire et unaire peut être trouvé en temps constant, alors la complexité temporelle de CAB avec C∅I devient* $\mathcal{O}(ed + \min\{\top, nd\} \times n)$*, tandis que la complexité spatiale reste inchangée.*

Preuve 4.17 Si le minimum de chaque fonction de coût peut se trouver en temps constant, alors les fonctions ProjectionUnaireBornes, ProjectionBinaire, ProjectionBorneInférieure et ProjectionBorneSupérieure prennent un temps constant.

En conséquence, une itération de la boucle **pour chaque** de la ligne **49**, prend un temps constant si l'on excepte les fonctions SuppressionBorneInférieure2 et SuppressionBorneSupérieure2. Comme la boucle itère au maximum $\mathcal{O}(ed)$ fois et que SuppressionBorneInférieure2 à l'intérieur de la condition de la ligne **30** a une complexité cumulée égale à $\mathcal{O}(ed)$, alors la complexité totale de la boucle de la ligne **49** est de $\mathcal{O}(ed)$. De plus, la complexité du code sous la condition de la ligne **50** reste inchangée, donc la complexité temporelle de l'algorithme passe à $\mathcal{O}(ed + \min\{\top, nd\} \times n)$. □

À la manière de l'article présentant AC-5 [HDT92], nous donnons maintenant quelques exemples de fonctions de coût sur lesquelles CAB avec C∅I peut être établie sensiblement plus rapidement que dans le pire cas. L'article mentionne notamment trois types de contraintes : les contraintes fonctionnelles, les contraintes anti-fonctionnelles et les contraintes monotones. Les auteurs généralisent ensuite ces types de contraintes en définissant les contraintes fonctionnelles par morceaux, les contraintes anti-fonctionnelles par morceaux et les contraintes monotones par morceaux. Nous n'évoquerons pas ces types de contraintes par morceaux, qui font intervenir des structures complexes qu'il serait sans doute trop long de présenter ici. Nous allons maintenant donner des définitions possibles de fonctions de coût fonctionnelles et anti-fonctionnelles.

Définition 4.13 *Une fonction de coût* c_{ij} *est* fonctionnelle *par rapport à la variable* x_i *si :*



(a) fonctionnelle (b) anti-fonctionnelle (c) monotone (d) semi-convexe (e) non semi-convexe

FIG. 4.11 – Caracteristiques de quelques fonctions de coût.

– $\forall (v_i, v_j) \in D_i \times D_j, c_{ij}(v_i, v_j) \in \{e_1, e_2\}, e_1 < e_2$ *et*
– $\forall v_i \in D_i, |\{v_j \in D_j, c_{ij}(v_i, v_j) = e_1\}| \leq 1$

Une fonction de coût c_{ij} *est* anti-fonctionnelle *par rapport à la variable* x_i *si :*
– $\forall (v_i, v_j) \in D_i \times D_j, c_{ij}(v_i, v_j) \in \{e_1, e_2\}, e_1 < e_2$ *et*
– $\forall v_i \in D_i, |\{v_j \in D_j, c_{ij}(v_i, v_j) = e_2\}| \leq 1$

Exemple 4.9 *La fonction de coût* $c_=(x,y) = \begin{cases} 0 & si\ x = y \\ 1 & sinon \end{cases}$ *est fonctionnelle par rapport aux deux variables (cf. figure 4.11(a)).*

La fonction de coût $c_{\neq}(x,y) = \begin{cases} 0 & si\ x \neq y \\ 1 & sinon \end{cases}$ *est anti-fonctionnelle par rapport aux deux variables (cf. figure 4.11(b)).*

Il s'avère que les fonctions de coût anti-fonctionnelles et monotones (cf. figure 4.11(c)) appartiennent à la classe des fonctions *semi-convexes*. Informellement, les fonctions semi-convexes n'ont qu'un seul sommet. Par exemple, la fonction sur la figure 4.11(d) est semi-convexe, alors que la fonction sur la figure 4.11(e) ne l'est pas. La fonction $x, y \mapsto x^2 - y^2$ est par exemple semi-convexe par rapport à y, mais pas par rapport à x.

Définition 4.14 *Une fonction* $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *est semi-convexe par rapport à la première variable si :*

$$\forall (y, a) \in \mathbb{N}^2, \{x \in \mathbb{R} : f(x, y) > a\}$$

est un intervalle.

f est semi-convexe [KMMR01] si elle est semi-convexe par rapport à toutes ses variables.

Si une fonction de coût binaire est semi-convexe par rapport à une seule variable, alors les minima de cette fonction peuvent être trouvés sur les bords de la matrice de coûts, et donc en temps $\mathcal{O}(d)$. Établir CAB avec C∅I peut donc se faire dans ce cas en temps proportionnel à d^2. Si la fonction est semi-convexe par rapport aux deux variables, comme la fonction $x, y \mapsto x + y$, le minimum peut se trouver en temps constant, car il est situé dans un des quatre coins de la matrice de coûts. Dans ce cas, CAB avec C∅I peut s'établir en temps linéaire par rapport à d.

On peut donc établir CAB avec C∅I sur les fonctions anti-fonctionnelles en temps $\mathcal{O}(ed + \min\{\top, nd\} \times n)$. En revanche, pour les fonctions de coût fonctionnelles, seules les propriétés 4.14 et 4.15 s'appliquent. CAB peut donc être établi en temps $\mathcal{O}(ed + \min\{\top, nd\} \times n)$, et CAB avec C∅I, en temps $\mathcal{O}(ed^2 + \min\{\top, nd\} \times n)$.

Un autre type de fonctions de coût apparaît fréquemment lorsque l'on modélise des problèmes réels : les fonctions de distance, de la forme $c_{1,2} = |x_2 - x_1 - d|$, où d est une constante définissant la distance entre x_1 et x_2. Cette fonction de coût permet de s'assurer que si x_2 n'est pas à une distance de d par rapport à x_1, alors une pénalité proportionnelle à l'éloignement est appliquée. Le minimum de cette fonction étant calculable en temps constant de la manière suivante :

$$min = \begin{cases} bi_1 + d - bs_2 & \text{si } bs_2 < bi_1 + d, \\ bi_2 - bs_1 - d & \text{si } bi_2 > bs_1 + d, \\ 0 & \text{sinon,} \end{cases}$$

on peut appliquer CAB avec C∅I sur ce type de fonction en temps linéaire par rapport à d.

Il est enfin bien sûr possible de rechercher les minima par n'importe quelle méthode (comme par la recherche des racines de la dérivée d'une fonction analytique). Et lorsque rechercher le minimum s'avère trop difficile (cela peut être parfois un problème NP-complet), on peut simplement rechercher une approximation du minimum par défaut, grâce à toute méthode heuristique.

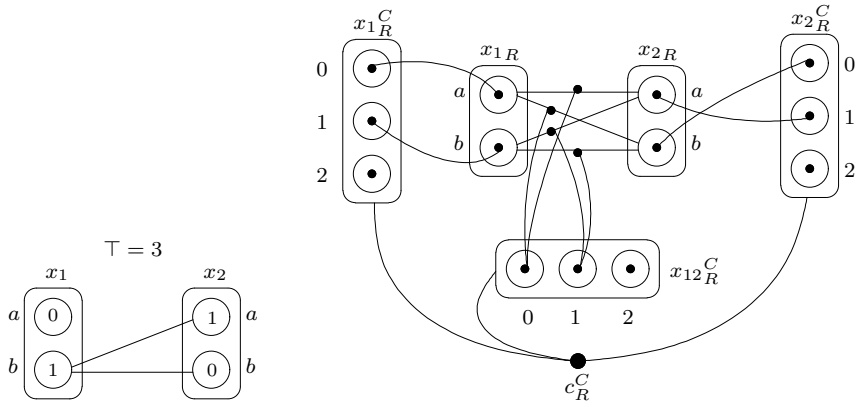
4.1.3.3 Relations avec la 2B-cohérence

La cohérence d'arc aux bornes s'inspire largement de la 2B-cohérence [Lho93] (voir définition 3.4). Si l'on modélise un réseau de contraintes classiques par un réseau de contraintes pondérées en donnant à $\top$ la valeur un, alors appliquer CAB sur ce réseau rend le problème de contraintes classiques 2B-cohérent. CAB peut donc être vue comme une extension de la 2B-cohérence. Pour prolonger la comparaison, nous pouvons utiliser une modélisation permettant d'exprimer des fonctions de coût par des contraintes dures proposée dans [PRB00]. Si l'on a un RCP, vaut-il mieux le transformer en RC pour y établir la 2B-cohérence, ou bien le résoudre avec CAB ? Nous allons tenter de répondre à cette question en comparant le pouvoir d'élagage dans l'arbre de séparation et d'évaluation de ces deux cohérences locales.

La modélisation de fonctions de coût par des contraintes classiques faite par [PRB00] propose l'introduction de *coûts réifiés* dans un schéma appelé *Soft as Hard*. L'exemple suivant illustre la réification de coûts.

Exemple 4.10 *Considérons par exemple le RCP de la figure 4.12(a). Il contient les variables x_1 et x_2, une fonction de coût binaire c_{12} et deux fonctions de coût unaires c_1 et c_2. Pour la clarté de l'énoncé, nous indexerons toute variable ou fonction de coût réifiée de la lettre* R.

La figure 4.12(b) décrit le réseau de contraintes classiques modélisant le réseau originel. Pour chaque fonction de coût, on crée une nouvelle variable, dite « variable de



(a) réseau de contraintes pondérées

(b) un RC équivalent

FIG. 4.12 – Un RCP et son RC équivalent.

coût » de la fonction, dont le domaine va de 0 *à* $\top$*. Dans la suite, nous marquerons toutes les variables de coût par des* C *en exposant. Chaque fonction de coût est ensuite remplacée par une contrainte dure, où l'arité a été augmentée de un ; ainsi, une fonction de coût unaire devient une contrainte dure binaire. La nouvelle variable de la contrainte dure est sa variable de coût, qui sert à exprimer le coût d'une affectation des variables de la fonction de coût.*

Par exemple, la fonction de coût c_1 *est remplacée par la contrainte* c_{1R}*, portant sur la variable* x_1*, mais aussi sur la nouvelle variable de coût* $x_1{}_R^C$*. La contrainte* c_{1R} *accepte toutes les paires* (v_1, a)*, telles que* $v_1 \in I_1$ *et* $c_1(v_1) = a$*. De même, nous créons une variable de coût* $x_2{}_R^C$ *pour la fonction de coût* c_2*, et nous remplaçons cette dernière par la fonction de coût réifiée* c_{2R}*. Nous ajoutons aussi la variable* $x_{12}{}_R^C$*, qui est la variable de coût de* c_{12}*, et remplaçons* c_{12} *par la contrainte ternaire* c_{12R}*. Enfin, nous ajoutons une dernière contrainte, qui porte sur toutes les variables de coûts, et qui s'assure que la somme de celles-ci est bien inférieure à* $\top$ *:* $x_1{}_R^C + x_2{}_R^C + x_{12}{}_R^C < \top$*. Nous avons maintenant un réseau de contraintes dures.*

L'exemple précédent motive la définition suivante du schéma *Soft as Hard*.

Définition 4.15 *Soit un RCP* $\mathcal{P} = \langle \mathcal{S}, \mathcal{X}, \mathcal{I}, \mathcal{C} \rangle$*. Soit* $\mathcal{P}_R = \langle \mathcal{X}_R, \mathcal{I}_R, \mathcal{C}_R \rangle$ *le réseau de contraintes classiques tel que :*

- *l'ensemble de variables* $\mathcal{X}_R$ *est l'union de :*
  - $\mathcal{X}$*, l'ensemble des variables du problème d'origine,*
  - $\mathcal{C}_1{}_R^C$*, l'ensemble des variables de coût* $x_i{}_R^C$ *des fonctions de coût unaires réifiées,*
  - $\mathcal{C}_2{}_R^C$*, l'ensemble des variables de coût* $x_{ij}{}_R^C$ *des fonctions de coût binaires réifiées.*
- *les intervalles* $\mathcal{I}_R$ *sont :*
  - $I_{iR} \equiv I_i$ *pour les variables originelles* x_{iR}*,*

  - $[bi_{i_R}^C..bs_{i_R}^C] \equiv [0, \top]$ *pour les variables de coûts unaires* $x_{i_R}^C$,
  - $[bi_{ij_R}^C..bs_{ij_R}^C] \equiv [0, \top]$ *pour les variables de coûts binaires* $x_{ij_R}^C$.
- *l'ensemble des fonctions de coût* $\mathcal{C}_R$ *est composé de :*
  - $\forall x_i \in \mathcal{X}, c_{i_R} \equiv \{(v_i, \mathbf{c_i}(v_i)) | v_i \in I_i\}$,
  - $\forall c_{ij} \in \mathcal{C}_1, c_{ij_R} \equiv \{(v_i, v_j, \mathbf{c_{ij}}(v_i, v_j)) | v_i \in I_i, v_j \in I_j\}$,
  - $c_R^C \equiv \sum_{x_i \in \mathcal{X}} x_{i_R}^C + \sum_{c_{ij} \in \mathcal{C}_1} x_{ij_R}^C < \top$, *une contrainte supplémentaire s'assurant que la somme des variables de coûts ne dépasse pas* $\top$.

Le problème $\mathcal{P}_R$ a une solution si et seulement si $\mathcal{P}$ en a une, et le coût d'une solution est la somme des valeurs des variables de coûts.

Notons d'ores et déjà qu'il n'est pas simple de comparer le pouvoir d'élagage des deux cohérences locales, puisqu'en général, dans les RCP, les algorithmes ne convergent pas vers une solution unique. La comparaison des cohérences locales doit donc prendre en compte le caractère potentiellement non confluent de celles-ci. Nous allons définir la *force* d'une cohérence locale de la manière suivante.

Définition 4.16 *Il est possible de décomposer l'établissement de chaque propriété de cohérence locale par une suite d'*opérations élémentaires. *Dans le cas de la 2B-cohérence, ces opérations sont :*

- *la suppression de valeurs.*

Dans le cas de CAB avec C∅I, ces opérations sont appelées « transformations préservant l'équivalence » [CS04]. Elles sont :

- *la suppression de valeurs,*
- *la projection,*
- *la projection unaire,*
- *la projection binaire.*

Un réseau de contraintes classiques est manifestement incohérent *si le domaine d'une de ses variables est vide. Un réseau de contraintes pondérées est manifestement incohérent si le domaine d'une de ses variables est vide, ou si* $c_\varnothing = \top$.

Une propriété de cohérence locale $\mathcal{T}$ *est* plus forte *qu'une autre propriété* $\mathcal{T}'$, *si, pour tout problème* $\mathcal{P}$ *tel qu'il existe une application d'opérations élémentaires de* $\mathcal{T}$ *qui transforme* $\mathcal{P}$ *en un problème manifestement incohérent, toute application d'opérations élémentaires de* $\mathcal{T}'$ *sur* $\mathcal{P}$ *le transforme en un problème manifestement incohérent.*

$\mathcal{T}$ *est* strictement plus fort *que* $\mathcal{T}'$ *si* $\mathcal{T}$ *est plus fort que* $\mathcal{T}'$, *et s'il existe un problème* $\mathcal{P}$ *tel que toute application d'opérations élémentaires de* $\mathcal{T}$ *sur* $\mathcal{P}$ *le transforme en problème manifestement incohérent, alors qu'il n'existe pas d'application d'opérations élémentaires de* $\mathcal{T}'$ *qui rende* $\mathcal{P}$ *manifestement incohérent.*

La définition précédente nous aide déjà à comparer CAB et 2B-cohérence sur l'exemple suivant.

Exemple 4.11 *Considérons le RCP défini par trois variables (*x_1, x_2 *et* x_3*), et deux fonctions de coût binaires (*c_{12} *et* c_{13}*). On définit* $I_1 = \{a, b, c, d\}$, $I_2 = I_3 = \{a, b, c\}$ *(avec* $a \prec b \prec c \prec d$*), et les coûts binaires sont donnés dans les matrices de coûts de la figure 4.13. Supposons de plus que* $\top = 2$.

$$\begin{array}{c} \\ (x_2)\end{array}\begin{array}{c}(x_1)\\ \begin{array}{cccc} a & b & c & d \end{array}\\ \begin{array}{c} a\\ b\\ c \end{array}\begin{bmatrix} 1 & 0 & 2 & 1\\ 1 & 0 & 2 & 1\\ 1 & 0 & 2 & 1 \end{bmatrix}\end{array} \qquad \begin{array}{c} \\ (x_3)\end{array}\begin{array}{c}(x_1)\\ \begin{array}{cccc} a & b & c & d \end{array}\\ \begin{array}{c} a\\ b\\ c \end{array}\begin{bmatrix} 1 & 2 & 0 & 1\\ 1 & 2 & 0 & 1\\ 1 & 2 & 0 & 1 \end{bmatrix}\end{array}$$

FIG. 4.13 – Deux matrices de coûts.

On peut aisément vérifier que le réseau réifié est 2B-cohérent. Par exemple, un support de la borne inférieure a du domaine de x_{1R} par rapport à c_{12R} est $(a, a, 1)$. Un support de sa borne supérieure d est $(d, a, 1)$. Les supports des bornes inférieure et supérieure du domaine de $x_{12_R}^C$ sont $(b, a, 0)$ et $(c, a, 1)$. Les autres supports peuvent de même être facilement trouvés.

En revanche, toute application de CAB rend le problème manifestement incohérent, par la projection de coûts binaires sur x_1, et en diminuant progressivement son domaine pour le réduire à l'ensemble vide. L'exemple montre simplement que la 2B-cohérence sur le problème réifié n'est pas strictement plus forte que CAB.

Nous allons maintenant prouver que CAB avec C∅I est plus forte que la 2B-cohérence. Pour cela, nous allons observer l'établissement de CAB avec C∅I sur un problème quelconque $\mathcal{P}$, et l'établissement de 2B-cohérence sur $\mathcal{P}_R$, le problème $\mathcal{P}$ réifié. On considèrera ensuite le problème $\mathcal{P}'_R$, qui est le problème $\mathcal{P}_R$ après un nombre arbitraire d'opérations élémentaires de 2B-cohérence. On montrera par induction que dans tout processus d'établissement de CAB avec C∅I, il existe un problème $\mathcal{P}'$ obtenu à partir de $\mathcal{P}$ après un certain nombre d'opérations élémentaires de CAB avec C∅I tel que $\mathcal{P}'$ est « au moins aussi contraint » que $\mathcal{P}'_R$. Cela prouvera que si la 2B-cohérence transforme $\mathcal{P}'$ en un problème manifestement incohérent, alors CAB avec C∅I le fera aussi et donc que CAB avec C∅I est plus forte que la 2B-cohérence.

Nous définissons maintenant la relation « au moins aussi contraint ».

Définition 4.17 *Soit $\mathcal{P}$ un RCP et $\mathcal{P}_R$ son problème réifié. Soit $\mathcal{P}' = \langle \mathcal{S}', \mathcal{X}', \mathcal{I}', \mathcal{C}' \rangle$ le problème $\mathcal{P}$ obtenu après l'application d'un nombre arbitraire d'opérations élémentaires de CAB avec C∅I. Soit $\mathcal{P}'_R = \langle \mathcal{X}'_R, \mathcal{I}'_R, \mathcal{C}'_R \rangle$ le problème $\mathcal{P}_R$ obtenu après un nombre arbitraire d'opérations de 2B-cohérence.*

$\mathcal{P}'$ est au moins aussi contraint *que $\mathcal{P}'_R$ si :*

1. $\forall x'_i \in \mathcal{X}', I'_i \subseteq I'_{iR}$,
2. $\forall x'_i \in \mathcal{X}', \Delta'(x'_i) \geq bi'^C_{iR}$,
3. $\forall c'_{ij} \in \mathcal{C}', \Delta'(x'_i, x'_j) \geq bi'^C_{ijR}$.

La propriété suivante prouve que la relation « au moins aussi contraint » capture bien les bonnes propriétés des réseaux de contraintes mis en relation.

Propriété 4.17 *Soit $\mathcal{P}$ un RCP et $\mathcal{P}_R$ son problème réifié. Soit $\mathcal{P}' = \langle \mathcal{S}', \mathcal{X}', \mathcal{I}', \mathcal{C}' \rangle$ le problème $\mathcal{P}$ obtenu après l'application d'un nombre arbitraire d'opérations élémentaires*

de CAB avec C∅I. Soit $\mathcal{P}'_R = \langle \mathcal{X}'_R, \mathcal{I}'_R, \mathcal{C}'_R \rangle$ le problème $\mathcal{P}_R$ obtenu après un nombre arbitraire d'opérations de 2B-cohérence.

Si $\mathcal{P}_R$ est au moins aussi contraint que $\mathcal{P}'_R$ et si $\mathcal{P}'_R$ est manifestement incohérent, alors $\mathcal{P}_R$ l'est aussi.

Preuve 4.18 $\mathcal{P}'_R$ peut être manifestement incohérent car :

- I'_{iR} est vide. Dans ce cas, et grâce à la ligne 1 de la définition de « au moins aussi contraint », alors le domaine de x'_i est aussi vide, et donc $\mathcal{P}'$ est aussi manifestement incohérent.
- I'^{C}_{iR}, le domaine de la variable de coût x'^{C}_{iR} de c'_i est vide. Examinons alors la cause de la suppression de la dernière valeur de I'^{C}_{iR}. Cette variable est utilisée dans deux contraintes, c'_{iR} et c'^{C}_{R}, et cela signifie donc que la dernière valeur de x'^{C}_{iR} n'avait pas de support par rapport à l'une de ces contraintes.
  - Si elle n'avait pas de support par rapport à c'_{iR}, cela signifie que I'_{iR} était aussi vide, et l'on retombe sur le point précédent.
  - Si elle n'avait pas de support par rapport à c'^{C}_{R}, alors :

$$
\begin{aligned}
& bi'^{C}_{iR} + \sum_{x_j \in \mathcal{X} - \{i\}} bi'^{C}_{jR} + \sum_{c_{jl} \in \mathcal{C}} bi'^{C}_{jlR} \geq \top \\
\Rightarrow & \sum_{x_j \in \mathcal{X}} bi'^{C}_{jR} + \sum_{c_{jl} \in \mathcal{C}} bi'^{C}_{jlR} \geq \top \\
\Rightarrow & \sum_{x_j \in \mathcal{X}} \Delta'(x'_j) + \sum_{c_{jl} \in \mathcal{C}} \Delta'(x'_j, x'_l) \geq \top \\
\Rightarrow & c'_{\varnothing} \geq \top
\end{aligned}
$$

    Et donc $\mathcal{P}$ est aussi manifestement incohérent.
  - I'^{C}_{ijR} est vide. Dans ce cas, la preuve est similaire à la précédente.

Donc, si $\mathcal{P}'_R$ est manifestement incohérent, alors $\mathcal{P}'$ l'est aussi. Ceci prouve la proposition. □

Le lemme suivant est nécessaire à la correction de la preuve par induction. Son sens est décrit dans la figure 4.14.

Lemme 4.1 *Soit $\mathcal{P}$ un RCP et $\mathcal{P}_R$ son problème réifié. Soit $\mathcal{P}'$ le problème $\mathcal{P}$ obtenu après l'application d'un nombre arbitraire d'opérations élémentaires de CAB avec C∅I. Soit $\mathcal{P}''$ le problème $\mathcal{P}'$ obtenu après exactement une opération de CAB avec C∅I. Soit $\mathcal{P}'_R$ le problème $\mathcal{P}_R$ obtenu après un nombre arbitraire d'opérations de 2B-cohérence.*

Si $\mathcal{P}'$ est au moins aussi contraint que $\mathcal{P}'_R$, alors $\mathcal{P}''$ l'est aussi.

Preuve 4.19 Pour prouver ce lemme, nous passerons en revue toutes les opérations élémentaires de CAB avec C∅I qui transforment $\mathcal{P}'$ en $\mathcal{P}''$. On verra qu'aucune opération ne brise la propriété « au moins aussi contraint », et donc que $\mathcal{P}''$ est au moins aussi contraint que $\mathcal{P}'_R$.

Les opérations élémentaires sont :

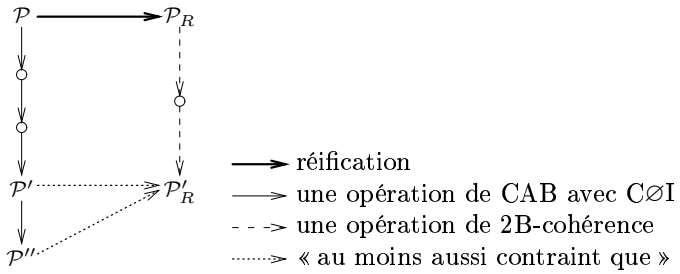


FIG. 4.14 – Soit $\mathcal{P}$ un RCP et $\mathcal{P}_R$ son problème réifié. Soit $\mathcal{P}'$ le problème $\mathcal{P}$ obtenu après l'application d'un nombre arbitraire d'opérations élémentaires de CAB avec C∅I. Soit $\mathcal{P}''$ le problème $\mathcal{P}'$ obtenu après exactement une opération de CAB avec C∅I. Soit $\mathcal{P}'_R$ le problème $\mathcal{P}_R$ obtenu après un nombre arbitraire d'opérations de 2B-cohérence. Si $\mathcal{P}'$ est au moins aussi contraint que $\mathcal{P}'_R$, alors $\mathcal{P}''$ l'est aussi.

- la suppression d'une valeur du domaine de x'_i. Dans ce cas, $I''_i \subsetneq I'_i \subseteq I'_{iR}$. La propriété 1 est donc toujours vraie, et les autres ne sont pas affectées.
- la projection. Cette opération ne modifie aucune valeur utilisée dans la définition de la relation « au moins aussi contraint ».
- projection unaire de x'_i. Dans ce cas, $\Delta''(x''_i) > \Delta'(x'_i) \geq bi'^{C}_{iR}$, et donc toutes les assertions restent vraies.
- projection binaire de c'_{ij}. Ce point est similaire au précédent.

Le lemme est donc correct. □

Ayant maintenant défini toutes les notions utiles, nous allons pouvoir prouver dans le lemme suivant la preuve d'induction. La figure 4.15 décrit le cheminement de la preuve du lemme.

Lemme 4.2 *Soit $\mathcal{P}$ un RCP et $\mathcal{P}_R$ son problème réifié. Soit $\mathcal{P}'$ le problème $\mathcal{P}$ obtenu après l'application d'un nombre arbitraire d'opérations élémentaires de CAB avec C∅I. Soit $\mathcal{P}'_R$ le problème $\mathcal{P}_R$ obtenu après un nombre arbitraire d'opérations de 2B-cohérence. Soit $\mathcal{P}''_R$ le problème $\mathcal{P}'_R$ obtenu après exactement une opération de 2B-cohérence.*

Si $\mathcal{P}'$ est au moins aussi contraint que $\mathcal{P}'_R$, alors toute série d'opérations CAB avec C∅I transformera $\mathcal{P}'$ en un problème $\mathcal{P}''$ qui est au moins aussi contraint que $\mathcal{P}''_R$, au bout d'un nombre d'opérations suffisamment grand.

Preuve 4.20 Nous allons ici considérer chaque suppression de domaine possible amenant ${\mathcal{P}'}_R$ à ${\mathcal{P}''}_R$. Nous prouverons ensuite que, pour chaque suppression, toute suite d'opérations CAB avec C∅I transformera, à un moment donné, $\mathcal{P}'$ en un problème $\mathcal{P}''$ équivalent, au moins aussi contraint que ${\mathcal{P}''}_R$. Les cas possibles sont les suivants.

- bi'^{C}_{iR} augmente parce qu'il n'a pas de support par rapport à c'_{iR}. Dans ce cas, l'assertion 2 de la définition 4.17 peut ne plus être valide. Si c'est le cas, alors cela

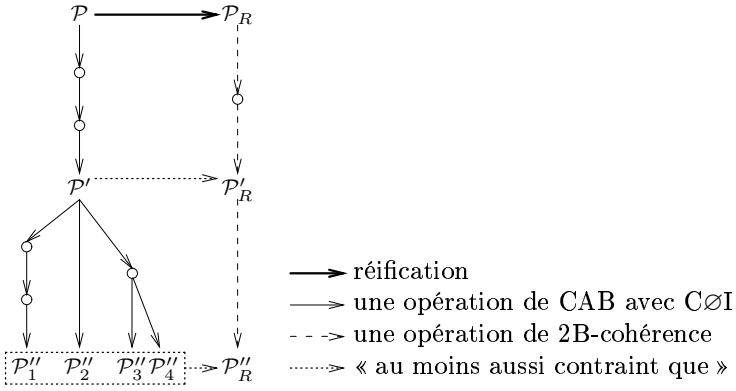


FIG. 4.15 – $\mathcal{P}$ est un RCP, et $\mathcal{P}_R$ son RC réifié. Le RCP $\mathcal{P}'$ est obtenu par un nombre arbitraire d'opérations de CAB avec C∅I. ${\mathcal{P}'}_R$ est obtenu par un nombre arbitraire d'opérations de 2B-cohérence. On suppose que $\mathcal{P}'$ est au moins aussi contraint que ${\mathcal{P}'}_R$. ${\mathcal{P}''}_R$ est obtenu par une opération de 2B-cohérence. $\mathcal{P}''_1$, $\mathcal{P}''_2$, $\mathcal{P}''_3$ et $\mathcal{P}''_4$ sont obtenus par un nombre suffisamment grand d'opérations de CAB avec C∅I (il existe en effet plusieurs manières d'établir CAB avec C∅I). $\mathcal{P}''_1$, $\mathcal{P}''_2$, $\mathcal{P}''_3$ et $\mathcal{P}''_4$ sont tous au moins aussi contraints que $\mathcal{P}''_R$.

signifierait qu'avant la suppression, $bi'^C_{iR} = \Delta'(x'_i)$, et donc, à ce moment :

$$
\begin{aligned}
&\nexists v'_{iR} \in I'_{iR}, (v'_{iR}, bi'_{iR}) \in {c_i}'_R \\
\Rightarrow &\nexists v'_i \in I'_i, (v'_i, bi'_{iR}) \in {c_i}'_R && (\text{car } I'_i \subseteq I'_{iR}) \\
\Rightarrow &\forall v'_i \in I'_i, {\mathbf{c_i}}'(v'_i) > bi'_{iR} \\
\Rightarrow &\forall v'_i \in I'_i, {\mathbf{c_i}}'(v'_i) > \Delta'(x'_i) \\
\Rightarrow &\forall v'_i \in I'_i, c'_i(v'_i) > 0 && (\text{car } c'_i(v'_i) \geq {\mathbf{c_i}}'(v'_i) \ominus \Delta'(x'_i))
\end{aligned}
$$

La dernière inégalité entraîne qu'à un moment donné CAB détectera que c'_i n'a pas de support unaire, et augmentera donc $\Delta'(x'_i)$, qui deviendra égal à ${bi''}_{i\,R}$. L'assertion 2 sera donc valide à ce moment.

- bs'^C_{iR} diminue car il n'a pas de support par rapport à c'_{iR}. Aucune assertion n'est alors violée. La propriété est toujours vraie.
- bi'_{iR} est supprimée parce qu'elle n'a pas de support par rapport à c'_{iR} et que $bi'_{iR} = bi'_i$. La propriété 1 est alors violée de la relation « est au moins aussi contraint que ». Si c'est le cas, alors cela signifie que la valeur support de bi'_{iR} dans le domaine de x'^C_{iR} a été éliminée par la contrainte c'^C_R. Si la valeur support

était d, alors :

$$
\begin{aligned}
&(\mathbf{c_i}(bi'_{iR}) = d) \wedge (d + \sum_{x_j \in \mathcal{X}-\{i\}} {bi'_{jR}}^{C} + \sum_{c_{jl} \in \mathcal{C}} {bi'_{jl}}_{R}^{C} \geq \top) \\
\Rightarrow &(\mathbf{c_i}(bi'_i) = d) \wedge (d + \sum_{x_j \in \mathcal{X}-\{i\}} \Delta'(x'_j) + \sum_{c_{jl} \in \mathcal{C}} \Delta'(x'_j, x'_l) \geq \top) \\
\Rightarrow &(\mathbf{c'_i}(bi'_i) = d) \wedge (d - \Delta'(x'_i) + \sum_{x_j \in \mathcal{X}} \Delta'(x'_j) + \sum_{c_{jl} \in \mathcal{C}} \Delta'(x'_j, x'_l) \geq \top) \\
\Rightarrow &(\mathbf{c_i}(bi'_i) = d) \wedge (d - \Delta'(x'_i) + c_\varnothing \geq \top) \\
\Rightarrow &\mathbf{c_i}(bi'_i) \geq \top - c_\varnothing + \Delta'(x'_i) \\
\Rightarrow &c'_i(bi'_i) \geq \top - c_\varnothing
\end{aligned}
$$

Et donc, à un moment donné, CAB détectera que la valeur est interdite, et la propriété 1 ne sera plus violée.

- bi'_{iR} est supprimée parce qu'elle n'a plus de support par rapport à la deuxième variable de $c'_{ij\,R}$. Cela peut violer la propriété 1 de la définition si $bi'_{iR} = bi'_i$. Mais puisque $c_{ij\,R}$ a été défini par $\{(v_i, v_j, \mathbf{c_{ij}}(v_i, v_j)) | v_i \in I_i, v_j \in I_j\}$, cela signifierait que $I'_{j\,R}$ est vide. Comme $I'_j \subseteq I'_{j\,R}$, alors I'_j est aussi vide, et dans ce cas, les deux problèmes sont manifestement incohérents.
- Tous les autres cas peuvent être facilement déduits des cas évoqués.

Ceci prouve le lemme. □

Voici maintenant la dernière étape.

Théorème 4.1 *CAB avec C∅I est strictement plus fort que 2B-cohérence.*

Preuve 4.21 Il faut tout d'abord prouver que pour tout RCP $\mathcal{P}$ et son RC reifié $\mathcal{P}_R$, pour toute suite d'opérations élémentaires de 2B-cohérence transformant $\mathcal{P}_R$ en ${\mathcal{P}'}_R$, toute suite suffisamment longue d'opérations de CAB avec C∅I transformera $\mathcal{P}$ en un problème $\mathcal{P}'$ qui est au moins aussi contraint que ${\mathcal{P}'}_R$.

Cela se prouve par induction, grâce au lemme 4.2 sur la longueur de la suite d'opérations 2B-cohérence. Le cas de base vient du fait que, par définition, $\mathcal{P}$ est au moins aussi contraint que $\mathcal{P}_R$.

L'induction entraîne que si une application de 2B-cohérence transforme $\mathcal{P}_R$ en un problème manifestement incohérent, alors CAB avec C∅I transformera $\mathcal{P}$ en un problème aussi manifestement incohérent. Donc, CAB avec C∅I est plus fort que 2B-cohérence.

Enfin, l'exemple 4.11 montre un cas où $\mathcal{P}$ est transformé en un problème manifestement incohérent par CAB, alors que 2B-cohérence ne peut pas transformer $\mathcal{P}_R$ en un problème manifestement incohérent. CAB avec C∅I est donc strictement plus fort que 2B-cohérence. □

Afin de compléter la comparaison entre 2B-cohérence et CAB, il faut fournir une complexité temporelle de l'établissement de la 2B-cohérence sur un RCP réifié. Nous

avons tenté de fournir un majorant de bonne qualité de cette complexité, sans pour autant avoir prouvé son optimalité. Nous présenterons un algorithme relativement optimisé établissant la 2B-cohérence. Il s'appuie sur AC2001/3.1 [BRYZ05] (ou, plus spécifiquement, GAC2001/3.1 puisque nous travaillons aussi sur des contraintes ternaires), et sur l'utilisation de supports. Nous avons aussi tenté d'exploiter autant que possible la sémantique des contraintes pour diminuer les complexités temporelle et spatiale.

Toutefois, deux détails diffèrent d'avec GAC2001/3.1. Tout d'abord, les éléments empilés dans Q sont des triplets variable / borne modifiée / contrainte, au lieu de la paire variable / contrainte. Connaître la borne responsable de la révision d'une variable nous aidera à optimiser les algorithmes. D'autre part, par continuité avec les algorithmes présentés dans les chapitres précédents, le triplet variable / borne modifiée / contrainte (x, v, c) indique que la borne v de la variable x a été révisée par la contrainte c, et pas que la borne v doit être révisée par c, comme c'est le cas pour GAC2001/3.1.

Le corps de l'algorithme, décrit dans 2B-cohérence, est le suivant. On commence par sortir une valeur $(x \leftarrow v)$ de Q (ligne **52**). Cette valeur a un support pour la contrainte c qui a mis à jour la borne, mais aucun support n'a jamais été recherché sur les autres contraintes (ce qui constitue une différence par rapport à la cohérence d'arc). On révise donc la borne de la variable par rapport aux autres contraintes (ligne **53**). Ensuite, on vérifie que les bornes des variables voisines ont bien un support par rapport à la variable x (lignes **54** et **55**).

Les procédures de révision des bornes sont ensuite décrites de Révision1 à Révision9 (Révision3, Révision5, Révision7, Révision9, qui s'appliquent aux bornes supérieures n'ont pas été écrits car ils peuvent directement être inférés à partir de Révision2, Révision4, Révision6, Révision8 respectivement, qui s'appliquent aux bornes inférieures).

Révision1 révise toutes les bornes supérieures des variables de coût unaires et binaires par rapport à la contrainte globale de coût c_R^C. Elle n'est appelée que lorsqu'une borne inférieure des variables de coût a augmenté. Révision2 révise la borne inférieure d'une variable qui n'est pas une variable de coût par rapport à une fonction de coût unaire réifiée. Révision3 procède de même pour la borne supérieure. Révision4 est l'équivalent de Révision2 pour les fonctions de coût binaires réifiées. Pour améliorer l'algorithme, on a fait appel à une structure de données, $supportBi(x_{iR}, c_{ij_R})$ (pour « support borne inférieure »), qui note la valeur de x_{j_R} telle qu'il existe un coût e tel que $(bi_{iR}, supportBi(x_{iR}, c_{ij_R}), e)$ est une affectation valide pour c_{ij_R}. Autrement dit, $supportBi(x_{iR}, c_{ij_R})$ est un support de bi_{iR} par rapport à la variable I_{j_R} et la contrainte c_{ij_R}. Révision5 traite les bornes supérieures. Révision6 révise la borne inférieure d'une variable de coût par rapport à une fonction de coût unaire réifiée. On utilise ici aussi une structure de donnée, $supportBi(x_{iR}^C)$, qui stocke le support de bi_{iR}^C par rapport à c_{iR}. Révision7 s'occupe de la borne supérieure. Révision8 est l'équivalent de Révision6 pour les fonctions de coût réifiées binaires. Révision8 utilise la structure de données $supportBi(x_{ij_R}^C, x_{iR})$. Cette structure stocke le support de $x_{ij_R}^C$ par rapport à la variable x_{iR} et la fonction de coût c_{ij_R}. Révision9 s'occupe des bornes supérieures.

Nous ne prouverons pas la correction de l'algorithme, supposé correct, mais nous donnerons ses complexités temporelle et spatiale.

Procédure 33 – 2B-cohérence

pour chaque $x_{iR} \in \mathcal{X}$ **faire**
  $supportBi(x_{iR}^C) \leftarrow bi_{iR}$;
  $supportBs(x_{iR}^C) \leftarrow bs_{iR}$;
  pour chaque $x_{jR} \in V(x_{iR})$ **faire**
    $supportBi(x_{iR}, x_{jR}) \leftarrow bi_{jR}$;
    $supportBi(x_{ijR}^C, x_{iR}) \leftarrow bi_{iR}$;
    $supportBs(x_{iR}, x_{jR}) \leftarrow bs_{jR}$;
    $supportBs(x_{ijR}^C, x_{iR}) \leftarrow bs_{iR}$;
$Q \leftarrow \{(x, v, c) \in \mathcal{X} \times \{bi, bs\} \times \mathcal{C} | x \in var(c)\}$;
51 **tant que** $(Q \neq \varnothing)$ **faire**
52 $(x, v, c) \leftarrow$ Extraire(Q) ;
  pour chaque $c' \in \{c' \in \mathcal{C} \setminus \{c\} | x \in var(c')\}$ **faire**
53 Révise(x, v, c') ;
  pour chaque $x' \in var(c) \setminus \{x\}$ **faire**
54 Révise(x', bi, c) ;
55 Révise(x', bs, c) ;

Procédure 34 – Révision1$(x_R^C \in \mathcal{C}_R^C, v \in \{bi\}, c \in \{c_{iR}^C\})$

$sommeMin \leftarrow \sum_{x_R^C \in \mathcal{C}_R^C} bi_R^C$;
pour chaque $x_{iR}^C \in \mathcal{C}1_R^C$ **faire**
  si $(bs_{iR}^C \geq \top - (sommeMin - bi_{iR}^C))$ **alors**
    $bs_{iR}^C \leftarrow \top - (sommeMin - bi_{iR}^C) - 1$;
    $supportBs(x_{iR}^C) \leftarrow bi_{iR}$;
    $Q \leftarrow Q \cup \{(x_{iR}^C, bs, c_R^C)\}$;
pour chaque $x_{ijR}^C \in \mathcal{C}2_R^C$ **faire**
  si $(bs_{ijR}^C \geq \top - (sommeMin - bi_{ijR}^C))$ **alors**
    $bs_{ijR}^C \leftarrow \top - (sommeMin - bi_{ijR}^C) - 1$;
    $supportBs(x_{ijR}^C, x_{iR}) \leftarrow bi_{iR}$;
    $supportBs(x_{ijR}^C, x_{jR}) \leftarrow bi_{jR}$;
    $Q \leftarrow Q \cup \{(x_{ijR}^C, bs, c_R^C)\}$;

Procédure 35 – Révision2($x_{iR} \in \mathcal{X}_R, v \in \{bi\}, c \in \{c_{iR}\}$)

tant que $(bi_{iR} \preceq bs_{iR})$ **faire**
56 **si** $(bi_{iR}^C \leq \mathbf{c_i}(bi_{iR}) \leq bs_{iR}^C))$ **alors**
retourner ;
$bi_{iR} \leftarrow succ(bi_{iR})$;
57 **pour chaque** $x_{kR} \in V(x_{iR})$ **faire**
$supportBi(x_{iR}, x_{kR}) \leftarrow bi_{kR}$;
$Q \leftarrow Q \cup \{(x_{iR}, bi, c_{iR})\}$;
lever incohérence ;

Procédure 36 – Révision4($x_{iR} \in \mathcal{X}_R, v \in \{bi\}, c_{ij_R} \in \{c' \in \mathcal{C}_{2R} | x \in var(c')\}$)

tant que $(bi_{iR} \preceq bs_{iR})$ **faire**
tant que $(supportBi(x_{iR}, x_{j_R}) \preceq bs_{j_R})$ **faire**
si $(bi_{ij_R}^C \leq \mathbf{c_{ij}}(bi_{iR}, supportBi(x_{iR}, x_{j_R})) \leq bs_{ij_R}^C)$ **alors**
retourner ;
$supportBi(x_{iR}, x_{j_R}) \leftarrow succ(supportBi(x_{iR}, x_{j_R}))$;
$bi_{iR} \leftarrow succ(bi_{iR})$;
58 **pour chaque** $x_{kR} \in V(x_{iR})$ **faire**
$supportBi(x_{iR}, x_{kR}) \leftarrow bi_{kR}$;
$Q \leftarrow Q \cup \{(x_{iR}, bi, c_{ij_R})\}$;
lever incohérence ;

Procédure 37 – Révision6($x_{iR}^C \in \mathcal{C}1_R^C, v \in \{bi\}, c \in \{c_{iR}\}$)

tant que $(bi_{iR}^C \leq bs_{iR}^C)$ **faire**
tant que $(supportBi(x_{iR}^C) \preceq bs_{iR})$ **faire**
59 **si** $(\mathbf{c_i}(supportBi(x_{iR}^C)) = bi_{iR}^C)$ **alors**
retourner ;
$supportBi(x_{iR}^C) \leftarrow succ(supportBi(x_{iR}^C))$;
$bi_{iR}^C \leftarrow bi_{iR}^C + 1$;
$supportBi(x_{iR}^C) \leftarrow bi_{iR}$;
$Q \leftarrow Q \cup \{(x_{iR}^C, bi, c_{iR})\}$;
lever incohérence ;

Procédure 38 – Révision8($x_{ij_R}^C \in \mathcal{C}_{2R}^C, v \in \{bi\}, c_{ij_R} \in \{c' \in \mathcal{C}_{2R} | x \in var(c')\}$)

tant que $(bi_{ij_R}^C \leq bs_{ij_R}^C)$ **faire**
  tant que $(supportBi(x_{ij_R}^C, x_{iR}) \preceq bs_{iR})$ **faire**
    tant que $(supportBi(x_{ij_R}^C, x_{j_R}) \preceq bs_{j_R})$ **faire**
      si $(\mathbf{c_{ij}}(supportBi(x_{ij_R}^C, x_{iR}), supportBi(x_{ij_R}^C, x_{j_R})) = bi_{ij_R}^C)$ **alors**
        retourner ;
      $supportBi(x_{ij_R}^C, x_{j_R}) \leftarrow succ(supportBi(x_{ij_R}^C, x_{j_R}))$;
    $supportBi(x_{ij_R}^C, x_{j_R}) \leftarrow bi_{j_R}$;
    $supportBi(x_{ij_R}^C, x_{iR}) \leftarrow succ(supportBi(x_{ij_R}^C, x_{iR}))$;
  $bi_{ij_R}^C \leftarrow bi_{ij_R}^C + 1$;
  $supportBi(x_{ij_R}^C, x_{iR}) \leftarrow bi_{iR}$;
  $Q \leftarrow Q \cup \{(x_{ij_R}^C, bi, c_{ij_R})\}$;
lever incohérence ;

Propriété 4.18 *L'algorithme* 2B-cohérence *appliqué à un problème réifié a une complexité temporelle de* $\mathcal{O}((e^2 + ed^2) \times \min\{\top, end\})$ *et une complexité spatiale de* $\mathcal{O}(e)$.

Preuve 4.22 Rappelons ici que n, e et d indiquent ici respectivement le nombre de variables, le nombre de fonctions de coût et la taille du plus grand domaine dans le réseau de contrainte *pondéré*.

Déterminons tout d'abord le nombre d'itérations effectuées par la boucle de la ligne **51**. Une valeur est empilée dans Q à chaque fois qu'elle est éliminée. Le problème réifié contient nd valeurs venant directement du problème d'origine, plus $(n+e)\top$ valeurs venant des variables de coût. La boucle de la ligne **51** itère donc $\mathcal{O}(nd + e\top)$ fois.

Trouvons maintenant la complexité cumulée de chaque procédure de révision. Révision1 peut être exécutée en temps $\mathcal{O}(n+e)$. Elle est de plus appelée à chaque fois qu'une variable de coût perd une valeur. Sa complexité cumulée est donc $\mathcal{O}(e^2\top)$. Révision2 ne passe la ligne **56** que lorsque la valeur d'une variable qui n'est pas une variable de coût est supprimée, soit au maximum $\mathcal{O}(nd)$ fois. De plus, la boucle de la ligne **57** n'itère qu'une fois qu'une valeur a été retirée du domaine, ce qui amène à une complexité de $\mathcal{O}(ed)$. Étant donné que cette procédure peut être appelée soit après la suppression de la valeur d'une variable qui n'est pas de coût, soit après la suppression d'une valeur d'une variable de coût unaire, la complexité de cette procédure est donc $\mathcal{O}(ed + n\top + nd) = \mathcal{O}(ed + n\top)$. Concernant la procédure Révision4, le mécanisme de mémorisation fait que, si l'on considère une contrainte binaire c_{ij_R}, et si un couple $(v_{iR}, v_{j_R}) \in I_{iR} \times I_{j_R}$ a été prouvé non valide, alors il ne sera jamais examiné une seconde fois. En comptant la boucle de la ligne **58**, cela porte donc la complexité de cette procédure à $\mathcal{O}(nd + e\top + ed^2 + ed) = \mathcal{O}(ed^2 + e\top)$, car cette fonction peut être appelée après la suppression d'une valeur d'une variable qui n'est pas une variable de coût ($\mathcal{O}(nd)$ fois) ou après la suppression d'une valeur d'une variable de coût binaire ($\mathcal{O}(e\top)$ fois). Comme Révision2, Révision6 peut être appelée $\mathcal{O}(nd + n\top)$ fois. Le mécanisme de

mémorisation fait que la complexité temporelle sous la condition de la ligne **59** a une complexité temporelle de $\mathcal{O}(nd\top)$. La complexité de la procédure est donc de $\mathcal{O}(nd\top)$. De même, la complexité de la procédure Révision8 est $\mathcal{O}(nd + e\top + ed^2\top) = \mathcal{O}(ed^2\top)$.

Après sommation des diverses complexités temporelles, on conclut que la complexité temporelle totale s'élève à $\mathcal{O}(e^2\top + ed^2\top)$.

On peut encore affiner ce résultat. On peut tout d'abord observer que la borne inférieure d'une variable de coût n'est mise à jour qu'au début de l'établissement de la 2B-cohérence et à la suite de la suppression d'une valeur d'une variable qui n'est pas une variable de coût, ce qui se produit dans les procédures Révision6 et Révision8. Cela ne se produit donc pas plus de $\mathcal{O}(nd)$ fois. Les bornes supérieures des variables de coûts sont mises à jour de la même manière dans les procédures Révision7 et Révision9, ainsi qu'à la suite d'une augmentation de la borne inférieure d'une variable de coût (procédure Révision1). Les bornes supérieures des fonctions de coûts sont donc modifiées au plus $\mathcal{O}(nd + (n + e)nd) = \mathcal{O}(end)$ fois. Les variables de coût ne sont donc mises à jour qu'au maximum $\mathcal{O}(end)$ fois. Donc, dans les calculs de complexité précédents, on peut remplacer le nombre de fois qu'une variable de coût est mise à jour (*a priori* $\mathcal{O}(\top)$) par $\mathcal{O}(\min\{\top, end\})$. Ceci nous donne une complexité de $\mathcal{O}((e^2 + ed^2) \times \min\{\top, end\})$.

Concernant la complexité spatiale, les bornes des intervalles sont au nombre de $2n$ pour les variables réifiées et $2n + 2e$ pour les variables de coût. De plus, le mécanisme de mémorisation ajoute $4n + 4e$ nouvelles structures de données. La complexité spatiale totale est donc de $\mathcal{O}(e)$. □

La complexité de l'algorithme établissant la 2B-cohérence est donc le minimum entre $\mathcal{O}((e^2 + ed^2) \times \top)$ et $\mathcal{O}((e^2 + ed^2) \times end)$. Dans le premier cas, nous avons une complexité qui est linéaire par rapport à $\top$. Cette complexité est problématique car elle fait perdre le caractère polynomial de l'algorithme de filtrage par rapport à la taille du problème, selon les critères de définition de la complexité [Pap94]. Dans le deuxième cas, nous avons un algorithme cubique par rapport à d. Cette complexité par rapport à d est la même que celle de CAB avec C∅I, qui est de $\mathcal{O}(ed^3 + \min\{\top, nd\} \times n)$. Les deux algorithmes ont donc une complexité temporelle comparable.

Afin de comparer totalement les deux propriétés de cohérence locale, il faudrait aussi implanter le mécanisme de réification des coûts ainsi que la 2B-cohérence et se livrer à des expérimentations. Néanmoins, pour des raisons de temps, nous n'avons pas poursuivi la comparaison jusque là.

4.2 Conclusion

Nous avons dans cette partie principalement proposé deux propriétés de cohérence locales : la cohérence d'arc existentielle et la cohérence d'arc aux bornes. Concernant la première, nous esperons qu'elle constituera une cohérence locale de choix pour les réseaux de contraintes pondérées « classiques », c'est-à-dire contenant des domaines dont la taille ne dépasse pas les quelques dizaines de valeurs. La comparaison sur un banc de test relativement large avec la cohérence d'arc complète et directionnelle, considérée comme une des cohérences locales les plus efficaces, semble le montrer. La cohé-

rence d'arc existentielle possède notamment l'avantage d'être facilement maintenue dans l'arbre de recherche des solutions, ce qui la rend compatible avec des cohérences plus fortes, mais plus lourdes, comme la cohérence d'arc optimale [CdGS07].

Nous avons aussi présenté la cohérence d'arc aux bornes, particulièrement adaptée aux problèmes dont la taille des domaines est grande. Nous avons envisagé plusieurs spécialisations de cette cohérence locale. Nous avons présenté un moyen de renforcer cette cohérence par une autre, la cohérence $\varnothing$-inverse. Nous avons considéré les gains de temps que l'on peut espérer lorsque la sémantique des fonctions de coût est connue et exploitable, en vue notamment d'appliquer les résultats démontrés au cadre du problème de la localisation d'ARN. Nous avons enfin confronté la cohérence d'arc aux bornes avec la 2B-cohérence, dans une modélisation des réseaux de contraintes pondérées utilisant des contraintes classiques. La comparaison a montré, au moins théoriquement, que la cohérence aux bornes était strictement plus forte et que les complexités temporelles étaient comparables.

Afin de confirmer la pertinence de la cohérence d'arc aux bornes, il faudrait comparer cette propriété de cohérence locale à la propriété habituellement utilisée : la cohérence d'arc. C'est ce que nous ferons au chapitre sept.

Chapitre 5

Algorithmes

5.1 Introduction

Afin de résoudre la question qui sous-tend notre recherche — trouver les membres d'une famille d'ARN donnée — nous avons supposé dans les chapitres précédents que nous connaissions les éléments structurels caractéristiques de cette famille. Nous avons aussi supposé que ces éléments étaient suffisament discriminants pour caractériser les ARN de la famille. Dans le chapitre 3, nous avons dressé la liste de ces éléments structurels (en l'occurrence, des mots, plusieurs types d'hélice, des répétitions, des compositions en (G+C)%, des interactions non-canoniques et des espaceurs), et nous avons décrit la modélisation que nous allions utiliser. Cette modélisation fait intervenir les réseaux de contraintes pondérées, que nous avons décrits dans le chapitre 3. Nous avons aussi montré comment les fonctions de coûts des réseaux de contraintes pondérées étaient à même de modéliser simplement les éléments de structures d'un ARN non-codant. Nous avons enfin donné une méthode de filtrage, appelée cohérence d'arc aux bornes, qui s'applique sur ces fonctions de coût et permet de rechercher efficacement les solutions du problème. Le présent chapitre s'ouvre donc sur l'implantation de la cohérence d'arc aux bornes sur chaque fonction de coût. Cette partie, très algorithmique, est spécialement tournée vers l'utilisation de méthodes de *pattern-matching*.

Les réseaux de contraintes pondérées permettant de dissocier efficacement algorithmes de *pattern-matching* utilisés pour le filtrage et recherche de solutions, nous avons aussi pu implanter plusieurs optimisations appliquées à cette recherche de solutions. Ces algorithmes seront donnés par la suite.

Nous présentons ensuite une caractéristique intéressante de notre prototype : la sélection de solutions. En effet, notre approche peut être amenée à prédire quelques solutions redondantes. Nous expliquerons comment ces solutions sont détectées et comment l'examen des solutions trouvées peut même accélérer la recherche. Nous estimerons enfin la complexité temporelle de notre outil.

5.2 Algorithmes sur les fonctions de coût

5.2.1 Introduction

Nous allons dans la suite donner le détail de l'implantation de chaque fonction de coût dans notre réseau de contraintes pondérées. Le lecteur pourra se référer au paragraphe 3.4.2.1 pour plus de détails sur ce formalisme. Lors de la recherche de solutions, nous maintiendrons une propriété de cohérence locale appelée cohérence d'arc aux bornes (CAB, cf. définition 4.11), éventuellement avec la cohérence ∅-inverse (C∅I, cf. définition 4.12). Nous avons vu dans le chapitre précédent que CAB avec C∅I était établie par une série de fonctions qui sont notamment ProjectionBorneInférieure et ProjectionBorneSupérieure (et ProjectionBinaire, si l'on souhaite appliquer C∅I). Ces fonctions sont essentiellement basées sur la recherche du minimum des fonctions de coût du réseau de contraintes pondérées. Par exemple, la fonction ProjectionBorneInférieure recherche le minimum d'une fonction de coût *sachant que le domaine d'une variable a été réduit à sa borne inférieure*. La première partie de ce chapitre se propose donc de détailler la recherche du minimum des fonctions de coût, sachant que le domaine d'une variable, par exemple x_i, a été réduit à sa borne inférieure. Les fonctions de coût étant relativement symétriques, on pourra facilement inférer comment trouver leur minimum, sachant que le domaine d'une variable a été réduit à la borne supérieure, ou que le domaine d'une autre variable, comme x_j, a été réduit.

Il n'existe qu'une seule fonction de coût pour laquelle nous avons décidé d'appliquer C∅I : il s'agit de l'espaceur. En effet, le minimum de cette fonction pouvant se trouver très rapidement, on a tout intérêt à appliquer la propriété de cohérence locale la plus forte possible sur une telle fonction de coût. Dans ce cas, nous détaillerons comment trouver le minimum de la fonction sans supposer qu'un domaine a été réduit.

Enfin, nous avons avons montré dans le paragraphe 3.4.4 que nous avions des fonctions de coût, de type dur, qui ne renvoient comme valeur que zéro ou $\top$. Ces fonctions peuvent alors être considérées comme des contraintes dures, et les mécanismes de maintien de la propriété de cohérence locale, en l'occurrence la 2B-cohérence (cf. paragraphe 3.4), sont donc plus performants. Dans l'implantation, nous avons bien créé des mécanismes différents pour les fonctions de coût et les contraintes dures. Mais pour la clarté de l'exposé, nous traiterons ici les deux types de contraintes (classiques et pondérées) de la même manière.

À la fin de la description de l'implantation de chaque fonction, nous décrivons brièvement la manière de trouver les supports de la valeur bi_i par rapport à cette fonction (voir la définition 3.3 pour une définition du support). Nous verrons en effet dans les paragraphes suivants que garder les supports des fonctions peut notablement accéler la recherche de solutions. Toutefois, les détails des mécanismes de recherche de supports ne seront pas donnés.

5.2.2 Le mot

mot[mot, $\top_{\text{loc}}$, transformateur](x_i, x_j)

La fonction de coût de mot recherche un mot mot entre les positions x_i et x_j. Un

nombre d'erreurs maximum $\top_{\text{loc}}$ est donné. Notre but est de trouver le minimum de la fonction lorsque x_i vaut bi_i, la valeur de sa borne inférieure, et pour toute valeur de x_j, c'est-à-dire pour l'intervalle $[bi_j..bs_j]$.

La première chose que l'on peut faire est de tenter de réduire l'espace de recherche des supports. En effet, connaissant le mot mot et la position de bi_i, on peut en déduire un ensemble de positions possibles pour x_j. Dans la pratique, on peut procéder à la réduction des domaines suivante :

- $bi_j = \max\{bi_j, bi_i + (|\mathsf{mot}| - 1 - \top_{\text{loc}})\}$,
- $bs_j = \min\{bs_j, bi_i + (|\mathsf{mot}| - 1 + \top_{\text{loc}})\}$,

Cela fait, la trame de la recherche suit l'algorithme de Needleman-Wunsch, présenté dans le paragraphe 2.2.1.1. On lit tous les nucléotides entre bi_i et bs_j et l'on calcule à la volée le nombre d'erreurs minimum.

Une différence existe toutefois entre notre algorithme et celui de Needleman-Wunsch. Nous avons vu dans la spécification de la fonction de mot (cf. paragraphe 3.2.1) que les insertions et suppressions n'étaient autorisées qu'à l'intérieur du mot. En d'autres termes, aux deux extrémités du mot, on n'autorise que des substitutions. Cela a pour effet que l'on est certain que le mot que l'on recherche commence bien à la position x_i, et finit en x_j. La formule de récurrence de l'algorithme de Needleman-Wunsch est donc légèrement différente : les formules sur la première et dernière ligne ne doivent pas autoriser les insertions ou les suppressions.

Pour expliciter cette récurrence, nous utiliserons les conventions suivantes. S sera la séquence principale, $S[i]$ sera le i-ème nucléotide de S et $S[i..j]$ sera le mot composé des nucléotides $S[i], \ldots, S[j]$. La première lettre d'un mot est $S[1]$ (et non $S[0]$). Pour la clarté de l'exposé, tous les coûts d'insertion et de suppression sont de un (même si c'est modifiable par l'utilisateur). Nous utiliserons aussi une fonction c_i qui prend en paramètre deux nucléotides et renvoie un coût nul si les deux nucléotides sont égaux et un dans le cas contraire.

Pour notre algorithme de programmation dynamique, nous utiliserons une matrice mat, dont le premier indice en abscisse est $bi_i - 1$ et le dernier indice est bs_j. En ordonnée, cette matrice s'étend de 1 à $|\mathsf{mot}|$. Rappelons la formule de récurrence pour l'algorithme de Needleman-Wunsch :

$$\forall a \in [bi_i..bs_j], \forall b \in [2..|\mathsf{mot}|-1], mat[a,b] = \min \left\{ \begin{array}{l} mat[a-1,b-1] + c_i(S[a], \mathsf{mot}[b]) \\ mat[a-1,b] + 1 \\ mat[a,b-1] + 1 \end{array} \right\}.$$

La formule indique que tout alignement peut être déduit d'un alignement plus petit, où l'on ajoute un nucléotide laissé non apparié de S ou de mot, ou bien un appariement entre S et mot.

Il faut maintenant donner les formules de base de la récurrence. Les deux premières sont :

$$\begin{array}{r} \forall a \in [bi_i..bs_j], mat[a,1] = c_i(S[a], \mathsf{mot}[1]); \\ \forall b \in [2..|\mathsf{mot}|], mat[bi_i - 1, b] = \top_{\text{loc}}. \end{array}$$

La première ligne indique que tout alignement composé de la première lettre de mot et d'une lettre de S ne peut être obtenu que par appariement. La seconde ligne indique que tout alignement entre une lettre de mot qui n'est pas la première lettre et une lettre de S reçoit un coût $\top_{\text{loc}}$. Il faut aussi spécifier qu'il ne peut pas y avoir d'insertion après avoir apparié la dernière lettre de mot. Cela se fait par la règle suivante :

$$\forall a \in [bi_i..bs_j], mat[a, |\mathsf{mot}|] = mat[a-1, \mathsf{mot}[|\mathsf{mot}|-1]] + c_i(S[a], \mathsf{mot}[|\mathsf{mot}|]).$$

Dans notre cas particulier, il faut aussi ajouter que les alignements ne peuvent commencer qu'à la position bi_i. On ajoute alors le cas de base :

$$\forall a \in [bi_i + 1..bs_j], mat[a, 1] = \top_{\text{loc}}.$$

Le score minimum peut alors être trouvé. Il correspond au plus petit coût calculé sur la dernière ligne (après avoir aligné toutes les lettres de mot), dont l'abscisse est entre bi_j et bs_j. Le score renvoyé par la fonction de recherche de minimum est donc :

$$score = \min_{a \in [bi_j..bs_j]} mat[a, |\mathsf{mot}|].$$

Dans l'algorithme implanté, nous avons de plus choisi d'ajouter une optimisation classique : la matrice bi-dimensionnelle de programmation dynamique mat est remplacée par un simple vecteur. La complexité spatiale de l'algorithme de recherche du minimum, donnée par le vecteur utilisé par l'algorithme de programmation dynamique, est donc de $\mathcal{O}(|\mathsf{mot}|)$. Concernant la complexité temporelle, puisque nous avons implanté l'algorithme de Needleman-Wunsch, elle est de $\mathcal{O}(D|\mathsf{mot}|)$, où D est la taille de la séquence examinée. Celle-ci est bornée par $|\mathsf{mot}| + \top_{\text{loc}}$, la complexité de la recherche de minimum est donc de $\mathcal{O}((|\mathsf{mot}| + \top_{\text{loc}}) \times |\mathsf{mot}|)$.

Nous verrons par la suite que le calcul du support de bi_i, c'est-à-dire la valeur de x_j telle que la fonction $v_j \mapsto \mathtt{mot}(bi_i, v_j)$ connaît un minimum, peut être important pour optimiser les algorithmes. Le support est ici la valeur où s'arrête l'alignement, soit la valeur $\arg\min_{a \in [bi_j..bs_j]} mat[a, |\mathsf{mot}|]$.

5.2.3 Les interactions

5.2.3.1 L'hélice

$\mathtt{hélice}[\mathsf{taille\_min}, \mathsf{taille\_max}, \mathsf{boucle\_min}, \mathsf{boucle\_max}, \mathsf{indels}, \mathsf{wobble}, \top_{\text{loc}}, \mathsf{transformateur}](x_i, x_j, x_k, x_l)$

Nous allons ici étudier le moyen de trouver rapidement le score minimum d'une fonction de coût d'hélice. Cette fonction évalue l'éventuelle hélice dont le premier brin est entre x_i et x_j et le second, entre x_k et x_l. La taille de l'hélice doit être comprise entre taille_min et taille_max, la taille de la boucle doit être comprise entre boucle_min et boucle_max.

La trame générale de la recherche est la suivante. Considérant la valeur bi_i de x_i, on peut utiliser les données de la taille d'hélice et de la taille de la boucle pour réduire les domaines d'exploration des trois autres variables. On pourra pour cela utiliser les formules données dans la figure 5.1.

$$bi_j = \max \left\{ \begin{array}{l} bi_j \\ bi_i + (\mathsf{taille\_min} - 1) \\ bi_k - (\mathsf{boucle\_max} + 1) \\ bi_l - 2(\mathsf{taille\_max} - 1) - (\mathsf{boucle\_max} + 1) \end{array} \right\}$$

$$bs_j = \min \left\{ \begin{array}{l} bs_j \\ bi_i + (\mathsf{taille\_max} - 1) \\ bs_k - (\mathsf{boucle\_min} + 1) \\ bs_l - 2(\mathsf{taille\_min} - 1) - (\mathsf{boucle\_min} + 1) \end{array} \right\}$$

$$bi_k = \max \left\{ \begin{array}{l} bi_k \\ bi_i + (\mathsf{taille\_min} - 1) + (\mathsf{boucle\_min} + 1) \\ bi_j + (\mathsf{boucle\_min} + 1) \\ bi_l - (\mathsf{taille\_max} - 1) \end{array} \right\}$$

$$bs_k = \min \left\{ \begin{array}{l} bs_k \\ bi_i + (\mathsf{taille\_max} - 1) + (\mathsf{boucle\_max} + 1) \\ bs_j + (\mathsf{boucle\_max} + 1) \\ bs_l - (\mathsf{taille\_min} - 1) \end{array} \right\}$$

$$bi_l = \max \left\{ \begin{array}{l} bi_l \\ bi_i + 2(\mathsf{taille\_min} - 1) + (\mathsf{boucle\_min} + 1) \\ bi_j + (\mathsf{taille\_min} - 1) + (\mathsf{boucle\_min} + 1) \\ bi_k + (\mathsf{taille\_min} - 1) \end{array} \right\}$$

$$bs_l = \min \left\{ \begin{array}{l} bs_l \\ bi_i + 2(\mathsf{taille\_max} - 1) + (\mathsf{boucle\_max} + 1) \\ bs_j + (\mathsf{taille\_max} - 1) + (\mathsf{boucle\_max} + 1) \\ bs_k + (\mathsf{taille\_max} - 1) \end{array} \right\}$$

FIG. 5.1 – Formules de propagation des distances d'hélice et de boucle dans la fonction de coût d'hélice.

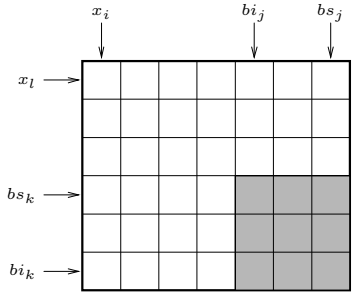


FIG. 5.2 – Une matrice de programmation dynamique calculant le score de l'alignement entre le brin situé entre x_i et bs_j et le brin situé entre bi_k et x_l. Les cases grisées contiennent les scores pour toutes les valeurs de x_j et x_k.

On itère ensuite sur les valeurs restantes de x_l. Nous avons alors le début et la fin de l'hélice. En théorie, il faudrait aussi itérer sur les valeurs des variables x_j et x_k et exécuter un algorithme de Needleman-Wunsch (cf. paragraphe 2.2.1.1) pour trouver le score minimum. Mais on peut faire mieux. On peut calculer l'alignement entre les deux régions maximales s'étendant entre bi_i et bs_j d'une part, et de x_l à bi_k d'autre part. Dans ce cas, la matrice de programmation dynamique calcule comme résultats intermédiaires tous les alignements entre x_i et les valeurs de x_j d'une part, et entre x_l et les valeurs de x_k d'autre part. C'est expliqué sur la figure 5.2.

On procède de même pour la distance de Hamming, si ce n'est qu'il n'est pas besoin de la matrice de programmation dynamique pour calculer les valeurs intermédiaires (cf. paragraphe 2.2.1.1). Finalement, le score trouvé est le plus petit score rencontré dans toutes les matrices de programmation dynamiques générées.

Pour la distance de Levenshtein, nous avons utilisé deux optimisations. Tout d'abord, on ne remplit la matrice de programmation dynamique qu'autour de la diagonale principale, comme expliqué dans le paragraphe 2.2.1.1. Une autre optimisation, spatiale cette fois-ci, est que la matrice bi-dimensionnelle de programmation dynamique est remplacée par un simple vecteur.

L'algorithme de recherche de minima a une complexité temporelle constante en la taille du texte. En voici la raison. Puisque la valeur de x_i est fixée à bi_i, la variable x_l n'a plus que $\mathcal{O}(\mathsf{long\_max} + \mathsf{boucle\_max}) = \mathcal{O}(1)$ valeurs possibles. Si l'on utilise la distance de Hamming, alors l'algorithme se termine en temps $\mathcal{O}(\mathsf{long\_max}) = \mathcal{O}(1)$. Sinon, l'algorithme de programmation dynamique utilisé est proportionnel à la longueur d'un mot et au nombre d'erreurs autorisées. La complexité spatiale est constante si l'on utilise la distance de Hamming et vaut $\mathcal{O}(\mathsf{long\_max})$ si l'on choisit la distance de Levenshtein.

Le support de bi_i par rapport à la fonction de coût d'hélice est trouvé durant la recherche de minimum. Le support par rapport à la variable x_l est donné par la valeur de l'itérateur sur les valeurs de cette variable lorsque le minimum est trouvé. Les supports par rapport à x_j et x_k sont donnés par les abscisse et ordonnée de la cellule de la matrice de programmation dynamique où se trouve le minimum.

5.2.3.2 L'hélice alternative

`hélice_alt`[taille_max, wobble, $\top_{\text{loc}}$, transformateur](x_i, x_j, x_k, x_l)
La fonction de coût d'hélice alternative évalue aussi l'éventuelle hélice entre les quatre variables utilisées par la fonction x_i, x_j, x_k et x_l. Cette fonction diffère de la précédente, dans la mesure où la taille de la boucle et la taille minimale de la longueur d'hélice ne sont pas spécifiées, et où une pénalité est donnée aux hélices dont la taille est plus petite que la taille maximale : une pénalité de i est donnée si l'hélice est inférieure de i paires de bases à la taille maximale.

L'algorithme de recherche du minimum de cette fonction est toutefois relativement différent de celui de la fonction d'hélice. Celui-ci calcule tous les scores d'hélice possibles d'un seul coup. Comme il utilise une matrice de programmation dynamique pour cela, l'algorithme attend que les tailles maximales des brins de l'hélice soient suffisamment réduites. Dans la pratique, on attendra que les valeurs $bs_j - bi_i$ et $bs_l - bi_k$ soient inférieures à une constante taille_hélice_max.

Pour cela, on utilise une matrice tri-dimensionnelle, de taille taille_hélice_max $\times$ taille_hélice_max $\times$ long_max. Le score contenu dans les cellules est le suivant : la case (i, l, n) stocke le plus petit coût parmi tous les alignements possibles dont le premier brin s'arrête à la position $bi_i + i$, et le second s'arrête à la position $bs_l - l$, sachant que l'alignement est de longueur n.

La récurrence de l'algorithme de programmation dynamique est classique. Elle utilise entre autres la fonction c_a, qui donne le score d'appariement de deux nucléotides.

$$matrice[i+1, l+1, n+1] = \min \left\{ \begin{array}{l} matrice[i, l+1, n], \\ matrice[i+1, l, n], \\ matrice[i, l, n] + c_a(S[bi_i + i], S[bs_l - l]) \end{array} \right\}$$

Un premier cas de base permet de n'accepter que les hélices commençant et finissant aux bons endroits :

$$matrice[i, l, 0] = \begin{cases} 0 & \text{si } (i = 0) \wedge (l \leq bs_l - bi_l), \\ \top_{\text{loc}} & \text{sinon.} \end{cases}$$

Les autres cas de bases obligent l'alignement à débuter au niveau zéro :

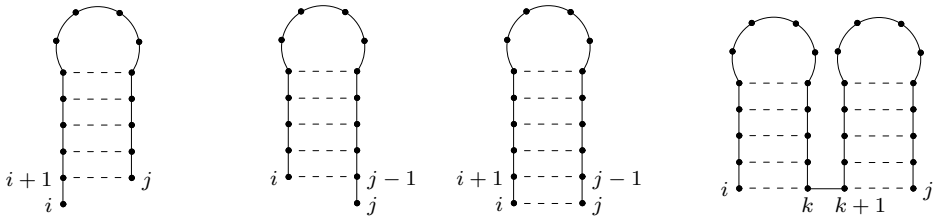
$$\forall i > 0, l > 0, n > 0, matrice[0, l, n] = matrice[i, 0, n] = \top_{\text{loc}}$$

Le coût minimum se trouve en lisant quelques cellules de la matrice. La valeur que l'on recherche est :

$$\min_{\substack{i \geq bi_l - bi_i \\ l \geq bs_l - bs_k \\ n}} \{matrice[i, l, n] + (\text{long\_max} - n)\}$$

La complexité temporelle est donnée par le temps nécessaire à remplir la matrice. Elle est de $\mathcal{O}(\text{taille\_hélice\_max}^2 \times \text{long\_max})$. La complexité spatiale est donnée par la même matrice. Elle est donc aussi de $\mathcal{O}(\text{taille\_hélice\_max}^2 \times \text{long\_max})$.

Toutefois, cette fonction de coût mériterait sans doute d'être améliorée. La technique consistant à attendre que les domaines soient rétrécis n'est pas très efficace, car elle ne



(a) premier cas (b) deuxième cas (c) troisième cas (d) quatrième cas

FIG. 5.3 – Les différents cas de l'algorithme de Nussinov.

réduit que très peu le domaine des variables. De plus, les tailles minimale et maximale de la boucle ne sont pas spécifiées. Cette information pourrait pourtant être exploitée pour propager plus efficacement la fonction de coût. L'implantation de cette fonction de coût a été un des premiers travaux de cette thèse et elle mériterait sûrement une réorganisation.

Le calcul du support de bi_i est fait en même temps que la recherche de minimum. Pour trouver le support par rapport à x_l, chaque cellule de la matrice de programmation dynamique contient non seulement le coût du meilleur alignement, mais aussi l'endroit où il se finit. Cette valeur est le support par rapport à x_l. Le support par rapport à x_j et x_k est donné par les abscisse et ordonnée de la cellule contenant le coût minimum.

5.2.3.3 Le repliement

`repliement`[taille_min, taille_max, wobble, $\top_{\text{loc}}$, transformateur](x_i, x_j)

Un troisième type d'hélice est proposé à l'utilisateur : le repliement. Cette fonction de coût évalue le repliement de la région délimitée par les deux variables de la fonction de coût. Sa procédure de recherche de minimum s'inspire directement de l'algorithme de Nussinov [CNC83]. Cet algorithme permet de prédire la structure secondaire d'une séquence en tentant de la replier sur elle-même et de maximiser les interactions. Il accepte les boucles multiples (cf. figure 1.7(d)) mais pas les pseudo-nœuds (cf. figure 1.8(a)). Intuitivement, l'algorithme de Nussinov se base sur le fait qu'un repliement entre les nucléotides $S[i]$ et $S[j]$ peut être construit à partir :

- du repliement $S[i+1..j]$ auquel on ajoute le nucléotide non apparié à gauche $S[i]$ (cf. figure 5.3(a)),
- du repliement $S[i..j-1]$ auquel on ajoute le nucléotide non apparié à droite $S[j]$, (cf. figure 5.3(b))
- du repliement $S[i+1..j-1]$ auquel on ajoute la paire de nucléotides appariés $S[i] \cdots S[j]$ (cf. figure 5.3(c)),
- de deux repliements $S[i..k]$ et $S[k+1..j]$ que l'on assemble, pour $k \in]i..j[$ (ce cas rend possible les boucles multiples, cf. figure 5.3(d)).

L'algorithme de Nussinov se présente en général sous la forme d'une matrice de programmation dynamique, dont il faut remplir le triangle supérieur droit. Chaque cellule (i, j) représente le nombre d'interactions maximales dans la sous-séquence $S[i..j]$. L'algorithme est présenté en détail dans la fonction 39.

Fonction 39 – Nussinov(S)

$matrice[0,0] \leftarrow 0$;
pour chaque $i \in [1..|S|-1]$ **faire**
  $mat[i, i-1] \leftarrow 0$;
  $mat[i,i] \leftarrow 0$;
pour chaque $d \in [1..|S|-1]$ **faire**
  pour chaque $i \in [0..|S|-1-i]$ **faire**
    $j \leftarrow i + d$;

$$mat[i,j] \leftarrow \min \left\{ \begin{array}{l} mat[i+1,j], \\ mat[i,j-1], \\ mat[i+1,j-1] + (1 - c_a(S[i], S[j])), \\ \min_{k \in]i..j[} \{mat[i,k] + mat[k+1,j]\} \end{array} \right\} ;$$

retourner $mat[0, |S-1|]$;

Pour notre algorithme, nous voulions tout d'abord un score qui soit compris comme une pénalité : un score haut doit correspondre à un repliement qui n'est pas préféré. Nous avons donc compté non pas le nombre d'appariements, mais le nombre d'insertions, de suppressions, de substitutions, et d'ouvertures de boucles multiples.

Toutefois, en utilisant cette approche, on se heurte à un problème. Notre variante de l'algorithme de Nussinov essaie d'apparier les nucléotides de boucle de la tige-boucle, et donne une pénalité s'il n'y parvient pas. C'est relativement incohérent si l'on considère la stéréochimie de l'ARN. Il est préférable de ne pas tenter d'apparier deux nucléotides trop proches (à moins de trois nucléotides, par exemple) l'un de l'autre. Nous nous interdisons ici d'apparier deux nucléotides s'ils sont à moins de trois nucléotides l'un de l'autre. Pour cela, il suffit de laisser à zéro toutes les cellules de la matrice de Nussinov qui ont la forme $mat[i,i]$, $mat[i,i+1]$, $mat[i,i+2]$ et $mat[i,i+3]$. Cela correspond à décaler la diagonale délimitant les valeurs à calculer vers la gauche et le haut (cf. figure 5.4).

Finalement, le coût du meilleur appariement est donné par une des cellules situées en haut, à droite 5.4. Le score renvoyé par la fonction est donc le plus petit score trouvé.

En pratique, comme dans le cas précédent, la fonction de coût attend que les domaines des variables soient suffisamment réduits avant d'effectuer une propagation (en l'occurrence, elle attend que $bs_j - bi_i$ soit inférieur à taille_hélice_max). La complexité temporelle de la recherche de minimum est donc constante. Sa complexité spatiale est donnée par la matrice de Nussinov : $\mathcal{O}(\text{taille\_hélice\_max}^2)$.

Le calcul du support de x_i se calcule simplement en regardant, dans les matrices de propagation, où s'arrête l'alignement.

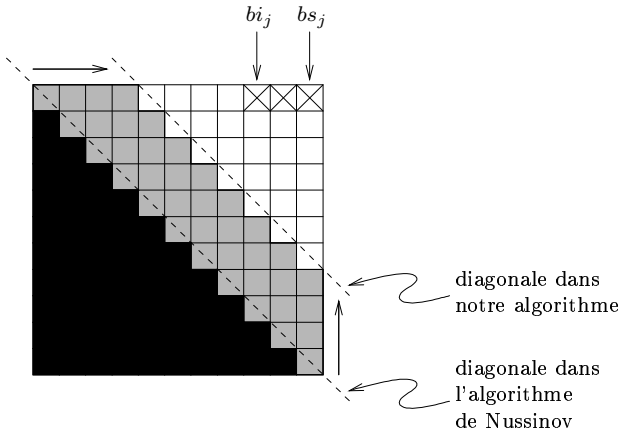


FIG. 5.4 – La matrice de l'algorithme de Nussinov laisse les cellules noires non remplies. Dans notre algorithme, les cellules grisées ne sont pas remplies non plus, car elles correspondent à un appariement de nucléotides trop proches. Les cellules pouvant contenir le coût minimum sont hachurées.

5.2.3.4 Le duplex

$\texttt{duplex}[\texttt{wobble}, \top_{\text{loc}}, \texttt{transformateur}](x_i, x_j, y_k, y_l)$
La fonction de coût de duplex évalue l'éventuel duplex formé par le mot contenu entre x_i et x_j d'une part, et le mot contenu entre y_k et y_l d'autre part (y_k et y_l sont des positions de la séquence cible).

Dans notre implantation, l'algorithme de recherche de minimum reste inactivé tant que x_i et x_j n'ont pas été affectées (le mécanisme d'ordonnancement des variables affecte toujours les variables sur la séquence principale d'abord). Une fois que x_i et x_j ont été affectées, il ne reste plus qu'à rechercher dans la séquence cible T quels mots peuvent s'apparier avec $S[x_i..x_j]$. C'est un problème de recherche de motif classique, largement étudié, et notre choix s'est porté sur l'utilisation d'un *tableau de suffixes amélioré* [AKO02, AKO04]. Celui-ci présente plusieurs avantages : comme un arbre des suffixes, son temps de contruction est linéaire en la taille du texte [KSB06] ; la recherche d'un mot dans la séquence peut se faire en temps linéaire en la taille du mot, indépendamment de la taille de la séquence ; sa structure ramassée de tableau permet un encodage plus concis que l'arbre des suffixes, et donne lieu à moins de *cache misses*.

Ce tableau *suf* contient tous les suffixes d'une longue séquence T, classés par ordre lexicographique. Dans la pratique, puisque le codage doit prendre le moins de place possible, ce tableau stocke la position de début de chaque suffixe. Une requête usuelle sur ce type de structure est de savoir si un mot donné est contenu dans la séquence. Si cette requête peut se traiter en temps proportionnel à $\ln(|T|)$ dans les tableaux de suffixes classiques, elle peut se faire, dans les tableaux de suffixes améliorés, en temps linéaire par rapport au mot donné, indépendamment de la taille de la séquence. Il faut pour cela ajouter quelques données supplémentaires dans notre tableau de suffixes, qui sont expliquées dans la définition suivante.

Définition 5.1 *pgpc*$[i]$ *est la taille du* plus grand préfixe commun *entre un élément du tableau suf*$[i]$ *et l'élément précédent suf*$[i-1]$.

Un intervalle (i, j) *est un* k*-intervalle si tous les éléments de suf*$[i]$, *suf*$[i+1], \ldots$ *suf*$[j]$ *commencent par un préfixe commun de taille* k. *Plus formellement, les* k*-intervalles sont définis de la manière suivante :*

- $\forall l \in]i..j], pgpc[l] \geq k$,
- $\exists l \in]i..j], pgpc[l] = k$,
- $pgpc[i] < k$,
- $pgpc[j+1] < k$.

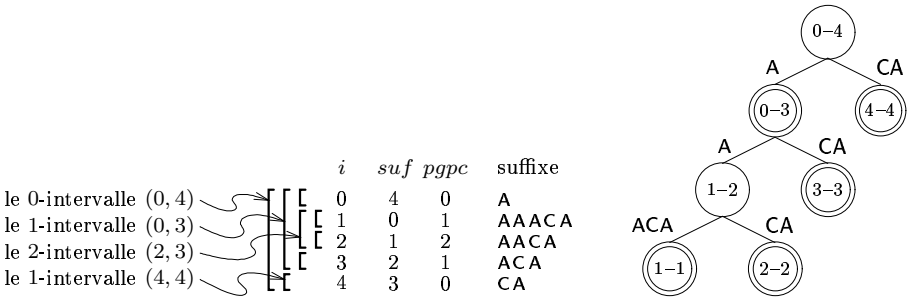
Remplir le tableau *pgpc* sur toutes les entrées du tableau sauf la première se fait en temps linéaire. Grâce à cette donnée et à d'autres valeurs intermédiaires calculées elles aussi en temps linéaire (cf. [AKO02, AKO04] pour les détails d'implantation), les k-intervalles se construisent eux aussi en temps linéaire.

Comme on le voit sur la figure 5.5(a), ces k-intervalles sont imbriqués les uns dans les autres. Ils ont en fait une forme d'arbre qui est exactement celle de l'arbre des suffixes de la séquence S. Dans cet arbre, les feuilles sont l'ensemble des k-intervalles de taille un, c'est-à-dire les suffixes du texte. Si (i, j) est un k-intervalle et (i', j') un k'-intervalle fils du précédent, alors nous noterons $\mathsf{lettres}(i, j) \rightarrow (i', j')$ les lettres étiquetant la branche menant de l'intervalle (i, j) à son fils (i', j').

Informellement, savoir si un mot est contenu dans un texte grâce un arbre des suffixes ou un tableau des suffixes s'effectue de la manière suivante. Supposons que chaque branche ne soit étiquetée que par une seule lettre. La recherche commence sur le nœud racine de l'arbre, et l'on regarde s'il existe une branche qui commence par la première lettre du mot. S'il n'en existe pas, alors on sait qu'il n'y a pas de sous-séquence qui commence par la première lettre du mot et la recherche est terminée. Si une telle branche existe, alors on la prend (nous dirons que la première lettre du mot est appariée) et on continue récursivement l'exploration avec le reste du mot dans le sous-arbre courant.

Les structures de données stockées dans le tableau de suffixes permettent de plus de donner le premier intervalle-fils d'un intervalle donné, ainsi que son premier intervalle-frère ; nous avons la structure nécessaire pour faire une recherche de mots en temps indépendant de la taille de la séquence. L'algorithme 40 décrit un algorithme de recherche de mot sans erreur grâce à un tableau de suffixes amélioré. Cette fonction n'est pas utilisée dans notre implantation puisque d'une part, nous acceptons les erreurs et d'autre part, nous recherchons un appariement et pas un mot dans une séquence. Toutefois, cet algorithme nous aidera à comprendre les développements ultérieurs.

La recherche commence par considérer le grand 0-intervalle qui inclut tous les suffixes de T. n_{pref} désignera ici le nombre de lettres déjà appariées du mot *mot* que l'on recherche dans T. Il est initialisé à 0 (ligne **60**). L'algorithme cherche ensuite le premier fils de l'intervalle courant commençant par la première lettre de *mot* à apparier (ligne **61**). S'il n'existe pas de tel fils, alors l'algorithme s'arrête : le mot recherché n'est pas dans T (ligne **62**). Ici, nous avons deux cas possibles. Soit l'intervalle-fils est une feuille de l'arbre, soit il est un nœud interne de l'arbre. Dans le premier cas (ligne **66**),



(a) le tableau des suffixes et les k-intervalles

(b) l'arbre des suffixes correspondant — les nœuds sont étiquetés par les k-intervalles correspondants

FIG. 5.5 – Deux structures de données stockant le mot AAACA.

il nous faut comparer le reste du mot avec la branche. Si les deux termes comparés sont égaux (ligne **67**), alors on a trouvé une solution, sinon, on a prouvé qu'il n'existe pas de solution (ligne **68**). Si l'intervalle-fils n'est pas une feuille de l'arbre (ligne **63**), il faut aussi comparer la branche et le reste du mot, et on a ici trois options. Les termes comparés peuvent être différents et l'on n'a pas de solution (ligne **64**). Les termes comparés peuvent être égaux et tout le mot a été comparé, et dans ce cas, on a une solution (ligne **65**). Si termes comparés sont égaux et que tout le mot n'a pas été comparé (c'est-à-dire que la branche était plus courte que le reste du mot), il faut alors continuer l'exploration dans le sous-arbre courant, en mettant à jour la valeur de l'intervalle courant, et la valeur n_{pref}.

L'algorithme de recherche de minimum vise à trouver le coût minimum de la fonction, pour y_k variant entre bi_k et bs_k, et y_l variant entre bi_l et bs_l. L'algorithme, récursif, explore l'arbre formé par les k-intervalles, recherchant tous les mots dans T pouvant s'apparier avec $S[x_i..x_j]$. L'algorithme est décrit en détail dans la procédure 41. Il suit la trame suivante. On tente d'apparier le mot $S[x_i..x_j]$ avec la première branche partant du nœud racine. Si tout le mot a été apparié, alors on a trouvé une solution potentielle, et on conserve le score. Si tout le mot n'a pas été entièrement apparié, alors il faut continuer récursivement la recherche dans le sous-arbre. Toutefois, plusieurs appariements entre la branche courante et le mot sont possibles. Chaque appariement donne un score différent, et un nombre de lettres du mot à rechercher différent. Il faut donc continuer la recherche pour tous les appariements possibles. Quand tout le sous-arbre a été exploré, on passe à la branche suivante.

Dans l'algorithme, la valeur n_{pref} désignera encore le nombre de lettres de *mot* déjà appariées, n_{err} désignera le nombre d'erreurs déjà enregistrées, et *min* sera le plus petit score de duplex trouvé jusqu'alors. Initialement, l'algorithme est appelé avec le 0-intervalle $(0, |T|)$, sachant qu'aucune erreur n'a été trouvée ($n_{\text{err}} = 0$), aucune lettre

Fonction 40 – RechercheExacte($mot \in \sigma^+$) : booléen

```
   (i, j) ← (0, |T|) ;
60 n_pref ← 0 ;
   tant que (vrai) faire
      (i', j') ← donneFils(i, j) ;
61    tant que (lettres((i, j) → (i', j')[0] ≠ mot[n_pref]) faire
         (i', j') ← donneFrère(i', j') ;
62       si ((i', j') = (⊥, ⊥)) alors
            retourner faux ;
63    si (i' ≠ j') alors
         m ← min{|lettres((i, j) → (i', j')|, |mot| − n_pref} ;
64       si ((lettres(i, j) → (i', j')[0..m] ≠ mot[n_pref..m]) alors
            retourner faux ;
65       si (m = |mot| − n_pref) alors
            retourner vrai ;
         (i, j) ← (i', j') ;
         n_pref ← n_pref + m ;
66    sinon
67       si (lettres((i, j) → (i', i'))[0..|mot| − n_pref] = mot[n_pref..|mot|]) alors
            retourner vrai ;
68       sinon
            retourner faux ;
```

du mot n'a été appariée ($n_{\text{pref}} = 0$), et que le meilleur appariement compte $\top_{\text{loc}}$ erreurs ($min = \top_{\text{loc}}$). Ligne **69**, on choisit le premier fils de l'intervalle. Comme dans le cas précédent, ce fils peut être ou pas une feuille de l'arbre des intervalles.

Supposons tout d'abord que ce ne soit pas une feuille et plaçons-nous ligne **70**. On essaie alors d'apparier les premières lettres du mot avec les lettres de la branche reliant l'intervalle-père à l'intervalle-fils, grâce à la fonction DonneCandidats, sachant que le score de cet alignement ne doit pas excéder $min - n_{\text{err}}$. Plusieurs appariements sont possibles, et DonneCandidats renvoie donc une liste d'éléments, dont chacun contient le nombre de lettres appariées ($long$), le score de l'appariement ($score$) et un booléen informant si toutes les lettres du mot ont été appariées ($tout$). Les appariements sont ensuite dépilés un à un. Si tout le mot n'a pas été apparié (ce cas est traité ligne **72**), alors la fonction est appelée récursivement pour poursuivre l'appariement. Sinon, comme c'est le cas dans la ligne **71**, on a une solution et il faut potentiellement mettre à jour le coût minimum.

Si l'intervalle courant est une feuille de l'arbre (ligne **73**), alors on appelle de la même manière DonneCandidats, mais on ne dépile que les éléments où mot a été totalement apparié. Si l'on a des solutions, on met alors à jour le coût minimum.

Fonction 41 – RechercheMinimum$((i,j), n_{\text{pref}}, n_{\text{err}}, min) : E$

```
69  (i', j') ← donneFils(i, j) ;
    tant que ((i', j') ≠ (⊥, ⊥)) faire
70      si (i' ≠ j') alors
            candidats ← DonneCandidats(lettres((i, j) →
            (i', j')), mot[n_pref..|mot|], min − n_err) ;
            tant que (¬candidats.vide()) faire
                (long, score, tout) ← candidats.depile() ;
                si (n_err + score ≤ min) alors
71                  si (tout) alors
                        min ← n_err + score ;
72                  sinon
                        min ← min{min, RechercheMinimum((i', j'), n_pref + long, n_err +
                        score, min)} ;
73      sinon
            candidats ← DonneCandidats(lettres((i, j) →
            (i', i')), mot[n_pref..|mot|], min − n_err) ;
            tant que (¬candidats.vide()) faire
                (long, score, tout) ← candidats.depile() ;
                si ((n_err + score < min) ∧ (tout)) alors
                    min ← n_err + score ;
        (i', j') ← donneFrère(i', j') ;
    retourner min ;
```

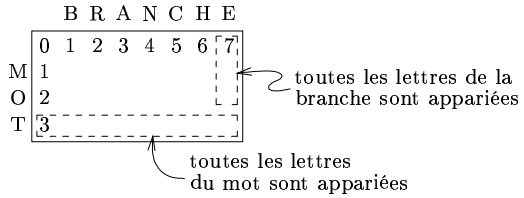


FIG. 5.6 – Une matrice de programmation comparant le mot recherché et la branche de l'arbre de recherche. Le côté bas contient tous les scores où le mot a été entièrement apparié. Le côté droit contient tous les scores où la branche a été entièrement appariée.

Il nous faut maintenant détailler le mécanisme d'alignement du mot avec une branche de l'arbre, décrit dans la fonction 42. Celle-ci doit générer tous les alignements dont le coût est inférieur à n_{err} entre la branche b et le reste du mot à aligner m. On peut distinguer deux cas principaux : soit tout le mot a été aligné et dans ce cas on a trouvé une solution, soit le mot n'a pas été entièrement aligné, et il faudra poursuivre l'exploration de l'arbre. Ce générateur d'alignement utilise dans notre cas une matrice de programmation dynamique, qui aligne le reste du mot avec la branche. La matrice est remplie selon la manière usuelle. Comme sur la figure 5.6, cette matrice a deux côtés importants : un côté bas, où tout le mot a été aligné, et le côté droit, où toute la branche a été alignée. On recherche sur ces deux côtés toutes les cellules ayant un coût inférieur à n_{err}, et on les renvoie toutes dans la liste *liste*.

Fonction 42 – DonneCandidats($b \in \sigma^+, m \in \sigma^+, n_{\text{err}} \in E$) : liste(entier, entier, booléen)

pour chaque ($i \in [0..|b|]$) **faire**
  $matrice[i, 0] \leftarrow i$;
pour chaque ($j \in [1..|m|]$) **faire**
  $matrice[0, j] \leftarrow j$;
pour chaque ($i \in [1..|b|]$) **faire**
  pour chaque ($j \in [1..|m|]$) **faire**

$$matrice[i,j] \leftarrow \min \left\{ \begin{array}{l} matrice[i-1,j]+1, \\ matrice[i,j-1]+1, \\ matrice[i-1,j-1]+c_a(b[i],m[j]) \end{array} \right\} ;$$

pour chaque ($i \in [0..|b|]$) **faire**
  si ($matrice[i, |m|] \leq n_{\text{err}}$) **alors**
    $liste.empile(b, matrice[i, |m|], \mathsf{vrai})$;
pour chaque ($j \in [0..|m| - 1]$) **faire**
  si ($matrice[|b|, j] \leq n_{\text{err}}$) **alors**
    $liste.empile(j, matrice[|b|, j], \mathsf{faux})$;
retourner *liste* ;

Il existe deux optimisations utilisées dans l'implantation. La première vient de l'observation que les intervalles de l'arbre sont souvent visités avec exactement la même configuration (c'est-à-dire avec un même nombre de lettres déjà appariées et un même nombre d'erreurs). On stocke alors dans chaque intervalle une information supplémentaire : l'état de la dernière configuration ayant traversé cet intervalle. Si l'on est amené à repasser par cet intervalle avec la même configuration que celle qui est stockée, alors l'algorithme ne poursuit pas la recherche dans le sous-arbre, puisqu'elle a été déjà été effectuée auparavant.

Une autre optimisation consiste à élaguer l'arbre d'un certain nombre d'intervalles grâce à la connaissance que l'on a des valeurs de y_k et y_l. Supposons par exemple que l'on sache que les solutions que l'on recherche sont dans la première moitié de la séquence, il est alors possible de n'explorer qu'une certaine partie — et non la totalité — de l'arbre. Cette optimisation nécessite l'utilisation d'informations, appelées $\mathit{borneInf}(i,j)$ et $\mathit{borneSup}(i,j)$, stockées dans chaque k-intervalle (i,j). Ces informations enregistrent les positions de la première et la dernière occurrence de la première lettre du préfixe commun du k-intervalle dans la séquence T. En d'autres termes, toutes les occurrences du mot qui étiquette le chemin partant de l'intervalle-racine et amenant à l'intervalle (i,j) commencent entre $\mathit{borneInf}(i,j)$ et $\mathit{borneSup}(i,j)$.

Lorsque l'on arrive sur un intervalle, il est alors possible de corréler ces informations avec les données des domaines de y_k. Plus précisément, si :

$$[\mathit{borneInf}(i,j)..\mathit{borneSup}(i,j)] \cap [bi_k..bs_k] = \varnothing$$

alors on sait que tous les mots que l'on pourra trouver dans le sous-arbre commenceront ailleurs que dans l'intervalle $[bi_k..bs_k]$, qui est l'intervalle où est censé débuter le mot recherché. La recherche ne se poursuit alors pas dans le sous-arbre courant. De même, on peut aussi tenter d'estimer les positions de la fin du mot et de les corréler avec le domaine de y_l. Si :

$$[\mathit{borneInf}(i,j) + |\mathsf{mot}| - 1 - \top_{\mathrm{loc}}..\mathit{borneSup}(i,j) + |\mathsf{mot}| - 1 + \top_{\mathrm{loc}}] \cap [bi_l..bs_l] = \varnothing$$

alors on peut ne pas explorer le sous-arbre.

Nous allons maintenant tenter de borner la complexité de l'algorithme de recherche de minimum. Supposons que l'on recherche un mot de taille m, avec $\top_{\mathrm{loc}}$ erreurs possibles. Dans le pire cas, chaque branche de l'arbre n'est étiquetée qu'avec un seul nucléotide. Le mot doit alors s'apparier avec au moins $m - \top_{\mathrm{loc}}$ branches, et au plus $m + \top_{\mathrm{loc}}$ branches. Le mot a $\binom{b}{m+b-1}$ possibilités pour s'apparier avec un chemin de b branches, et puisque b n'est pas supérieur à $m + \top_{\mathrm{loc}}$, alors $\binom{b}{m+b-1} \leq (2m + \top_{\mathrm{loc}})^m$. Sachant qu'il y a au plus σ^b chemins possibles à b branches, alors le nombre total de chemins examinés est borné par $\sum_{i=m-\top_{\mathrm{loc}}}^{m+\top_{\mathrm{loc}}} \sigma^i \times (2m + \top_{\mathrm{loc}})^m$, qui lui-même est borné par $(2 \times \top_{\mathrm{loc}} + 1) \times \sigma^{m+\top_{\mathrm{loc}}} \times (2m + \top_{\mathrm{loc}})^m$. Sur chaque chemin, l'algorithme de programmation dynamique effectué sur chaque branche a une complexité de $\mathcal{O}(m + \top_{\mathrm{loc}})$. La complexité totale est donc bornée par $\mathcal{O}(\top_{\mathrm{loc}} \times \sigma^{m+\top_{\mathrm{loc}}} \times (2m + \top_{\mathrm{loc}})^{m+1})$, qui est bien indépendante de la taille de la séquence T.

Puisque l'on a supposé que x_j était affecté, le support de bi_i par rapport à x_j est donc x_j. On utilise ensuite la fonction *borneInf* pour le support par rapport y_k et les valeurs $\mathsf{lettres}(0, |T|) \rightarrow (i, j)$ et *long* permettent de retrouver la taille du mot apparié dans T, et donc y_l.

5.2.3.5 Les interactions non-canoniques

$\mathtt{paire}[\mathsf{côté1}, \mathsf{côté2}, \mathsf{orientation}, \mathsf{famille}, \top_{\text{loc}}, \mathsf{transformateur}](x_i, x_j)$
La fonction de coût d'interactions non canoniques évalue une interaction entre deux faces données de deux nucléotides, dont la position est donnée par x_i et x_j. Le sens et la famille de l'interaction sont aussi donnés par l'utilisateur. Afin de calculer ces fonctions de coût, nous avons stocké une table *familles* qui, pour deux nucléotides donnés, deux faces d'interactions et un sens, donne l'ensemble des familles d'isostéricité (éventuellement vide) acceptant ces types d'interactions. Pour deux nucléotides donnés, n_1 et n_2, on peut alors définir une fonction f qui retourne :

$$\begin{cases} 0 & \text{si } \mathsf{famille} \in \mathit{familles}[n_1, n_2, \mathsf{côté1}, \mathsf{côté2}, \mathsf{sens}], \\ \mathsf{pénalité} & \text{sinon, et si } \mathit{familles}[n_1, n_2, \mathsf{côté1}, \mathsf{côté2}, \mathsf{sens}] \neq \varnothing. \\ \top_{\text{loc}} & \text{sinon.} \end{cases}$$

où pénalité est la pénalité donnée pour trouver une interaction qui appartient à une autre famille d'isostéricité.

Nous n'avons pas ici implanté de propagateur efficace pour calculer le minimum de cette fonction de coût. L'algorithme attend toujours que x_i ou x_j soit affectée avant de calculer le minimum de la fonction, ce qui est une forme de *partial forward checking* [FW92] limité aux bornes. On utilise la fonction f en itèrant sur les valeurs possible de x_j. On repère quel est le coût minimum donné par l'interaction entre le nucléotide pointé par x_i et les nucléotides du domaine de x_j. La fonction est donnée *in extenso* dans l'algorithme 43.

Fonction 43 – RechercheMinimum : E

```
min ← ⊤_loc ;
pour chaque j ∈ [bi_j..bs_j] faire
    min ← min{min, f(S[x_i], S[j])} ;
retourner min ;
```

Concernant la complexité du propagateur, il est clair que la complexité temporelle est de $\mathcal{O}(d)$, car aucune distance entre les deux variables n'est donnée en paramètre de la fonction de coût. Nous verrons par la suite comment garder malgré tout une complexité temporelle faible.

Le support de bi_i est ici calculé simplement durant la recherche de minimum.

5.2.4 La répétition

répétition[taille_min, taille_max, dist_min, dist_max, indels, $\top_{\text{loc}}$, transformateur](x_i, x_j, x_k, x_l)
La fonction de coût de répétition utilise quatre variables et évalue la similarité entre le mot inclus entre les deux premières variables et le mot inclus entre les deux dernières. Cette fonction de coût est extrêmement similaire à la fonction de coût d'hélice décrite dans le paragraphe 5.2.3.1. Nous utilisons donc le même algorithme, si ce n'est que le second brin est lu de gauche à droite (alors que dans la fonction de coût d'hélice, le second brin est lu dans l'autre sens), et que les scores d'appariement sont remplacés par des scores d'identité (un coût nul est donné si deux nucléotides sont identiques et un coût de un est donné dans le cas contraire). Nous ne détaillerons donc pas les algorithmes ici.

5.2.5 La composition

composition[nucléotides, relation, seuil1, seuil2, coût_min, $\top_{\text{loc}}$, transformateur](x_i, x_j)
La fonction de composition donne un coût en fonction de la proportion de nucléotides nucléotides (le plus souvent, {G, C}) entre les positions x_i et x_j. Ce score dépend des deux seuils seuil1 et seuil2 donnés en paramètres, et des coûts coût_min et coût_max. La fonction demande aussi un opérateur de comparaison et nous supposerons ici qu'il s'agit de $\geq$. Dans la pratique, le score $score(p)$ donné à une proportion p est défini par :

$$\begin{cases} \text{coût\_max} & \text{si } p < \text{seuil1}, \\ \frac{\text{coût\_min} - \text{coût\_max}}{\text{seuil2} - \text{seuil1}} \times p + \frac{\text{coût\_max} \times \text{seuil2} - \text{coût\_min} \times \text{seuil1}}{\text{seuil2} - \text{seuil1}} & \text{si seuil1} \leq p < \text{seuil2}, \\ \text{coût\_min} & \text{sinon.} \end{cases}$$

L'algorithme est calqué sur la fonction de coût d'interaction non-canonique : on attend que x_i ou x_j soit affectée avant de calculer le minimum de la fonction. L'algorithme 44, qui suppose que la variable x_i est affectée, lit les nucléotides entre x_i et bs_j et compte le nombre de nucléotides qui sont dans l'ensemble spécifié par l'utilisateur. Le score minimum est maintenu durant la lecture, et finalement renvoyé.

Fonction 44 – RechercheMinimum : E

```
min ← ⊤_loc ;
nbPos ← 0 ;
pour chaque j ∈ [x_i..bi_j[ faire
    si (S[j] ∈ nucleotides) alors
        nbPos ← nbPos + 1 ;
pour chaque j ∈ [max{x_i, bi_j}..bs_j] faire
    si (S[j] ∈ nucleotides) alors
        nbPos ← nbPos + 1 ;
    min ← min{min, score(nbPos / (j − bi_j + 1))} ;
retourner min ;
```

Les complexités temporelle et spatiale sont identiques à la fonction de coût d'interaction non-canonique. Elles sont respectivement linéaire en d et constante.

Le support de bi_i est ici calculé simplement durant la recherche de minimum.

5.2.6 L'espaceur

$\mathsf{espaceur}[\mathsf{dist1}, \mathsf{dist2}, \mathsf{dist3}, \mathsf{dist4}, \mathsf{seuil1}, \mathsf{seuil2}, \mathsf{coût\_min}, \top_{\text{loc}}, \mathsf{transformateur}](x_i, x_j)$

La fonction de coût d'espacement détermine la distance entre deux positions x_i et x_j. Cette information se propage très facilement et rapidement. C'est pour cela que la cohérence d'arc aux bornes, mais aussi la cohérence ∅-inverse (cf. définition 4.12), sont appliquées sur cette fonction de coût.

La fonction de coût prend en paramètre quatre distances, de dist1 à dist4. Elle utilise aussi deux coûts, coût_min et coût_max. Il est possible de connaître le minimum de la fonction en temps constant, même lorsqu'aucune variable n'a de domaine réduit. Il est donné par :

$$\begin{cases} \mathsf{coût\_max} & \text{si } bi_j - bs_i > \mathsf{dist4}, \\ \dfrac{\mathsf{coût\_min} - \mathsf{coût\_max}}{\mathsf{dist2} - \mathsf{dist1}} \times (bi_j - bs_i) + \dfrac{\mathsf{coût\_max} \times \mathsf{dist2} - \mathsf{coût\_min} \times \mathsf{dist1}}{\mathsf{dist2} - \mathsf{dist1}} & \text{sinon, et si } bi_j - bs_i > \mathsf{dist3}, \\ \mathsf{coût\_min} & \text{sinon, et si } bs_j - bi_i \geq \mathsf{dist2}, \\ \dfrac{\mathsf{coût\_max} - \mathsf{coût\_min}}{\mathsf{dist4} - \mathsf{dist3}} \times (bs_j - bi_i) + \dfrac{\mathsf{coût\_min} \times \mathsf{dist4} - \mathsf{coût\_max} \times \mathsf{dist3}}{\mathsf{dist4} - \mathsf{dist3}} & \text{sinon, et si } bs_j - bi_i \geq \mathsf{dist1}, \\ \mathsf{coût\_max} & \text{sinon.} \end{cases}$$

Dans le cas des contraintes dures, où il n'y a que deux distances, distance_min et distance_max, qui donnent les distances minimale et maximale entre les deux variables, il est particulièrement simple de propager la contrainte. On peut, comme dans AC-5 [HDT92], trouver des propagateurs efficaces. Ils sont alors :

- $bi_j \leftarrow bi_i + \mathsf{distance\_min}$,
- $bs_j \leftarrow bs_i + \mathsf{distance\_max}$,
- $bi_i \leftarrow bi_j - \mathsf{distance\_max}$,
- $bs_i \leftarrow bs_j - \mathsf{distance\_min}$.

où bi_i est la borne inférieure du domaine de x_i, et bs_i, sa borne supérieure.

Le support de bi_i n'est ici pas stocké car la recherche de minimum de la fonction se fait ici en temps constant.

5.2.7 Justification des choix effectués

Nous avons ici donné les grandes lignes des algorithmes établissant CAB et la 2B-cohérence sur chacun des éléments de structure. Nous avons tenté d'optimiser nos algorithmes, mais malgré tout, on peut se demander pourquoi ces choix d'implantation

ont été faits. Il existe en effet pour les algorithmes de *pattern-matching* un très grand nombre d'outils proposés. Nous nous sommes donc appuyés sur le livre [NR02], qui dresse un état de l'art complet des algorithmes *pattern-matching*, et notamment sur la recherche de mot approchée, comme c'est le cas pour les fonctions de coût de mot et de duplex. On peut y voir par exemple une série d'algorithme appelés *bit-parallel*, qui se servent du fait que les fonctions logiques « et », « ou », « ou exclusif », etc. peuvent se faire sur un mot machine (soit trente-deux voire soixante-quatre bits aujourd'hui) en temps constant. Ceci permet d'accélerer les algorithmes en pratique. Mais en revanche, il devient alors extrêment compliqué de prendre en compte des coûts non unitaires. En effet, l'utilisateur peut vouloir spécifier qu'une substitution d'un A en C donne un coût de un, mais qu'une substitution d'un A en G donne un coût de deux. Alors que cela ne pose pas de problème dans notre implantation, c'est plus difficilement réalisable sur des algorithmes *bit-parallel*.

D'autres types d'algorithmes sont présentés. Ils divisent un mot à rechercher avec erreurs en d'autres mots, plus petits, à rechercher sans erreur. Dès lors que l'on trouve un petit mot sans erreur, on essaie d'étendre la région trouvée, pour voir si le mot complet correspond. Cette stratégie paraît pour nous peu adaptée, car les mots à rechercher sont déjà souvent petits, et le fait que l'alphabet soit ici composé de quatre lettres fournirait beaucoup de solutions à la recherche sans erreur qui ne donneraient pas d'extension pouvant correspondre au mot complet recherché. Il n'en reste pas moins vrai que, pour les mots à rechercher sans erreur (cela peut être fréquent dans une signature), un type de recherche *ad hoc* pourrait être mis en place.

La même question du choix de l'implantation peut légitimement se poser lorsque l'on compare les fonctions de coût de mot et de duplex. En effet, toutes deux doivent rechercher un mot dans une longue séquence. Pourquoi ne pas alors utiliser le même algorithme pour les deux éléments de signature ? Notre sentiment est que les situations sont malgré tout légèrement différentes. La fonction de coût de mot s'utilise en général sur la séquence principale. Et lorsqu'une ancre est trouvée (c'est-à-dire lorsqu'un élément de structure où la probabilité d'apparaître par hasard est très faible) ou bien lorsqu'une variable est affectée, l'espace de recherche est largement réduit grâce aux contraintes de distance. En revanche, lorsque les mots sont à trouver sur la séquence cible — c'est ce que fait la fonction de coût de duplex —, où l'on n'a en général pas d'information, la région d'exploration peut être grande (c'est d'autant plus vrai que, dans des travaux futurs, nous souterions modéliser l'interaction entre une séquence principale et une parmi plusieurs séquences cibles). D'autre part, l'utilisation d'un algorithme de programmation dynamique pour la fonction de coût de mot donne une complexité temporelle linéaire en la taille de l'espace de recherche et la taille du mot. L'algorithme utilisé dans la fonction de coût de duplex a une complexité temporelle indépendante de la taille de l'espace de recherche, mais au moins exponentielle en la taille du mot et le nombre d'erreurs autorisées. En somme, lorsque l'espace de recherche est très grand, il semble préférable d'utiliser l'algorithme du duplex, alors que celui-ci ne semble plus très compétitif quand l'espace de recherche est réduit. C'est cette idée qui nous a conduit à choisir les algorithmes présentés. Mais bien sûr, seul un banc de tests serait à même de confirmer ou d'infirmer ces intuitions.

5.3 Optimisations

5.3.1 Introduction

Nous avons présenté dans les paragraphes précédents les algorithmes de propagation. Nous avons tenté d'optimiser ces algorithmes de *pattern-matching*. Nous avons vu que le formalisme des contraintes pondérées permettait de dissocier efficacement ces types d'algorithme du mécanisme de recherche de solutions proprement dit. On peut apporter encore quelques optimisations sur le second point. Nous avons donc repris quelques techniques classiques proposées par l'intelligence artificielle permettant de guider la recherche plus rapidement vers les solutions ou permettant de ne pas refaire les mêmes calculs plusieurs fois.

5.3.2 Gestion des supports

Le mécanisme d'établissement de CAB ou de la 2B-cohérence travaille souvent par vérification, c'est-à-dire qu'il cherche souvent à s'assurer qu'un support n'a pas été perdu lors de la suppression d'une valeur d'une variable voisine (voir l'algorithme 30 établissant CAB et la définition 4.3 sur les supports). Pour ne pas refaire le même travail plusieurs fois, nous stockons ce support à chaque fois que nous le trouvons. Ce support peut-être celui de la borne inférieure ou supérieure d'une des variables de la fonction de coût ou bien le support binaire. À chaque fois qu'une recherche similaire est lancée, nous recherchons d'abord si les supports stockés sont toujours valides. Cela permet un gain de temps considérable, sans augmenter la complexité spatiale, qui reste de $\mathcal{O}(e)$.

Ce mécanisme est en pratique très important, car les mécanismes de recherche de minimum trouvent le plus souvent exactement le même support d'une recherche à l'autre. Étant donné que la recherche de minimum est parfois coûteuse en temps (c'est particulièrement vrai pour les hélices), ce mécanisme permet de gagner un temps considérable.

5.3.3 Files de priorité

Lorsque l'on veut établir CAB ou la 2B-cohérence, il est assez intuitif de se dire qu'il vaut mieux vérifier les fonctions de coût d'espacement avant les fonctions de coût d'hélice. Cette idée est utilisée grâce au mécanisme suivant. En théorie, le mécanisme de révision ajoute dans un ensemble Q toutes les variables à reconsidérer. Dans la pratique, nous empilons plutôt toutes les fonctions de coût à revérifier. Ces fonctions de coût sont ensuite classées selon plusieurs critères et les plus prioritaires sont vérifiées en premier, à la manière de Choco [Lab00], un solveur de contraintes. Les contraintes dures, c'est-à-dire les fonctions de coût de type dur, sont toutes prioritaires par rapport aux fonctions de coût car elles réduisent plus efficacement les domaines des variables. Ensuite, les fonctions de coût simples à établir, comme la fonction de coût d'espacement, sont prioritaires par rapport aux fonctions de coût ayant une complexité temporelle plus élevée, comme la fonction de coût d'hélice. Enfin, puisque les événements ayant entraîné les révisions de fonctions de coût sont aussi stockés avec la fonctions de coût à réviser, les affectations, très informatives, sont traitées avant les modifications des bornes inférieures

et supérieures. Dans la pratique, on utilise autant de files que nécessaire pour stocker les contraintes, ce qui éviter de les réordonner.

Nous verrons que ce mécanisme permet de diminuer la complexité temporelle théorique de notre problème. En pratique, il vaut mieux les contraintes dures de distance avant toute chose. Cela permet de ne lancer les recherches de minima sur les fonctions de coût que lorsque les domaines des variables sont les plus réduits possibles. Ce mécanisme fait gagner beaucoup de temps.

5.3.4 Ordonnancement dynamique des variables

Lorsque les séquences considérées pour trouver des ARNnc sont très longues, il est vital de trouver des *ancres*, c'est-à-dire des éléments très contraints, c'est-à-dire très conservés dans les séquences déjà observées de la famille et ayant très peu de chances d'apparaître par hasard. Ce peut être par exemple un long mot avec peu d'erreurs et de nucléotides ambigus, ou une longue hélice avec peu d'erreurs autorisées. À l'inverse, les variables sur la séquence cible doivent à tout prix être affectées en dernier, puisque l'on a le plus souvent très peu d'informations sur elles. Il faut donc ordonnancer les variables de la façon la plus efficace possible.

De nombreuses heuristiques ont été proposées pour ordonnancer les variables. Dans le cas précis de la recherche de motifs, où toutes les fonctions de coût ont une sémantique claire, nous avons suivi l'idée de RnaMot [GHC90, LGC94]. Nous avons donc affecté un poids à toutes les fonctions de coût, prenant en compte les valeurs de leurs paramètres. Les variables sont alors pondérées par la somme des poids pesant sur les fonctions de coût portant sur elle. La variable ayant le plus fort poids est affectée en premier.

Le poids que porte chaque fonction de coût a été empiriquement déterminé ainsi :

- le mot : $|\mathsf{mot}| - \top_{\mathrm{loc}}$, où la taille du mot est ici le nombre de nucléotides non ambigus ;
- l'hélice simple : $\mathsf{long\_min} - \top_{\mathrm{loc}}$;
- l'hélice complexe : $\mathsf{long\_max} - \top_{\mathrm{loc}}$;
- le repliement : $\mathsf{long\_min}/2$;
- la répétition : $\mathsf{long\_min} - \top_{\mathrm{loc}}$;
- la distance : 1.

Toutes les autres fonctions de coût ont un poids nul et les variables sur le brin cible ont un poids négatif fixé à 1000.

Dans la pratique, si l'on peut trouver des problèmes où l'ordonnancement est important, cette optimisation ne change pas fondamentalement la rapidité de résolution de la plupart des instances.

5.3.5 Branchement binaire

Nous avons présenté dans le paragraphe 3.4.2.2 un l'algorithme de recherche basé sur une méthode de séparation et évaluation en profondeur explorant un arbre d-aire, c'est-à-dire où chaque nœud de l'arbre peut avoir d nœuds-fils. Nous avons choisi de transformer cet arbre d-aire en arbre binaire, en utilisant le schéma donné dans la fi-

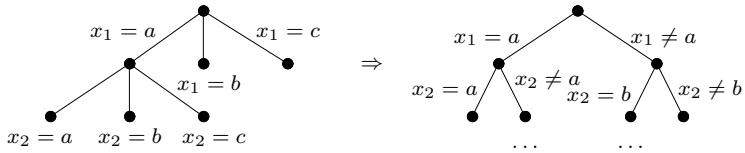


FIG. 5.7 – La transformation d'un arbre ternaire en un arbre binaire et le mécanisme de réfutation.

gure 5.7. Les branches du type $x_i = v_i$ sont des branches d'affectation et les branches du type $x_i \neq v_i$ sont des branches de réfutation [Rég95]. En pratique, exécuter les algorithmes de filtrage non seulement après chaque affectation, mais aussi après chaque réfutation permet de gagner beaucoup de temps. La procédure 45 décrit comment implanter un tel mécanisme (les régions grisées désignent le code identique à l'algorithme 6 originel).

Procédure 45 – RechercheSolutions($x_i \in \mathcal{X}$)

```
essayer
    tant que (I_i ≠ ∅) faire
        essayer
            x_i ← bi_i ;
            Filtrage() ;
            x_j ← ProchaineVariable() ;
            RechercheSolutions(x_j) ;
        bi_i ← succ(bi_i) ;
        Filtrage() ;
```

Ce mécanisme est en fait extrêmement important pour résoudre les instances de manière efficace. Le branchement binaire et la réfutation permettent de gagner plusieurs ordres de grandeur temporelle dans la résolution des problèmes.

5.3.6 Backjumping

Les techniques de *backjumping* sont assez souvent implantées dans les solveurs de contraintes. Étudions ses mécanismes sur le cas suivant. Supposons que l'on ait un réseau de contraintes dures avec trois variables x_1, x_2 et x_3 (cf. 5.8). Supposons de plus qu'il existe une contrainte binaire entre x_1 et x_2 (par exemple une contrainte de distance) et une contrainte binaire entre x_2 et x_3. On décide d'affecter la variable x_2. On affecte ensuite x_1 puis x_3 et l'on se rend compte que, pour toute valeur de x_3, on ne trouve aucune solution. Un algorithme normal procède à un retour-arrière sur x_1, pour tester une nouvelle valeur. Mais c'est manifestement inutile, puisque x_1 n'est reliée à x_3 que par la variable déjà affectée x_2. Essayer une autre valeur de x_1 nous ramènera au même problème : aucune valeur de x_3 ne donnera de solution. Il faut directement procéder à un retour-arrière sur x_2.

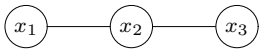


FIG. 5.8 – Un exemple de réseau de contraintes dures.

Une optimisation consiste donc à détecter ce type de problème et à tenter d'en tirer profit pour accélerer la recherche. C'est le *backjumping*. Il existe beaucoup de types de *backjumpings* différents. Nous avons implanté le plus simple : le *backjumping* basé sur le graphe [Dec90]. Malheureusement, cette optimisation n'est valable que si les contraintes sont dures ou si $c_\varnothing$ est nul. Il existe d'autres techniques de *backjumping* dans les RCP mais celles-ci sont souvent plus coûteuses [ZM07].

Supposons qu'une exception se lève lorsque la variable x_k est affectée et que cette exception soit transmise à la variable précédente x_i dans l'ordre d'affectation des variables. On souhaite sauter x_i si x_i et x_k ne sont plus connectées dans le graphe. Si une propriété de filtrage au moins aussi forte que le *forward checking* [GB65, HE80] est maintenue dans l'arbre de recherche, on peut de plus retirer du graphe toute variable affectée. Nous avons vu dans les paragraphes précédents que c'était le cas pour toutes les fonctions de coût sauf pour la fonction de duplex. Dans cette dernière fonction, la fonction ne propage rien tant que les deux variables qui sont sur la séquence principale ne sont pas affectées. Nous avons donc traité ce cas spécial. L'algorithme 46 décrit ce mecanisme (pour des raisons de clarté, sans le traitement spécifique de la fonction de duplex). On suppose que, quand une variable x_k est affectée, si une exception est levée, il est possible de rattrapper cette exception avec cette variable x_k.

Procédure 46 – RechercheSolutionsBackjumping($x_i \in \mathcal{X}$)

```
essayer
    tant que (I_i ≠ ∅) faire
        essayer
            x_i ← bi_i ;
            Filtrage() ;
            x_j ← ProchaineVariable() ;
            RechercheSolutions(x_j) ;
        rattraper x_k
            si ((c_∅ = 0) ∧ (x_k ∉ EnsembleConnexe(x_i))) alors
                lever x_k ;
        bi_i ← succ(bi_i) ;
        Filtrage() ;
rattraper x_k
    si ((c_∅ = 0) ∧ (x_k ∉ EnsembleConnexe(x_i))) alors
        lever x_k ;
```

Dans la procédure RechercheSolutionsBackjumping, on calcule l'ensemble des va-

riables connectées à une variable donnée x_i. La fonction 47 effectue ce travail.

Fonction 47 – EnsembleConnexe($x_i \in \mathcal{X}$) : $\wp(\mathcal{X})$

```
X ← {x_i} ;
pile.empile(x_i) ;
tant que (¬pile.vide) faire
    x_j ← pile.depile ;
    pour chaque c_k ∈ {c ∈ C | x_j ∈ var(c)} faire
        pour chaque x_l ∈ var(c) \ {x_j} faire
            si (x_l ∉ X) alors
                X ← X ∪ {x_l} ;
                si (¬EstAffectée(x_l)) alors
                    pile.empile(x_l) ;
retourner X ;
```

Ce mécanisme change en pratique assez peu la rapidité de résolution des problèmes car les cas où $c_\varnothing$ est nul sont en fait assez rares, ce qui réduit la portée de cette optimisation.

5.3.7 Conclusion

Nous avons donc utilisé plusieurs optimisations classiques appliquées au mécanisme de recherche de solutions : gestion des supports, files de priorités, ordonnancement dynamique, branchement binaire et *backjumping*. Ces mécanismes peuvent être encore améliorés. On peut penser par exemple à un meilleur ordonnancement des variables. Nous verrons dans le prochain paragraphe que la complexité temporelle théorique du problème est grandement liée à l'incertitude de distance entre deux variables. En d'autres termes, plus la différence entre la distance minimale et la distance maximale entre deux variables est grande, plus la recherche sera longue. La question est alors de savoir quelle est la variable à choisir pour que, une fois affectée, celle-ci réduise ces incertitudes entre chaque couple de variables. Il semble que le meilleur choix soit alors la variable pointant une position située au milieu du motif à rechercher. Mais comment estimer ce milieu ? Un moyen serait peut-être d'étudier les contraintes de distances et de tenter d'estimer les positionnements des variables les unes par rapport aux autres. Il serait ensuite possible de choisir la variable centrale. Nous n'avons pas encore mis cette idée en application.

Une autre optimisation intéressante serait d'étudier la structure du graphe des contraintes. Celle-ci est en effet très particulière car elle est presque linéaire. Il serait avantageux d'exploiter cette structure afin de trouver les variables-clefs qui, une fois affectées, divisent le problème global en deux ou plusieurs problèmes indépendants. Il faudrait alors aussi ajouter les mécanismes nécessaires pour que chaque problème indépendant soit traité séparément.

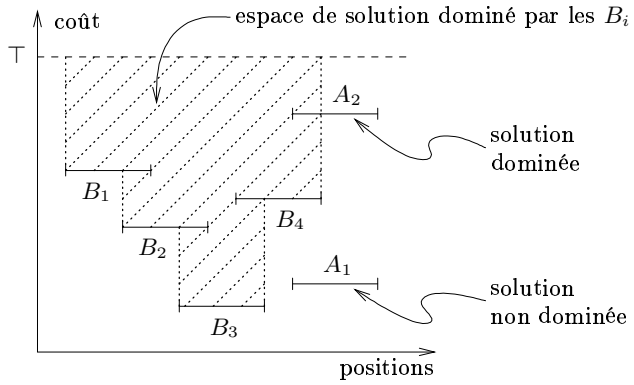


FIG. 5.9 – La dominance — les B_i sont des solutions anciennement trouvées, et les A_i sont des nouvelles solutions ; A_1 est localement optimale, alors que A_2 ne l'est pas car elle est dominée par B_4.

5.4 Choix des solutions

Pour un ARN candidat que l'on recherche, on trouve parfois plusieurs solutions, ne différant entre elles que par le décalage d'un élément de structure sur la gauche ou la droite de quelques nucléotides. L'idéal serait ici de ne retenir qu'une solution, de préférence celle se rapprochant le plus possible du candidat à retrouver. On utilise donc un mécanisme permettant de ne sélectionner que les *solutions localement optimales*. Pour cela, on définit une relation de dominance qui permet de comparer deux solutions.

Définition 5.2 *Une solution A* domine *une solution B si :*
- *A et B se chevauchent sur la séquence principale, et $\mathcal{V}(A) < \mathcal{V}(B)$, ou*
- *A et B se chevauchent sur la séquence principale, et $\mathcal{V}(A) = \mathcal{V}(B)$, et A est située plus à gauche que la solution B.*

Une solution est localement optimale *si elle n'est dominée par aucune autre solution.*

En d'autres termes, si deux solutions se chevauchent, alors celle de plus petit coût domine l'autre. En cas d'égalité, la solution la plus à gauche domine l'autre. Seules les solutions localement optimales sont données à l'utilisateur (voir figure 5.9).

Pour savoir si une solution nouvellement trouvée est localement optimale, il faut *a priori* la comparer avec toutes les solutions déjà trouvées et stockées. De plus, une nouvelle solution peut aussi dominer une solution localement optimale déjà trouvée auparavant. Étant donné que les deux problèmes sont très similaires, nous ne détaillerons que le premier : savoir si une nouvelle solution A est dominée par les solutions B_i déjà trouvées. Pour cela, nous utilisons une liste L_i recensant toutes les positions $L_i.debut$ marquant le début des solutions déjà trouvées. Dans cette liste, les élements sont classés par ordre croissant de $L_i.debut$. Partant de chaque élément de la liste, on crée une nouvelle liste L_{ij}. Celle-ci recense, pour chaque solution commençant en $L_i.debut$, sa position de fin $L_{ij}.fin$ et son coût $L_{ij}.coût$. Pour accéler l'algorithme, on ajoute dans

chaque cellule $L_i.fin$ la position de fin la plus à droite de toutes les solutions commençant en $L_i.debut$. En d'autres termes, $L_i.fin = \max_j\{L_{ij}.fin\}$.

La fonction 48 détermine si une nouvelle solution A est localement optimale. On suppose que l'on connaît sa position de début sur la séquence principale ($A.debut$), sa position de fin ($A.fin$) et son coût ($A.coût$). La ligne **74** lit toutes les positions de début des solutions déjà connues. En fait, seuls les derniers éléments de la liste contiennent une solution capable de dominer A, car notre solveur instancie toujours les variables à la borne inférieure de leur domaine. L'algorithme lit donc la liste L_i de la dernière cellule vers la première, en s'arrêtant lorsque la différence $A.debut - L_i.debut$ est plus grande que la taille de n'importe quel ARN (fixée à taille_ARN_max, cf. ligne **75**). Pour chaque cellule L_i lue, on regarde si les solutions partant de cette cellule et A peuvent se chevaucher (ligne **76**). Si c'est le cas, alors on parcourt toutes les solutions de L_i (ligne **77**) et on regarde si les deux solutions se chevauchent (ligne **78**). Si c'est le cas, alors l'une domine forcément l'autre et c'est déterminé par la ligne **79**. Si aucune solution déjà enregistrée ne domine A, alors la solution est localement optimale.

Fonction 48 – EstLocalementOptimal(A) : booléen

```
74 pour chaque i ∈ [1..|L|] à l'envers faire
75   si (A.debut − L_i.debut > taille_ARN_max) alors
       retourner vrai ;
76   si ([L_i.debut..L_i.fin] ∩ [A.debut..A.fin] ≠ ∅) alors
77     pour chaque j ∈ [1..|L_i|] faire
78       si ([L_i.debut..L_ij.fin] ∩ [A.debut..A.fin] ≠ ∅) alors
79         si
           ((L_ij.coût < A.coût) ∨ ((L_ij.coût = A.coût) ∧ (L_i.debut ≤ A.debut)))
           alors
             retourner faux ;
   retourner vrai ;
```

La double liste L_{ij} ne réduit bien sûr pas la complexité spatiale du stockage des solutions mais elle permet de se prononcer rapidement sur l'existence d'une solution localement optimale. En pratique, ce mécanisme est appelé à chaque fois qu'une solution est trouvée. De plus, la double liste s'avère une structure de donnée efficace pour un autre mécanisme : l'exploitation de la dominance pour la mise à jour de la valeur $\top$.

D'après la définition de la dominance, lorsque l'on découvre une solution A chevauchant une solution déjà connue B et dont le coût est strictement supérieur au coût de B, A est alors dominée et donc inintéressante. On peut utiliser cette information pour élaguer l'arbre de recherche : si l'on explore une région dont on sait qu'elle chevauche une solution B, alors on peut abaisser la valeur de $\top$ à $B.coût + 1$. La fonction 49 donne la valeur qu'il faut affecter à $\top$.

Sur la ligne **80**, la fonction considère toutes les variables affectées du problème courant et détermine les positions la plus à gauche (min) et la plus à droite (max). La

valeur val a été initialisé à $\top^0$, qui est la valeur originale de $\top$, telle que donnée par l'utilisateur. L'algorithme parcourt la double liste de la même manière que dans EstLocalementOptimal pour connaître le coût de la meilleure solution chevauchant l'intervalle $[min..max]$. La valeur renvoyée est ce coût, incrémenté de un.

Fonction 49 – MiseÀJourTop : E

```
min ← |S| + 1 ;
max ← −1 ;
val ← ⊤^0 ;
80 pour chaque x_i ∈ 𝒳 faire
    si (EstAffectée(x_i)) alors
        min ← min{min, x_i} ;
        max ← max{max, x_i} ;
pour chaque i ∈ [1..|L|] à l'envers faire
    si (min − L_i.debut > taille_ARN_max) alors
        retourner val ;
    si ([L_i.debut..L_i.fin] ∩ [min..max] ≠ ∅) alors
        pour chaque j ∈ [1..|L_i|] faire
            si ([L_i.debut..L_ij.fin] ∩ [min..max] ≠ ∅) alors
                val ← min{val, L_ij.coût + 1} ;
retourner val ;
```

Cette fonction met la valeur de $\top$ à jour à chaque fois que l'on affecte une valeur ou que l'on réfute (le mécanisme de réfutation est expliqué dans le paragraphe 5.3.5).

5.5 Complexité théorique du programme

Nous allons ici donner un majorant de la complexité temporelle théorique de notre programme. Considérons donc un problème quelconque, qui pour l'instant ne fait intervenir qu'une seule séquence. Les problèmes contenant les duplex seront étudiés par la suite. Nous allons ici supposer que la signature est donnée, nous calculerons la complexité de notre algorithme par rapport à la séquence d'entrée. Cela nous conduit à considérer que le nombre de variables et le nombre de fonctions de coût est constant.

Nous avons vu que la cohérence d'arc aux bornes sur des réseaux binaires avait une complexité temporelle de $\mathcal{O}(ed^2 + \min(\top, nd) \times n)$. Dans les réseaux d'arité quatre, comme ceux que nous avons, cette complexité passe à $\mathcal{O}(ed^4 + \min(\top, nd) \times n)$. Néanmoins, une propriété vue dans le chapitre 4 est susceptible de nous intéresser pour faire descendre cette complexité. Il s'agit de la propriété 4.14, qui indique que si le minimum de toutes les fonctions de coût, lorsqu'une de ses variables a une valeur fixée, peut être trouvé en temps constant, alors la complexité temporelle de CAB devient $\mathcal{O}(ed + \min(\top, nd) \times n)$, c'est-à-dire linéaire par rapport d. Nous avons vu que c'était le cas de toutes les fonctions de coût, sauf pour les interactions non-canoniques et pour la

composition. La propriété ne peut donc pas pour l'instant s'appliquer. La suite va nous montrer que ces deux fonctions ne posent en réalité pas problème.

Nous allons supposer que les n variables sont liées à leurs voisines par une fonction de coût modélisant une contrainte de distance dure. En d'autres termes, il doit exister une contrainte de distance dure entre les variables x_i et x_{i+1}, pour $i \in [1..n-1]$. Cette supposition est motivée par le fait qu'en général, le modélisateur a une idée de la distance relative des éléments de signature. Considérant toutes ces contraintes de distance, nous appellerons D la plus grande valeur $d_2 - d_1$, c'est-à-dire la plus grande différence possible entre les distances maximales et les distances minimales autorisées.

Supposons maintenant que l'on affecte la variable x_1. Après propagation de la contrainte de distance reliant x_1 à x_2, la variable x_2 a un domaine réduit à au plus D positions possibles. Ensuite, la variable x_3 voit son domaine se réduire à $2D$ positions possibles, etc. Donc, une fois que la variable x_1 est affectée, les variables x_i ont une taille de domaine réduite à $(i-1)D$, pour $i \in [2..n]$. Le nombre d'affectations totales possibles, si x_1 est affectée, est donc borné par $1 \times D \times \cdots \times (n-1)D = (n-1)! \times D^{n-1}$. Le nombre total d'affections possibles, lorsque x_1 n'est pas encore affectée, est donc borné par $d \times (n-1)! \times D^{n-1}$. Puisque nous considérons que le réseau est donné, alors $d \times (n-1)! \times D^{n-1} = \mathcal{O}(d)$.

Avant d'arriver à une affectation totale, il aura fallu appliquer CAB. Puisque la taille du plus grand domaine, lorsqu'une variable est affectée, est de $(n-1)D$, la complexité temporelle de CAB est de $\mathcal{O}(enD + \min(\top, nnD) \times n)$. Puisque CAB est maintenue dans l'arbre de recherche, alors le temps total passé à appliquer CAB, lorsqu'une variable est affectée, est $\mathcal{O}\left((enD + \min(\top, nnD) \times n) \times \left(d \times (n-1)! \times D^{n-1}\right)\right) = \mathcal{O}(d)$.

Revenons maintenant sur le cas des interactions non-canoniques et de la composition. Nous avons vu d'une part que leurs propagateurs ne sont lancés que lorsqu'une des deux variables de ces fonctions de coût sont affectées. Nous avons d'autre part évoqué le mécanisme de gestion de files de priorité. Les files lancent la propagation des contraintes dures de distances avant la propagation d'interactions non-canoniques et de compositions. Les propagateurs de ces fonctions sont donc lancés quand leurs deux variables ont déjà une distance bornée entre elles. En pratique, cela signifie que les propagateurs des interactions non-canoniques et de la composition prennent un temps constant. La propriété 4.14 peut donc s'appliquer. Incluons maintenant le temps passé à appliquer CAB avant qu'une variable ne soit affectée. La complexité est de $\mathcal{O}(ed + \min(\top, nd) \times n) = \mathcal{O}(d)$. S'il n'existe pas de duplex, alors la complexité totale est de $\mathcal{O}(d)$.

S'il existe un duplex, alors le mécanisme suivant est utilisé. Les variables représentant des positions sur la séquence cible ne sont affectées que lorsque toutes les variables de la séquence principale le sont, et les contraintes de duplex ne sont réveillées que lorsque les deux variables de la séquence principale sont affectées. Il n'y a donc propagation sur les variables de la séquence cible qu'au début (lors de la propagation initiale), lorsque les deux variables de la séquence principale d'une fonction de coût de duplex sont affectées, ou lorsqu'une variable de la séquence cible est affectée. Lorsqu'une variable est affectée dans la séquence cible, nous avons vu que la complexité temporelle de CAB est linéaire par rapport à d', où nous appellerons d' la taille de la séquence cible, souvent bien plus petite que la séquence principale. Lorsqu'aucune variable n'est affectée, alors chaque

établissement de CAB prend un temps $\mathcal{O}\left(ed' + \min(\top, nd') \times n\right) = \mathcal{O}(d')$. Nous avons vu qu'il ne pouvait y avoir que $d \times (n-1)! \times D^{n-1}$ affectations totales possibles sur le brin principal, si bien que les fonctions de coût de duplex ne peuvent être réveillées que $ed \times (n-1)! \times D^{n-1} = \mathcal{O}(d)$ fois. Le temps passé à établir CAB sur le brin cible est donc borné par $\mathcal{O}(dd')$. Si l'on inclut le temps passé sur le brin principal et avec les même simplifications que précédemment, la complexité totale reste de $\mathcal{O}(dd')$.

Nos algorithmes ont donc une complexité linéaire par rapport à la taille de la plus longue séquence. Avec l'apparition d'un duplex, la complexité augmente d'un facteur d', où d' est la taille de la séquence cible.

Il faut aussi noter que la complexité fait intervenir le terme $(n-1)!$, qui peut augmenter très rapidement avec le nombre de variables, puisque $n! \sim \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n$. Un problème dépassant la centaine de variables devient donc difficilement soluble.

La formule fait aussi intervenir une dépendance en D^{n-1}, où D est la plus grande différence entre la distance minimale et la distance maximale des contraintes dures de distances. Dans les faits, si les contraintes de distances sont lâches, alors le problème devient aussi difficile à résoudre.

5.6 Conclusion

Nous avons présenté dans ce chapitre les algorithmes de filtrage basés sur du *pattern-matching*. Nous avons essayé d'implanter des algorithmes relativement optimisés. Nous avons utilisé des structures de données récentes comme les tableaux de suffixes améliorés et des algorithmes éprouvés et optimisés comme dans la fonction de coût de mot. Nous avons aussi tenté d'apporter de nouveaux algorithmes quand ceux qui étaient proposés ne répondaient pas à nos questions. C'est par exemple le cas de la recherche de duplex avec erreurs.

De plus, un certain nombres d'algorithmes développés par la communauté de l'intelligence artificielle prennent leur place ici simplement. Les algorithmes de filtrage s'appuient sur une sémantique clairement définie des fonctions de coût et s'appliquent avec des complexités temporelle et spatiale connues. La gestion des supports, l'ordonnancement des variables, les files de priorité contenant les fonctions de coût à réviser, le *backjumping* sont autant de mécanismes simples qu'il a été facile d'insérer dans notre prototype.

Nous avons aussi donné les détails d'un mécanisme de choix des solutions. Celui-ci se base sur les préférences fournies par le modélisateur pour supprimer certaines solutions qui sont *a priori* non voulues par l'utilisateur. Nous verrons dans les chapitres suivants que ce mécanisme est parfois crucial pour ne pas afficher un grand nombre de solutions non pertinentes à l'utilisateur. Ce mécanisme est d'ailleurs utilisé pour accélerer la recherche lorsqu'une solution a été trouvée dans le voisinage de la région explorée.

Peut-être pourrait-on tenter d'apporter quelques améliorations à la définition de la dominance, qui elle-même sert à définir les solutions localement optimales. On peut tout d'abord objecter que, lorsque deux solutions se chevauchent, l'une d'entre elles est systématiquement éliminée. Les deux solutions peuvent être deux ARNnc. C'est en

pratique rarement le cas. Dans les exemples que nous avons, ce cas n'intervient jamais. Il peut ensuite s'avérer qu'une solution prédite soit plus grande que l'ARN recherché et que cette solution chevauche un autre véritable ARN, qui serait alors éliminé. Ce cas arrive en pratique et nous l'avons observé lorsqu'une signature n'était pas suffisamment spécifique. Dans ce cas, le logiciel interprète mal ce qu'il faut considérer comme « bonne solution » et est amené à supprimer des candidats. Mais il suffit alors de mieux préciser son modèle pour corriger ce problème. Un troisième reproche que l'on pourrait avoir est que lorsque deux solutions se chevauchent et que le score des deux solutions est identique, une solution est éliminée de façon arbitraire. Une autre possibilité, lorsque l'outil n'a pas les moyens de séparer deux ex æquo, est de n'en éliminer aucun. Mais, en pratique, on se heurte au même problème que précédemment. Les deux candidats trouvés se différencient en général uniquement par le décalage de quelques variables. Deux solutions chevauchantes ex æquo reflètent donc plutôt un seul ARN non-codant, sur lequel deux repliements sont possibles. Le mieux est alors de n'en proposer qu'un seul car le fait de ne pas séparer les ex æquo amène souvent une dizaine de solutions pour un seul ARN, qui sont en fait très difficilement interprétables. Une dernière critique possible porte sur l'utilisation de données thermodynamiques. Il serait possible soit d'intégrer l'énergie de la structure trouvée [ZS81] dans le calcul du score d'une solution, soit au moins d'utiliser cette énergie pour séparer deux solutions ex æquo. Nous considérons que cet ajout pourrait être fait simplement et que cela pourrait améliorer la qualité de la détection des solutions localement optimales.

Nous refermons ce chapitre sur le constat que même si le problème de recherche de signatures d'ARN est en théorie un problème NP-complet [Via04], nous pouvons faire quelques hypothèses simples et souvent réalistes, qui permettent de le résoudre en temps linéaire en la taille de la séquence. Le caractère NP-complet du problème reste néanmois intéressant pour identifier les clases d'algorithmes permettant de le résoudre.

Chapitre 6

Génération automatique de signatures

6.1 Introduction

Dans les chapitres précédents, nous avons décrit un outil capable de rechercher les membres d'une famille d'ARN, lorsque cette famille est décrite par une signature. Cette signature capture l'ensemble des éléments structurels conservés dans les membres de la famille. Cette approche suppose donc que cette signature soit connue et que l'utilisateur traduise cette signature en un descripteur, c'est-à-dire qu'il traduise la signature à l'aide d'un langage compréhensible par l'outil de recherche d'ARN.

Même lorsque le langage du descripteur est simple et concis, la recherche de la signature et la traduction en descripteur sont parfois des tâches fastidieuses, nécessitant un fort investissement. Nous nous proposons dans ce chapitre d'automatiser ce processus. Nous avertissons toutefois le lecteur que ce travail est encore préliminaire.

Nous considèrerons un jeu de données observées qui sera en l'occurrence un alignement de séquences d'ARN non-codants d'une famille donnée, où un ensemble d'interactions est donné (nous ne devrons donc pas rechercher de structure secondaire). Notre but sera de chercher une signature qui reflète au mieux les données observées. Nous voudrions par exemple générer une signature qui reconnaisse au moins toutes les données observées mais qui soit aussi à même de déclarer comme non-candidats les séquences qui ne sont pas des membres de la famille.

Le travail à effectuer doit commencer par trouver les éléments conservés dans toutes les séquences. Nous supposerons qu'en première approximation ceux-ci seront modélisables par des mots, des hélices et des espaceurs définis dans le chapitre 3. Notre approche ne remettra pas en question l'alignement et supposera que celui-ci est correct. Un alignement de mauvaise qualité donnera donc nécessairement une signature de mauvaise qualité. Par exemple, si un mot conservé est mal aligné, il sera impossible de détecter le signal et la signature sera peu spécifique, c'est-à-dire peu à même de discriminer les ARN recherchés du reste de la séquence. De plus, si un alignement n'inclut que les membres d'un seul organisme, et que l'on veut effectuer une recherche d'ARN

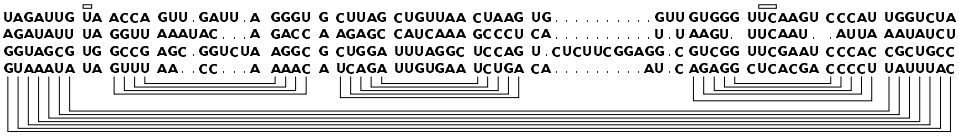


FIG. 6.1 – Un alignement d'ARN de transfert.

sur un autre organisme, alors la variabilité que l'on rencontre d'un organisme à l'autre ne sera pas prise en compte et la signature sera sans soute peu sensible, c'est-à-dire qu'elle oubliera des solutions. Mais en général, ce défaut est partagé par tous les outils d'apprentissage de signature.

Il existe en effet un certain nombre d'outils couplant apprentissage de signature de séquences nucléiques et recherche de signature. On peut par exemple citer la boîte à outils Infernal [ED94] qui utilise les grammaires hors-contexte probabilistes (cf. paragraphe 2.2.2.1). Elle contient notamment `cmbuild`, qui construit une signature nucléique sous forme de grammaires hors-contexte probabiliste, et `cmsearch` qui recherche une occurrence d'une grammaire. Erpin [GL01, LG03, LLG05, LLFG05] génère aussi une signature utilisant des chaînes de Markov cachées à partir d'un alignement. Il est ensuite possible d'utiliser ces signatures pour des recherches d'ARN. Ces outils ont plusieurs inconvénients. Ils sont tout d'abord relativement lents (c'est particulièrement vrai pour les formalismes utilisant les grammaires hors-contextes probabilistes), ils ne peuvent pas modéliser de duplex, et surtout il est extrêmement difficile d'en modifier les signatures. Même si, dans le cas de Infernal, la signature est située dans un fichier et modifiable à la main, l'accumulation de paramètres dans la signature fait qu'en pratique une telle manipulation est très délicate.

L'intérêt de notre approche est que la signature apprise par notre outil sera facilement modifiable par l'utilisateur. En d'autres termes, notre approche ne se substitue pas à l'expert mais l'aide à créer une première version de la signature, avec des séquences alignées qui sont souvent accessibles *via* le Web.

Le reste du chapitre s'articule de la manière suivante. Nous commencerons par un exemple, celui de l'ARN de transfert. Nous verrons comment il est possible d'apprendre la signature de cet ARN à partir d'un alignement. Nous ferons ensuite un rapide état de l'art sur l'apprentissage de signatures et sur l'apprentissage dans les réseaux de contraintes. Cela fait, nous exposerons notre approche et nous détaillerons les choix que nous avons faits lors de l'apprentissage de signatures.

6.2 Exemple de l'ARN de transfert

Pour comprendre les enjeux de l'apprentissage de signature, nous commencerons par un exemple simple. Considérons l'alignement de la figure 6.1. Celui-ci contient plusieurs séquences d'ARN de transfert modifiés pour les besoins de notre exemple, ainsi que leur structure secondaire commune. Essayons d'en dégager une signature.

On peut tout d'abord commencer par voir s'il existe des mots conservés. L'aligne-

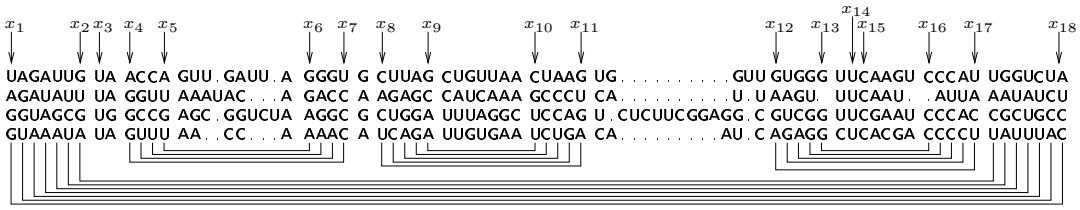


FIG. 6.2 – Positionnement des variables.

ment ne contenant que quatre séquences, nous considèrerons qu'un mot n'est conservé que s'il apparaît dans toutes les séquences. Il n'en existe que deux : le nucléotide U situé sous le premier rectangle placé au-dessus de l'alignement, et le mot UC situé sous le second rectangle.

On peut ensuite rechercher les hélices dans l'ensemble des interactions. Il y en a quatre. Aucune ne comporte d'insertion. Après examen, on peut voir que la première hélice, celle qui englobe les autres, contient sept paires de bases, une boucle de cinquante-deux à soixante-sept nucléotides et aucun mésappariement. La deuxième hélice contient quatre paires de bases, une boucle de cinq à neuf nucléotides et éventuellement un mésappariement (apparaissant dans la première et dans la troisième séquence). La troisième hélice contient cinq paires de bases, une boucle de sept nucléotides et éventuellement un mésappariement (dans la deuxième séquence). La quatrième hélice contient quatre à cinq paires de bases, une boucle de six à sept nucléotides et éventuellement un mésappariement (troisième et quatrième séquences).

On peut alors ajouter des variables pour positionner les éléments de structure que l'on a trouvés. Si l'on numérote les variables dans le sens $5'$ vers $3'$ (c'est-à-dire en lisant la séquence de la gauche vers la droite), la première hélice utilisera les variables x_1 et x_2 pour délimiter le premier brin et x_{18} et x_{19} pour délimiter le second brin. Le mot conservé U sera pointé à la position x_3. La deuxième hélice utilisera les variables x_4 à x_7. La troisième hélice utilisera les variables x_8 à x_{11}. La quatrième hélice utilisera les variables x_{12} et x_{13} pour le premier brin et x_{16} et x_{17} pour le second brin. Le mot conservé UC sera encadré par les positions x_{14} et x_{15}. La figure 6.2 représente le placement de ces variables sur l'alignement.

Il reste ensuite quelques espaceurs à ajouter. Le mot conservé U apparaît juste après la fin du premier brin de l'hélice et il y a un nucléotide entre le mot U et le début de la deuxième hélice. Il y a un seul nucléotide entre la deuxième et troisième hélice et il y a quatre à treize nucléotides entre les deux dernières hélices. Le second brin de la première hélice est placé juste après la quatrième hélice. Enfin, le mot UC commence à partir du second nucléotide dans la boucle de la quatrième hélice.

Pour conclure, on peut exprimer des préférences. On peut par exemple tout d'abord préférer que les hélices n'aient pas d'erreur. On peut aussi préférer que la distance entre la troisième hélice et la quatrième hélice soit inférieure à cinq nucléotides et que les longues distances soient pénalisées par un coût de un.

En somme, cela nous donne la signature suivante :

– $\texttt{hélice}[\mathsf{taille} = 7, \mathsf{boucle} = 52..67, \top_{\mathrm{loc}} = 0, \mathsf{transformateur} = \mathsf{dur}](x_1, x_2, x_{17}, x_{18})$,

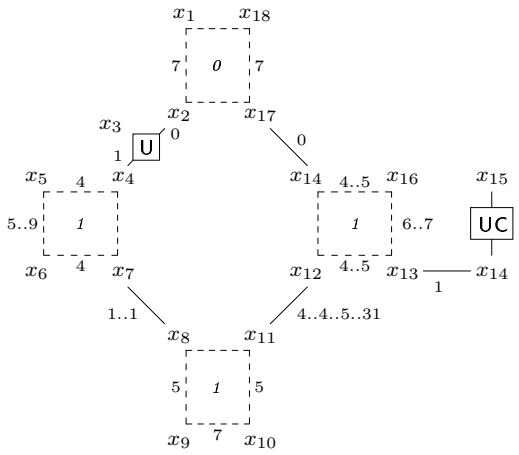


FIG. 6.3 – Le réseau de contraintes pondérées généré par l'apprentissage de signature.

- espaceur[distance $= 0$, transformateur = dur](x_2, x_3),
- mot[mot = U, $\top_{\text{loc}} = 0$, transformateur = dur](x_3, x_3),
- espaceur[distance $= 1$, transformateur = dur](x_3, x_4),
- hélice[taille $= 4$, boucle $= 5..9$, $\top_{\text{loc}} = 1$, transformateur = souple](x_4, x_5, x_6, x_7),
- espaceur[distance $= 1$, transformateur = dur](x_7, x_8),
- hélice[taille $= 5$, boucle $= 7$, $\top_{\text{loc}} = 1$, transformateur = souple]$(x_8, x_9, x_{10}, x_{11})$,
- espaceur[distance $= 4..4..5..13$, coût $= 0..1$, transformateur = souple](x_{11}, x_{12}),
- hélice[taille $= 4..5$, boucle $= 6..7$, $\top_{\text{loc}} = 1$, transformateur = souple]$(x_{12}, x_{13}, x_{16}, x_{17})$,
- espaceur[distance $= 1$, transformateur = dur](x_{14}, x_{15}),
- mot[mot = UC, $\top_{\text{loc}} = 0$, transformateur = dur](x_{15}, x_{16}).

Rappelons que la valeur $\top_{\text{loc}}$ correspond au plus grand nombre d'erreurs acceptées et que le paramètre transformateur permet de spécifier l'interprétation de ce paramètre (cf. paragraphe 3.4.4).

La valeur $\top$ est finalement fixée à deux, le plus petit coût acceptant toutes les séquences données dans l'alignement.

Le réseau de contraintes pondérées peut alors être créé ; il aura la forme donnée dans la figure 6.3. Les traits pleins représentent les fonctions de coût d'espacement et les intervalles qui étiquettent ces traits sont les distances données en paramètres de la fonction. Les rectangles pleins sont les mots. Les rectangles pointillés sont les fonctions de coût d'hélice. Les nombres en italique à l'intérieur de ces rectangles sont les nombres maximum d'erreurs autorisées et les intervalles placés à proximité des arêtes sont les tailles d'hélice ou de boucles.

Nous avons finalement appris et généré notre signature.

6.3 État de l'art

Faisons maintenant un bref état de l'art des outils permettant de combiner apprentissage et recherche de signatures. Nous étudierons d'abord les outils bio-informatiques de recherche de signatures, puis les approches basées sur les contraintes, éventuellement pondérées.

6.3.1 Outils de recherche de signatures

Il existe un certain nombre d'outils couplant apprentissage et recherche de signatures. Le premier est le couple Pratt [JCH95, Jon97] / Prosite [SCH+02b, dCSG+06] : le premier outil permet de générer une signature utilisant des grammaires régulières (cf. paragraphe 2.2.1.1) sur des séquences protéiques et le second permet de rechercher les occurrences de cette signature. À partir d'un alignement, Pratt génère des signatures simples et compréhensibles. Dans la pratique, les signatures sont de la forme :

```
C-x(2)-[DE]-x(1,2)-F
```

La signature signifie que l'on cherche un acide aminé `C`, suivi de deux autres acides aminés (n'importe lequels), puis l'acide `D` ou l'acide `E`, puis un ou deux acides et enfin l'acide aminé `F`. La signature est donc facilement modifiable et l'utilisateur peut simplement ajouter des acides à la signature, en enlever, assouplir les contraintes de distance (comme remplacer `x(2)` par `x(2,3)`), etc.

Pour rechercher des protéines, on peut aussi utiliser la boîte à outils HMMER (dont le principe est donné dans [DEKM98]). HMMER contient deux logiciels principaux, `hmmbuild` et `hmmsearch`. La boîte à outils se base sur les chaînes de Markov cachées (cf. paragraphe 2.2.2.1). `hmmbuild` construit une chaîne de Markov cachée maximisant la vraisemblance de l'alignement donné et `hmmsearch` permet de rechercher une occurrence de la chaîne dans une nouvelle séquence. Mais cette signature, même si elle est modifiable car stockée dans un fichier plat, reste très difficilement manipulable, tant le nombre de paramètres qu'elle fait intervenir est grand. Il en est de même pour le programme Erpin [GL01, LG03, LLG05, LLFG05], qui utilise aussi les chaînes de Markov cachées dans ses signatures.

Le nombre de paramètres est encore plus grand dans Infernal [ED94], qui reprend le même principe que HMMER mais pour les séquences nucléiques et en utilisant les grammaires hors-contexte probabilistes (cf. paragraphe 2.2.2.1).

6.3.2 Approches basées sur les contraintes

Quelques méthodes ont aussi été développées dans le domaine des contraintes (cf. 3.3.1.1 pour plus de précision sur les réseaux de contraintes) pour faire de l'apprentissage. Certains travaux [CBO+03, BCFO04, BCKO05, BCOP07] supposent connus l'ensemble des variables du problème, le domaine de chaque variable, la structure du réseau et le type de contraintes utilisables. Le problème consiste ici à placer les contraintes sur le réseau. L'apprentissage se base ici sur des exemples et des contre-exemples. De

plus, pour affiner l'apprentissage, l'approche propose des affectations que l'utilisateur peut classer en solutions ou non-solutions. Les auteurs se basent ici sur la technique de l'*espaces de version* [Mit82] et SAT.

D'autres travaux [BCP05, BCP07] se proposent d'apprendre les paramètres de certaines contraintes dites implicites alors que le réseau de contraintes est déjà connu. Les contraintes implicites d'un problème sont des contraintes qui n'ont pas été formulées explicitement, mais qui, lorsqu'on les ajoute au réseau, ne diminuent pas l'espace des solutions. Ces contraintes implicites ont souvent une sémantique bien définie, comme la contrainte de cardinalité `Gcc` [Rég96], qui peut être exploitée par le solveur pour résoudre le problème plus rapidement. Les auteurs proposent ici, en faisant quelques suppositions sur la sémantique de la contrainte implicite, une méthode pour apprendre ces contraintes en temps limité.

D'autres travaux encore ont été appliqués aux réseaux de contraintes valuées (cf. 3.4.2.1 pour plus de précision sur les réseaux de contraintes valuées). [RSV$^+$02] propose par exemple d'estimer les paramètres d'un réseau de contraintes valuées temporels simples où seules les valuations sont inconnues à l'aide d'un ensemble d'entraînement. Les auteurs utilisent ici une méthode de descente de gradient [Hay94] visant à trouver les coûts des fonctions donnant les solutions les plus proches possibles de l'ensemble d'entraînement.

Enfin, dans [RS04], les auteurs proposent une méthode interactive d'apprentissage de valuation de contraintes valuées. Comme dans le cas précédent, le réseau est supposé connu et seules les valuations sont à estimer. La méthode part d'un réseau de contraintes valuées fixé et de valuations *a priori*. Elle propose à l'utilisateur les solutions du problème et si elles ne le satisfont pas, il est invité à rectifier les valuations du problème.

Dans notre cas, aucune des solutions proposées ne convient. Notre problème n'est en effet pas simple :

- le nombre de variables n'est pas connu,
- le domaine de chaque variable dépend de la séquence d'entrée,
- l'arité des fonctions de coût est variable et peut aller jusqu'à quatre,
- les fonctions de coût sont toutes paramétrées et peuvent avoir plusieurs paramètres,
- les scores donnés par les fonctions sont eux aussi à apprendre.

Supposer par exemple que l'on connaisse le nombre de variables de notre problème implique que l'on connaisse les éléments de structure. Et si l'on connaît les éléments de structure, on connaît aussi les fonctions de coût. De plus, il est à la limite possible de fournir des contre-exemples (notamment en donnant des séquences aléatoires) mais nous ne voulons pas demander à l'utilisateur si un candidat appartient ou non à la famille d'ARN non-codants recherchée. En revanche, pour estimer la valeur des paramètres, la méthode de descente de gradient est tout à fait acceptable, même si nous avons préféré estimer directement nos paramètres.

6.4 Apprentissage de signatures

6.4.1 Introduction

L'approche heuristique préliminaire que nous avons développée est la suivante. Nous ne modéliserons ici que des mots, des hélices et des espaceurs. Ceux-ci permettent de modéliser dans la pratique un grand nombre de signatures. De plus, seules des fonctions de coût de type dur ou souple seront utilisées (cf. paragraphe 3.4.4 pour les différentes modélisations d'un élément de structure). Ceci implique que les fonctions de coût de type optionnel ne seront pas générées lors de l'apprentissage. Enfin, puisque chaque fonction de type souple peut être décomposée en une fonction de type dur (une partie dure) et optionnel (une partie souple), les fonctions de ce type vont être apprises en deux temps : tout d'abord la partie dure, puis la partie souple.

Dans la pratique, nous apprendrons en premier la structure du réseau. Cette structure comprend les variables, le nombre de fonctions de mots, hélices et espaceurs que nous allons utiliser, et la portée de chaque fonction.

Dans un second temps, nous apprendrons les paramètres des fonctions de coût. Nous commencerons par les parties dures, de type dur. Ces paramètres seront fixés pour que la sensibilité soit de 100 % et que la spécificité soit la plus haute possible. En d'autres termes, nous choisirons les valeurs des paramètres les plus strictes possibles, telles que toutes les séquences données dans l'alignement soient acceptées par le réseau.

Nous apprendrons ensuite les coûts des fonctions de type souple, dont la partie dure a déjà été apprise à l'étape précédente. Nous nous inspirerons ici d'un cadre probabiliste. Nous supposerons que chaque fonction de coût représente un potentiel, comme dans les champs de Markov [GG84, DJ89]. Plus spécifiquement encore, pour les hélices et les mots, la fonction de coût générée f décrira une distribution de probabilité sur les mots formant l'élément de structure, du type :

$$P(\text{mot}) = K\mathrm{e}^{-f(\text{mot})}$$

où K est un facteur de normalisation. En fait, nous ne chercherons pas à calculer ce facteur de normalisation car notre formalisme recherche les valeurs de chaque variable qui maximisent la probabilité jointe. Les solutions sont donc indépendantes des facteurs constants de normalisation. On suppose de plus que pour chaque élément cette probabilité ne dépend que du nombre d'erreurs par rapport à une structure parfaite de référence (qui est le mot donné en paramètres pour la fonction de coût de mot, et une hélice sans erreur pour la fonction d'hélice). Nous ferons encore la supposition que toutes les hélices ou mots à e erreurs peuvent être observés de façon équiprobable, ce qui entraîne, nous le verrons dans les paragraphes suivants, que la recherche de la distribution de probabilité se réduit à un problème de dénombrement. Enfin, chaque distribution sera estimée de façon indépendante, ce qui est une approximation, puisque les éléments de structure peuvent se chevaucher. Concernant les espaceurs, les paramètres seront estimés directement, à partir d'une distribution gaussienne des distances.

La modélisation des fonctions de coût par des potentiels permet d'ailleurs de donner un sens aux coûts que l'on utilise. Ils traduisent une adéquation sous un modèle donné

par rapport à un jeu de données observé. Et si, comme dans notre cas, chaque potentiel est effectivement l'opposé d'un logarithme de probabilité, trouver toutes les solutions ayant un score inférieur à $\top$ revient à trouver toutes les régions ayant une probabilité supérieure à $e^{-\top}$ d'appartenir à la famille d'ARN donnée.

Dans la suite de ce chapitre, nous détaillerons les choix faits pour estimer la structure et les paramètres nécessaires pour décrire une famille d'ARN donné.

6.4.2 Apprentissage de la structure

La première partie de l'apprentissage consiste à reconnaître la structure du réseau que nous génèrerons, c'est-à-dire les variables, les fonctions de coût à choisir parmi les mots, les hélices et les distances, ainsi que la portée de chaque fonction.

6.4.2.1 Mots

Le mieux serait de créer une contrainte seulement pour les mots qui discriminent entre vrais et faux positifs. En effet, générer beaucoup de contraintes de mots, contenant par exemple un mot-consensus petit, et avec un grand nombre d'erreurs autorisées, ralentit la recherche, sans pour autant la guider vers les vrais positifs.

Nous procédons alors en deux passes pour reconnaître les mots. Tout d'abord, nous recherchons les mots conservés à 100 %, contenant éventuellement des nucléotides ambigus. Pour cela, nous recherchons tous les nucléotides conservés à 100 %. Ils constituent des ancres pour des fonctions de coût de mot. Chaque ancre est étendue à gauche et à droite, si l'on observe des nucléotides conservés à 100 %. Deux mots conservés à 100 % sont fusionnés s'ils sont séparés par des colonnes contenant au plus deux nucléotides différents et pas de brèche.

La seconde passe procède de manière similaire, pour des nucléotides relativement conservés. Une ancre est ici un nucléotide conservé à plus de 80 %. Il est étendu à gauche et à droite s'il existe d'autres nucléotides conservés à plus de 80 %, et deux mots relativement conservés peuvent être fusionnés s'ils se situent à moins de deux nucléotides l'un de l'autre.

En fait, il serait intéressant de trouver les mots significativement discriminants, en s'appuyant par exemple sur les travaux de comptage de mots [NSF02]. Certains mots fortement dégénérés que nous trouvons n'aident sans doute pas à la recherche des solutions, ne faisant que ralentir les algorithmes. On pourrait aussi utiliser un modèle plus fin pour ces fonctions de coût de mot. On peut par exemple penser ici aux matrices de poids positions-spécifiques et aux chaines de Markov cachées.

6.4.2.2 Hélices

Les hélices sont ensuite formées. Celles-ci sont constituées directement à partir des informations de structure du fichier d'entrée. Les hélices sont créées à partir des ensembles d'interactions contiguës. Les renflements parfois observés dans les hélices créent donc, dans notre modèle, deux hélices distinctes.

Nous avons aussi implanté une version qui accepterait les renflements de taille limitée mais nous avons préféré désactiver cette fonctionnalité dans un premier temps. En effet, ne pas autoriser les insertions et les suppressions accélère grandement la recherche des solutions.

6.4.2.3 Distances et variables

Chaque hélice est ensuite divisée en deux brins. Les mots et brins sont ensuite classés par position de départ croissante. Des variables sont créées aux positions de début et de fin des éléments de structure et les distances sont générées entre deux différents éléments de structure consécutifs. Les longueurs minimales et maximales données en paramètres à la contrainte sont les longueurs minimales et maximales observées.

Cette génération des variables est parfois simpliste. On peut parfois avoir deux variables pointant sur la même position sur l'alignement. Des travaux futurs tenteront de corriger ce problème.

6.4.3 Apprentissage des paramètres

Les paramètres des fonctions de coût sont ensuite apprises. Nous avons vu qu'il existait deux types de fonctions : les fonctions dures, de type dur, et les fonctions souples, de type souple.

6.4.3.1 Apprentissage des paramètres des fonctions de type dur

Les paramètres des fonctions du cœur dur sont choisis de manière à créer une signature la plus stricte possible, acceptant toutes les séquences de l'alignement d'entrée.

Mots Dans chaque fonction de coût de mot, le mot-consensus et le nombre maximal d'erreurs sont à estimer. Nous avons généré deux types de mots : les premiers ont été créés à partir d'ancres conservées à 100 %, et les seconds, à partir d'ancres conservées à 80 %.

Pour la première série de mots, le mot-consensus est formé de la manière suivante. Pour chaque colonne où apparaît le mot, on sélectionne un nucléotide. Les colonnes contenant des nucléotides conservés à 100 % donnent ce nucléotide, et les colonnes où il existe deux nucléotides différents donnent le nucléotide ambigu correspondant à ces deux nucléotides. Le nombre maximum d'erreurs autorisées est zéro.

Concernant la seconde série de mots, on procède de même. Chaque colonne où apparaît le mot conservé donne un nucléotide. Si un nucléotide est conservé à plus de 80 %, ce nucléotide est choisi. S'il n'existe que deux nucléotides dans la colonne, alors on choisi le nucléotide ambigu correspondant à ces deux nucléotides. Sinon, on choisit le nucléotide N. Le nombre maximum d'erreurs autorisées est le nombre maximum d'erreurs observées dans l'alignement. Toutefois, si le nombre d'erreurs maximum est supérieur à la moitié du nombre de nucléotides non-ambigus, cette fonction de coût est jugée trop peu informative et elle est supprimée.

Hélices Pour chaque hélice, il faut estimer les tailles minimales et maximales de l'hélice et de sa boucle et le nombre maximum d'erreurs autorisées.

Les tailles sont estimées par les tailles minimales ou maximales des hélices et des boucles observées sur l'alignement. Le nombre d'erreurs maximum pour l'hélice est donné par le nombre maximum d'erreurs observées dans les alignements pour cette hélice.

Si l'on s'arrête à ces directives, alors de nombreuses hélices de taille réduite (souvent de taille un) apparaissent dans la signature. De plus, certaines hélices contiennent un nombre d'erreurs maximales tolérées de l'ordre de la taille de ces hélices, ce qui rend en pratique la contrainte inutile. Nous avons alors décidé d'enlever toutes les hélices dont le nombre maximum d'erreurs était supérieur ou égal à la taille de l'hélice, moins deux. Nous imposons de plus que les hélices de taille un aient un nombre d'erreurs nul et que la taille de la boucle soit raisonnablement petite, inférieure à cinquante nucléotides.

Espaceurs Les tailles minimales et maximales des espaceurs sont estimées par les tailles minimales et maximales observées dans l'alignement.

6.4.3.2 Apprentissage des paramètres des fonctions de type dur

Une fonction de coût est éventuellement ajoutée à chaque contrainte dure. Les modélisations suivantes sont très approximatives, du fait du peu de temps investi dans cette thématique.

Mots Si le nombre maximum d'erreurs autorisées dans une contrainte dure de mot est strictement positif, alors une fonction de coût de mot est ajoutée à la contrainte. Le mot-consensus choisi est le même que celui de la contrainte.

Le score attribué pour chaque nombre d'erreurs observées est calculé à partir de la probabilité d'observer un mot donné sous le modèle suivant. Notons donc C le mot consensus et M un autre mot. La transformation de C en M est notée $C \rightarrow M$, et le nombre d'erreurs de cette transformation, $n(C \rightarrow M)$. Le score que nous cherchons à donner est une fonction s qui, en fonction de M, donne l'opposé du logarithme de la probabilité $P(C \rightarrow M)$ d'observer une transformation de C en ce mot M. L'égalité suivante traduit que la probabilité d'observer une transformation de C en M dépend du nombre d'erreurs e de la transformation, déductible de C et de M.

$$P(C \rightarrow M) = P(C \rightarrow M | n(C \rightarrow M) = e).P(n(C \rightarrow M) = e))$$

Nous supposerons ici que la distribution du nombre d'erreurs dans un mot suit une loi multinomiale à $\top_{\text{loc}} + 1$ résultats possibles. On peut alors estimer le terme $P(n(C \rightarrow M) = e)$ par $\hat{p}_e$, la probabilité empirique d'avoir e erreurs, qui est un estimateur de maximum de vraisemblance [Kay93].

Il nous reste donc à estimer le second terme. Nous avons fait l'hypothèse que tous les mots à une distance e de C peuvent se trouver de façon équiprobable. Il faut donc compter le nombre $a(e)$ de tous ces mots. Ce nombre ne dépend que de la taille l

du mot M et du nombre de lettres σ de l'alphabet. En première approximation, nous supposerons que l'ensemble des mots ayant e erreurs avec C peut être généré à partir de C en lui substituant, supprimant ou insérant des lettres. Une substitution peut transformer une lettre en $\sigma - 1$ autres lettres. Une suppression peut être considérée comme une substitution d'une lettre en ε, une lettre vide. L'ensemble des e erreurs peut donc être divisée entre i insertions d'un côté et $e - i$ suppressions / substitutions de l'autre. Nous appelerons donc $a_{\text{sub/sup}}$ le nombre de substitutions et suppressions que l'on a, et a_{ins} et le nombre d'insertions.

$$\begin{aligned} a(e) &= \sum_{i=0}^{e} a_{\text{sub/sup}}(e-i) \times a_{\text{ins}}(i) \\ a_{\text{sub/sup}}(e-i) &= \binom{e-i}{l}\sigma^{e-i} \\ a_{\text{ins}}(i) &= (l+1)^i\sigma^i \end{aligned}$$

Le nombre de substitutions et de suppressions peut être généré en sélectionnant $e - i$ lettres puis en substituant chacune des lettres sélectionnées par une des autres $\sigma - 1$ lettres de l'alphabet ou ε. Le nombre d'insertions peut être calculé en trouvant i places pour y insérer des lettres. Pour chaque lettre, il existe $l + 1$ places : $l - 1$ places entre deux lettres, une en début du mot, une à la fin du mot. Il est bien évidemment possible de choisir plusieurs fois le même intervalle.

La formule n'est pas exacte, dans la mesure où une insertion ne devrait pas être placée à côté d'une suppression ; il s'agit plutôt d'une substitution.

Nous pouvons maintenant donner une fonction de coût ne dépendant que du nombre d'erreurs e observées sur un mot :

$$P(C \to M) = \hat{p}_e . \frac{1}{\sum_{i=0}^{e} \binom{e-i}{l}\sigma^{e-i}(l+1)^i\sigma^i}$$

Nous définissons le score de la fonction de mot comme l'opposé du logarithme de la probabilité d'apparition du mot observé, qui ne dépend que du nombre d'erreurs e. Il est :

$$s(e) = -\ln(\hat{p}_e) + e\ln(\sigma) + \ln\left(\sum_{i=0}^{e}\binom{e-i}{l}(l+1)^i\right)$$

À titre d'exemple, la courbe $e \mapsto s(e)$ est donnée dans la figure 6.4, pour un mot de six nucléotides, avec deux erreurs acceptées. La proportion de mots sans erreur observé est de 10 %, la proportion de mots à une erreur est de 60 % et la proportion de mots à deux erreurs est de 30 %.

Hélices Si la contrainte dure d'hélice accepte des erreurs, une fonction de coût est ajoutée. Les tailles d'hélice et de boucle sont les mêmes que pour la contrainte dure. Restent à trouver les scores à donner. Nous nous sommes encore basé sur la probabilité

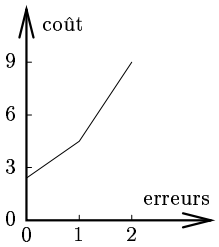


FIG. 6.4 – Fonction de coût pour un mot.

de trouver une hélice. Nous exprimons que la probabilité de trouver une hélice H dépend du nombre d'erreurs e de l'hélice :

$$P(H) = P(H|erreurs(H) = e).P(erreurs(H) = e)$$

Nous supposons ici aussi que la probabilité que le nombre d'erreurs de H soit e est donnée par la loi multinomiale à $\top_{\text{loc}} + 1$ valeurs possibles. Nous supposerons de plus que les hélices à e erreurs peuvent se trouver de façon équiprobable. Il reste donc à estimer le nombre total d'hélices à e erreurs.

Nous faisons alors les hypothèses simplificatrices suivantes. Nous supposons tout d'abord que les hélices parfaites n'ont pas de liaison *wobble*. Cela entraîne que lorsqu'il n'y a pas d'erreur et que l'une des branches de l'hélice est connue, alors l'autre l'est aussi. Nous allons donc considérer qu'une hélice avec e erreurs est composée d'un brin de taille l_1 à l_2 (ce qui fait $\sum_{i=l_1}^{l_2} \sigma^i$ possibilités) et d'un autre brin, obtenu avec e substitutions, suppressions ou insertions. Le nombre $a(e)$ d'hélices à e erreurs possibles est donc accessible en générant tous les mots de longueur l_1 à l_2 — ce qui constituera le premier brin — et en y ajoutant e erreurs pour l'autre brin. On utilise alors une formule similaire à la formule donnée pour les mots :

$$\begin{aligned} a(e) &= \sum_{i=l_1}^{l_2} \sigma^i \sum_{j=0}^{e} \binom{e-j}{i} \sigma^{e-j} \times (i+1)^j \sigma^j \\ &= \sigma^e \sum_{i=l_1}^{l_2} \sigma^i \sum_{j=0}^{e} \binom{e-j}{i} (i+1)^j \end{aligned}$$

On a donc :

$$P(H) = \hat{p}_e . \frac{1}{\sigma^e \sum_{i=l_1}^{l_2} \sigma^i \sum_{j=0}^{e} \binom{e-j}{i} (i+1)^j}$$

Comme pour la fonction de mot, le score renvoyé est l'opposé du logarithme de la probabilité d'apparition de l'hélice observée, qui ne dépend que du nombre d'erreurs e :

$$s(e) = -\ln(\hat{p}_e) + e\ln(\sigma) + \ln\left(\sum_{i=l_1}^{l_2} \sigma^i \sum_{j=0}^{e} \binom{e-j}{i} (i+1)^j\right)$$

Le modèle présenté ici comporte plusieurs inconvénients. En plus d'être extrêmement approximatif, il pénalise de la même manière les hélices longues et les hélices courtes. Il aurait été préférable de donner un score légèrement plus faible aux hélices longues, en vue de privilégier celles-ci.

Distances Pour chaque fonction de coût de distance, il faut estimer plusieurs paramètres : les quatre distances, ainsi que les scores minimum et maximum de la fonction de coût. Nous choisirons d'approximer la fonction de distribution des longueurs de la manière suivante. Nous supposerons que la distribution est gaussienne. Nous approximerons ensuite cette gaussienne par un trapèze. Ceci nous donnera quatre points : les quatre sommets du trapèze. Nous prendrons ensuite l'opposé du logarithme de la probabilité comme score, ce qui revient à faire l'approximation que les côtés du trapèze ont une forme exponentielle.

On commence donc par rechercher la moyenne et l'écart-type de la distribution de distance, afin de trouver la meilleure gaussienne possible. Si la variance est nulle, alors la fonction de coût n'est pas générée, puisqu'elle est inutile. Nous avons ensuite interpolé la gaussienne par cinq segments, de c_1 à c_5, comme représenté sur la figure 6.5. Les segments c_1, c_3 et c_5 sont horizontaux. c_1 et c_5 sont d'abscisse nulle, c_3 est au maximum de la gaussienne. c_1 est en fait une demi-droite, partant à l'abscisse $-\infty$ et s'arrêtant en d_1. Les abscisses de c_2, c_3 et c_4 sont respectivement comprises entre d_1 et d_2, d_2 et d_3, d_3 et d_4. c_5 est la demi-droite partant à l'abscisse d_4 et s'arrêtant en $+\infty$. c_2 et c_4 sont approximées par les deux tangentes à mi-hauteur de la courbe. Les abscisses d_1 à d_4 sont donc trouvées comme les points de rencontre entre les différentes demi-droites ou segments.

Les calculs suivants donnent les points de rencontre recherchés.

La gaussienne a la forme suivante, où m est la moyenne et σ l'écart-type :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-\frac{1}{2}\left(\frac{x-m}{\sigma}\right)^2}$$

La hauteur maximale est donc atteinte pour $x = m$:

$$\frac{1}{\sigma\sqrt{2\pi}}$$

À mi-hauteur, l'ordonnée est donc :

$$\frac{1}{2\sigma\sqrt{2\pi}}$$

Les abscisses à mi-hauteur sont donc :

$$x_{\frac{1}{2}} = m \pm \sigma\sqrt{2\ln(2)}$$

La dérivée en ces points vaut alors :

$$f'(x_{\frac{1}{2}}) = \pm\frac{\sqrt{\ln(2)}}{2\sigma^2\sqrt{\pi}}$$

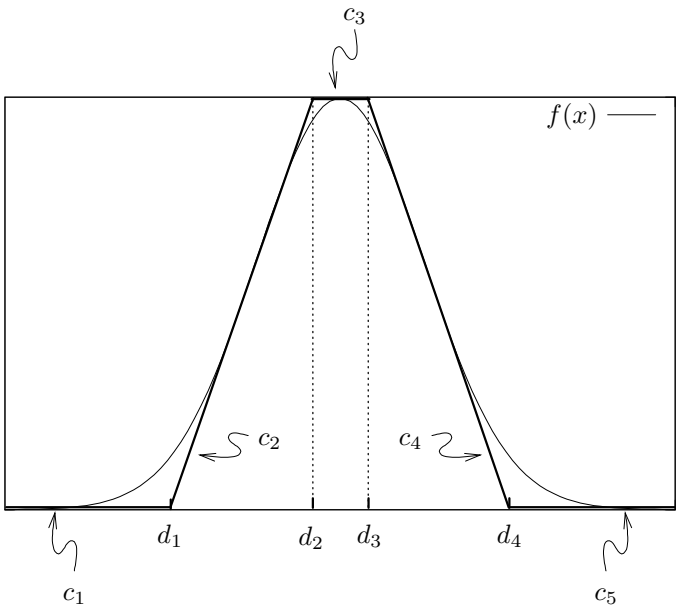


FIG. 6.5 – Approximation d'une gaussienne par un trapèze.

Les équations de droite de c_2 et c_4 sont donc de la forme :

$$c_2(x) = \frac{\sqrt{\ln(2)}}{2\sigma^2\sqrt{\pi}}x + b_2$$

et

$$c_4(x) = -\frac{\sqrt{\ln(2)}}{2\sigma^2\sqrt{\pi}}x + b_4$$

où b_2 et b_4 sont à déterminer. Connaissant la pente de la droite et un point de celle-ci, on trouve :

$$b_2 = \frac{\sigma + \sqrt{2\ln(2)}\left(\sigma\sqrt{2\ln(2)} - m\right)}{2\sigma^2\sqrt{2\ln(2)}}$$

et

$$b_4 = \frac{\sigma + \sqrt{2\ln(2)}\left(\sigma\sqrt{2\ln(2)} + m\right)}{2\sigma^2\sqrt{2\ln(2)}}$$

Une fois obtenues les équations de droite, il est maintenant possible de déterminer leur point de rencontre. Le point de rencontre entre c_1 et c_2, dont l'ordonnée est nulle, détermine la valeur d_1 :

$$d_1 = \frac{2m\sqrt{\ln(2)} - 2\sigma\ln(2)\sqrt{2} - \sigma\sqrt{2}}{2\sqrt{\ln(2)}}$$

Similairement, le point de rencontre entre c_4 et c_5 détermine d_4 :

$$d_4 = \frac{2m\sqrt{\ln(2)} + 2\sigma\ln(2)\sqrt{2} + \sigma\sqrt{2}}{2\sqrt{\ln(2)}}$$

Le point de rencontre entre c_2 et c_3 a pour ordonnée le maximum de la fonction. Son abscisse détermine d_2 :

$$d_2 = \frac{2\sigma\sqrt{2} + 2m\sqrt{\ln(2)} - 2\sigma\ln(2)\sqrt{2} - \sigma\sqrt{2}}{2\sqrt{\ln(2)}}$$

Le point de rencontre de c_3 et c_4 détermine d_3 :

$$d_3 = \frac{-2\sigma\sqrt{2} + 2m\sqrt{\ln(2)} + 2\sigma\ln(2)\sqrt{2} + \sigma\sqrt{2}}{2\sqrt{\ln(2)}}$$

Il est possible que c_2 et c_4 se coupent. Dans ce cas, c_3 n'est pas définie. Leur point de rencontre se fait en :

$$\begin{aligned} x &= m \\ y &= \frac{2\ln(2)\sqrt{2} + \sqrt{2}}{4\sigma\sqrt{\pi}} \end{aligned}$$

c_3 est alors non définie quand l'ordonnée du point de rencontre entre c_2 et c_4 est inférieur au maximum de la fonction, c'est-à-dire quand :

$$\begin{aligned} \frac{1}{\sigma\sqrt{2\pi}} &\geq \frac{2\ln(2)\sqrt{2} + \sqrt{2}}{4\sigma\sqrt{\pi}} \\ 4 &\geq 4\ln(2) + 2 \end{aligned}$$

c_3 est donc toujours définie puisque $4\ln(2) + 2 \approx 4.772588722$.

Nous avons donc bien des valeurs pour d_1, d_2, d_3, d_4 et le coût minimum de la fonction de coût, qui est l'opposé du logarithme du maximum de la gaussienne. En revanche, le coût maximum n'est toujours pas trouvé. Les demi-droites c_1 et c_5 ont pour ordonnée zéro, ce qui implique que le minimum de la fonction de coût vaut $-\ln(0) = +\infty$. Cette valeur est approximée à $\top$.

Nous procédons donc de la manière suivante :

1. Les valeurs d_1 à d_4 sont estimées, ainsi que le minimum de la fonction.
2. On calcule la somme des coûts des mots et des hélices pour chaque séquence observée. Le maximum sera notre valeur de $\top$.
3. Les maxima de toutes les fonctions de coût de distance sont approximés à la valeur de $\top$ précédemment trouvée.
4. Afin d'assurer une sensibilité de 100 %, on recalcule la valeur de $\top$, en prenant cette fois-ci en compte les distances.

Cette façon de procéder n'est pas très satisfaisante. Une meilleure solution consisterait à se donner un seuil de 95 % par exemple, ce qui permettrait d'attribuer à c_1 et c_5 des ordonnées non nulles et de ne pas réestimer la valeur de $\top$. De plus, les distances d_1 à d_4 sont estimées par cette gaussienne qui est une fonction symétrique, alors que la distribution des distances n'a que peu de raisons de l'être. Enfin, la gaussienne n'est normalement utilisable que dans le cas continu. Une loi multinomiale serait ici plus adaptée. Et si l'on veut mieux faire, il serait plus pertinent d'estimer directement le trapèze par maximum de vraisemblance.

6.5 Conclusion

L'apprentissage de signatures est une fonctionnalité intéressante. À partir d'un alignement, elle permet la génération d'une première signature que l'utilisateur pourra ensuite affiner, s'il le souhaite.

Nous sommes conscients que le travail de génération de signature présenté ici en est encore à ses balbutiements. Nous avons, dans l'exposé de la méthode suivie, proposé quelques méthodes d'amélioration possible et nous proposerons ici quelques pistes supplémentaires. Tout d'abord, il pourrait être intéressant de ne pas avoir à spécifier obligatoirement la structure secondaire de l'alignement. Si cette structure n'est pas connue, le module d'apprentissage pourrait n'avoir qu'un alignement où il devrait apprendre la structure conservée. On peut même envisager le cas où l'on ne donne pas d'alignement, mais simplement l'ensemble des séquences, et le module d'apprentissage pourrait alors être couplé avec un outil de recherche de structure conservée comme RNAalifold [HFS02].

Il serait aussi intéressant d'apprendre d'autres éléments de structure. Le (G+C)% semble le plus facile mais les interactions non-canoniques peuvent aussi être intégrées. De plus, si l'on modifie la syntaxe de l'alignement, il est aussi possible de spécifier des interactions extra-moléculaires, modélisables par la fonction de coût de duplex.

Nous avons aussi décidé que notre module devait avoir une sensibilité de 100 % et une spécificité la plus forte possible par rapport aux séquences de l'alignement. Cela peut être remis en cause, en diminuant la sensibilité, ou en cherchant une spécificité plus faible. Dans l'idéal, l'utilisateur pourrait avoir en main un réglage qui ajuste la spécificité à ses besoins.

De plus, l'approche décrite ne génère aucune fonction de coût de type `option` (voir le paragraphe 3.4.4 pour plus de détails sur les modélisations des éléments de structure). C'est sans doute dommage et il faudrait trouver un moyen d'exprimer des préférences.

L'apprentissage de la structure du réseau repose sur des règles de décision qui utilisent la valeur de certains paramètres. C'est le cas pour les fonctions de coût de mot, où l'on décide que les nucléotides relativement conservés ont une fréquence d'apparition de plus de 80 %. Il serait intéressant de régler ces paramètres sur de grands jeux de données. Il en est de même de la valeur des coûts des fonctions. Des tests à grande échelle justifiant l'intérêt de ces types de coûts restent à faire. C'est une des raisons pour lesquelles nous considérons ce travail comme une ébauche.

Chapitre 7

Expérimentations

7.1 Introduction

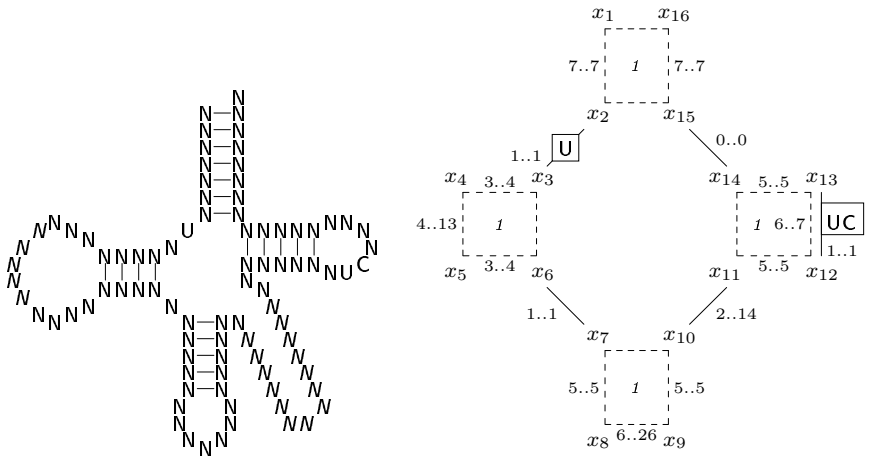
Nous avons exposé dans le chapitre 3 une méthode pour localiser les signatures d'ARN. Cette méthode se base sur les réseaux de contraintes pondérées (cf. paragraphe 3.4.2.1) qui utilisent une propriété de cohérence locale appelée cohérence d'arc aux bornes (cf. définition 4.1.3) pour accélerer la recherche. Le chapitre 4 décrit tous les éléments de structure que nous étions à même de modéliser. Dans le chapitre 5, nous avons donné quelques détails d'implantation de notre approche de recherche de signatures dans un outil appelé DARN !, et le paragraphe 5.4 donne notamment une manière de supprimer un certain nombre de solutions non pertinentes, en sélectionnant les solutions localement optimales. Dans le chapitre 6, nous avons décrit un petit module de génération de signatures de signature, qui, à la lecture d'un alignement, est capable de fournir à DARN ! un descripteur de la famille alignée.

Nous allons donc ici tester ces différentes fonctionnalités. Nous comparerons l'utilisation de la cohérence d'arc aux bornes avec la propriété de cohérence locale sans doute la plus utilisée : la cohérence d'arc. Nous comparerons aussi DARN ! à d'autres logiciels comme tRNAscan-SE [ED94], snoScan [LE99] ou MilPat [TdGSG06]. Nous testerons de plus la sensibilité, la spécificité et la rapidité de DARN !, ainsi que sa capacité à rechercher de longs ARN. La qualité des signatures générées automatiquement sera aussi évaluée. Nous rechercherons aussi de nouveaux candidats de gènes d'ARN. Les expérimentations ont été menées sur des ARN de transfert, des *k-turn*, des petits ARN à boîte H/ACA et C/D, des *riboswitches* et des RNase P.

Les expérimentations suivantes, sauf mention contraire, ont toutes été réalisées sur une machine Intel Xeon à 2,4 GHz avec 8 Go de mémoire vive, utilisant Linux.

7.2 ARN de transfert

Nous commencerons cette série d'expérimentations par la comparaison entre, d'une part, la cohérence d'arc aux bornes (CAB, cf. définition 4.11) et la cohérence ∅-inverse (C∅I, cf. définition 4.12) et, d'autre part, la cohérence locale sans doute la plus classique



(a) Une structure secondaire d'ARNt, où les nucléotides optionnels sont indiqués en italique.

(b) Une signature d'ARNt avec des contraintes et des fonctions de coût. Les intervalles étiquetant les arcs indiquent les intervalles autorisés de la contrainte de distance. Les intervalles près des rectangles en pointillés donnent les tailles possibles de leur hélice ou de leur boucle. Les nombres en italiques situés à l'intérieur de ces rectangles donnent le nombre maximum d'erreurs autorisées dans ces hélices. Seule une erreur est acceptée au total.

FIG. 7.1 – Deux représentations possibles d'un ARN de transfert.

dans les RCP : la cohérence d'arc (CA, cf. définition 4.3). CAB ayant été conçue pour trouver des ARN non-codants, il est logique de tester sa performance sur ce problème. Un jeu de tests classique pour la localisation de motifs d'ARN est la recherche d'ARN de transfert (ARNt). Ceux-ci ont une structure secondaire particulièrement spécifique, contenant quatre hélices en forme de trèfle (cf. figure 7.1(a)).

Nous avons recherché des ARN de transfert, qui peuvent se modéliser avec seize variables, quatre espaceurs, deux mots et quatre hélices. La recherche a été effectuée sur plusieurs sous-séquences du génome d'*Escherichia coli* de différentes tailles. Pour chacune des sous-séquences, nous avons effectué deux résolutions, en utilisant la cohérence d'arc généralisée (CAG) et en utilisant la cohérence d'arc aux bornes généralisée (CABG). Concernant le jeu de tests utilisant CABG, nous avons plus précisément appliqué CABG sur les fonctions de coût de mot et d'hélice, et CABG avec C∅I sur les espaceurs. En effet, la complexité temporelle de CABG avec C∅I est faible pour les fonctions de coût linéaires par morceaux comme la fonction de coût d'espaceur.

Les résultats sont consignés dans le tableau 7.1. Pour chaque instance du problème, la

taille	10k	50k	100k	500k	1M
nb. solutions	0	64	73	92	245
temps (s.) CAG	4	196	2129	-	-
temps (s.) CABG	0	0	0	0	1

TAB. 7.1 – Nombre de nœuds explorés et temps passé à résoudre différentes instances du problème de la localisation d'ARNt.

taille de la séquence génomique est renseignée (10k signifie par exemple que le domaine des variables compte 10 000 valeurs), ainsi que le nombre de solutions à trouver. Le signe « - » signifie que l'instance n'a pas pu être résolue car la mémoire allouée était trop importante.

Comme on le voit, CAG ne peut pas résoudre des instances dont la taille des domaines est supérieure ou égale à 500 000, ce qui rend déjà la propriété inutile en pratique car les séquences à étudier ont une taille plutôt de l'ordre de la dizaine ou de la centaine de millions. Ce problème vient du fait que la représentation des domaines en extension amène la complexité spatiale de CAG à $\mathcal{O}(ed)$ et cela ne permet pas de résoudre de grosses instances. En revanche, le fait de représenter les domaines par des intervalles, et donc simplement par des bornes inférieures et supérieures, réduit considérablement la complexité spatiale et permet de résoudre efficacement les instances de toute taille.

De plus, même pour les « petites » instances (pour des tailles de séquence allant de 10 000 à 100 000), CABG est largement plus rapide que CAG. C'est tout d'abord dû au fait que les fonctions de coût d'espacement ont une sémantique particulièrement bien exploitée par CABG : si une variable est affectée, alors, par une simple propagation de ces fonctions de coût par CABG, toutes les autres variables voient leur domaine se réduire grandement. Ensuite, CAG cherche systématiquement à trouver un support pour chaque valeur de chaque variable, par rapport à chaque fonction de coût, ce qui est particulièrement fastidieux au vu de la taille des domaines. En revanche, CABG ne cherche que les supports des bornes des domaines, et cela se fait rapidement. La recherche se fait en temps linéaire par rapport à la taille de la séquence génomique dans le pire cas, et en temps constant pour les fonctions de coût d'espacement.

Nous avons aussi mené un autre jeu de tests ayant pour but de savoir si DARN! permettait de décrire précisément une signature donnée. Nous avons pour cela recherché les ARNt dans les parties du génome de la levure annotées par RFAM [GJBM+03, GJMM+05], plus particulièrement les 165 534 premiers nucléotides du chromosome 16 de la levure. Nous avons donc construit une signature reconnaissant exactement les ARNt recherchés. La figure 7.1(b) décrit le réseau de contraintes et le descripteur est accessible sur la page Web de DARN![1]. Le lecteur pourra d'ailleurs se référer à l'annexe A pour plus de détails sur la syntaxe utilisée par DARN!. Celle-ci ne change que très peu par rapport la syntaxe précédemment utilisée, si ce n'est que les mots-clefs ont été écrits en anglais et que les contraintes et les préférences sont écrites dans deux parties différentes.

[1]DARN! est accessible à l'adresse http://carlit.toulouse.inra.fr/Darn.

chromosome 16, premiers 165 534 nucléotides			
	temps (s.)	nb. solutions	nb. solutions dominées
DARN !	0,5 s.	5	4

TAB. 7.2 – Résultats de la recherche d'ARNt.

Les ARNt recherchés sont au nombre de cinq et nous avons lancé DARN ! avec notre signature. Les résultats sont donnés dans le tableau 7.2.

On peut en tirer deux observations principales. Tout d'abord, toutes les solutions données par DARN ! sont exactement celles à trouver. Ce point renforce l'idée que la précision des descripteurs de DARN ! est satisfaisante en terme de spécificité. De plus, le fait que le nombre de solutions dominées sur le chromosome 16 soit presque aussi important que le nombre de solutions localement optimales indique que ce mécanisme est particulièrement utile. Il est de plus pertinent, puisqu'il choisit effectivement les bonnes solutions.

À titre de comparaison, nous avons utilisé sur la même séquence tRNAscan-SE [ED94], qui emploie dans une première passe deux programmes pour filtrer les données, puis les grammaires hors-contexte probabilistes (voir le paragraphe 2.2.2.1 pour plus de détails sur ce formalisme). Ce logiciel trouve exactement les mêmes solutions en 0,7 secondes. Le fait que les temps de résolution donnés par les deux approches soient comparables indique que DARN ! est relativement compétitif.

Nous avons aussi voulu nous comparer avec MilPat [TdGSG06] sur la recherche de l'ARN de transfert. Nous avons repris le descripteur dédié à la recherche d'ARNt dans les eucaryotes conçu par P. Thébault et l'avons traduit en un format compréhensible par DARN ! (ce descripteur, qui ne contient que des contraintes dures, est disponible sur le site Web de DARN !). Nous avons effectué la recherche sur tous les chromosomes de la levure. Le but est ici de comparer les deux approches et nous ne tiendrons pas compte de la pertinence du descripteur. Les résultats de l'expérimentation sont consignés dans le tableau 7.3. Les recherches ont ici été menées sur une machine AMD Opteron, cadencée à 2 GHz, et contenant 16 Go de RAM.

Les résultats montrent que MilPat est environ quatre fois plus rapide que DARN ! : MilPat analyse le génome en 2189 secondes alors que DARN ! le fait en 6466 secondes. Mais DARN !, en revanche, propose environ cent soixante-huit fois moins de solutions.

Ce banc de test est relativement difficile pour DARN ! : il ne contient aucune préférence et DARN ! ne peut donc pas couper efficacement l'arbre de recherche. De plus, comme le descripteur est extrêmement peu spécifique, le nombre de solutions est très important. En conséquence, DARN ! passe beaucoup de temps à évaluer la dominance des solutions. Deux éléments le montrent. Tout d'abord, le *profiling* de DARN ! sur ces instances (utilisant gprof) montre que c'est ce mécanisme qui prend le plus de temps (de 12 % à 73 % du temps total passé). De plus, lorsque l'on regarde le chromosome VI, qui est un des chromosomes contenant le moins de solutions, on voit que DARN ! résout l'instance en un temps similaire à MilPat (notre outil passe soixante-seize secondes à résoudre l'instance alors que MilPat en passe soixante-treize).

Les résultats de cette dernière comparaison mettent en évidence le compromis auquel

	MilPat		DARN !	
chromosome	nb. solutions	temps (s.)	nb. solutions	temps (s.)
I	219601	42	1256	62
II	765140	149	4333	423
III	282346	56	1653	96
IV	1389130	275	8085	1195
V	528779	104	3156	235
VI	243509	73	1430	76
VII	979199	195	5833	652
VIII	506841	100	2999	235
IX	391111	78	2311	155
X	669282	135	3976	355
XI	581213	118	3564	296
XII	978146	195	5785	649
XIII	854161	168	4954	490
XIV	693774	139	4175	382
XV	952211	193	5902	648
XVI	851583	169	5092	517
Total	10886026	2189	64504	6466

TAB. 7.3 – Résultats de la comparaison entre MilPat et DARN ! sur le génome de la levure.

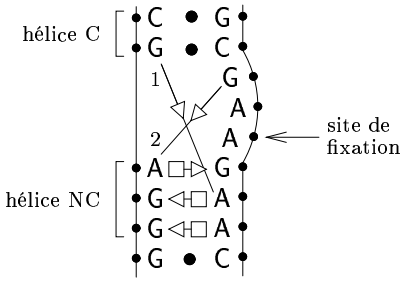


FIG. 7.2 – Un *k-turn*.

on pouvait s'attendre : l'élimination des solutions redondantes multiplie le temps de recherche par quatre, mais elle permet de diviser le nombre de solutions redondantes par cent soixante-huit. Mais ici, toutes les fonctions de coût étant dures, la dominance pourraît être traitée de façon beaucoup plus simple de la manière dont nous l'avons fait. Par ailleurs, quelques optimisations concernant le test de dominance permettront sans doute de gagner beaucoup de temps.

7.3 *k-turn*

Le *k-turn* [KSMS01, LLMW05] est un élément que l'on retrouve dans de nombreux ARN non-codants, qui intervient notamment dans le processus de reconnaissance de protéines. Le *k-turn*, comme présenté sur la figure 7.2, est composé de deux hélices : une hélice formée de liaisons canoniques, appelée hélice C, et une hélice faite de liaisons non-canoniques, appelée hélice NC. Entre ces deux hélices, deux liaisons non-canoniques (1 et 2) obligent la structure à adopter un virage à environ 120°, ce qui a pour effet de présenter le A à l'extérieur, et le rend plus réactif.

Le test que nous avons effectué ici tente de montrer l'intérêt des interactions non-canoniques.

Nous avons modélisé simplement le *k-turn* avec treize variables, cinq mots, deux hélices et deux interactions non-canoniques, modélisant les interactions 1 et 2 sucre–sucre. Seules trois erreurs étaient autorisées. Le descripteur est accessible sur le site Web de DARN !.

Nous avons recherché dans le génome de *P. abyssi* toutes les occurrences du *k-turn*. Nous en avons trouvé 1126 localement optimales. Des travaux sont en cours pour déterminer la quantité se trouvant dans des régions intergéniques, ou sur des petits ARN à boîte H/ACA ou C/D.

À titre de comparaison, si l'on supprime les liaisons sucre–sucre, le nombre de candidats passe à 3644 localement optimaux. La différence entre les deux ensembles de solutions est vraisemblablement pour la plupart des faux positifs. Ce test montre bien l'intérêt de modéliser les interactions non-canoniques.

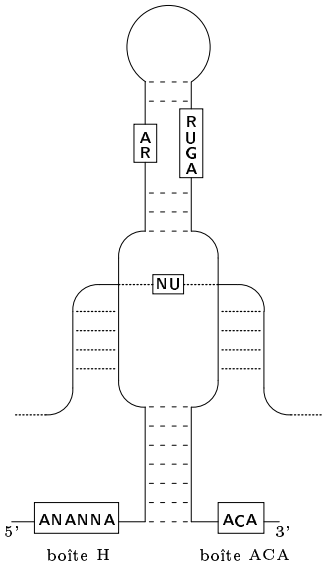


FIG. 7.3 – Une représentation d'un pARN à boîte H/ACA interagissant avec sa cible.

7.4 Petits ARN à boîte H/ACA

Un autre jeu de tests a été constitué par la recherche de petits ARN à boîte H/ACA [BCH02] (cf. figure 7.3) dans le génome de *T. kodakarensis*. Cet ARN intervient dans un processus de pseudo-uridylation, qui transforme un groupement ≡CH d'un nucléotide U en un groupement =NH. Cette transformation a pour effet principal d'accroître la capacité d'interaction du nucléotide transformé (se trouvant en général dans l'ARN ribosomique, mais aussi parfois dans un ARN messager ou un ARN de transfert) et permet donc la formation de structures secondaires et tertiaires plus stables.

Dans la pratique, le petit ARN à boîte H/ACA est un guide, c'est-à-dire qu'il est le responsable de la localisation du site à pseudo-uridyler. La pseudo-uridylation elle-même est assurée par une protéine qui interagit étroitement avec l'ARN à boîte H/ACA.

La structure secondaire est souvent composée de deux hélices dont une, régulière, se situe à la base et l'autre, plus irrégulière, forme le haut de la tige-boucle. Les deux hélices sont séparées par une boucle boucle interne. La base de l'hélice est en général mieux conservée que la seconde partie. À gauche et à droite de la base, on trouve respectivement les mots H (ANANNA) et ACA, parfois dégénérés.

Les nucléotides situés sur la boucle interne servent à sélectionner le nucléotide à pseudo-uridyler, en s'appariant de part et d'autre de ce nucléotide. Une fois la sélection faite, la protéine qui est accrochée dans la partie supérieure de l'hélice effectue la modification. Il s'avère que ce point d'accroche est en fait un *k-turn*, parfois relativement dégénéré.

En recherchant ces ARN, nous avons voulu tester deux choses : la capacité de notre logiciel à repérer de nouveaux ARNnc et la fonctionnalité de duplex. Pour cela, nous avons repris les pARN à boîte H/ACA trouvés par [TdGSG06] dans les génomes de *M.*

T. kodakarensis (plus de 20 millions de nucléotides) Temps : 65 s. Nombre de solutions : 6941 Nombre de solutions localement optimales : 22				
H/ACA	nb. tiges-boucles	DARN !	coût(s)	BLAST
H/ACA-1	1	oui	0	oui
H/ACA-2	2	oui	0, 1	non
H/ACA-3	1	oui	0	oui
H/ACA-4	2	oui	0, 1	oui
H/ACA-5	3	oui	0, 0, 2	oui
H/ACA-6	1	oui	0	oui

TAB. 7.4 – Résultat de la recherche des petits ARN à boîte H/ACA dans le génome de *T. kodakarensis*.

jannaschii, *P. abyssi*, *P. furiosus* et *P. horikoshii*, nous les avons alignés et nous avons conçu manuellement un descripteur, qui est donné dans la figure 7.4.

Le descripteur utilisé pour rechercher les pARN à boîte H/ACA a notamment permis de pénaliser des mots dégénérés (GA et ACA) ainsi que de favoriser les candidats possédant certaines régions à fort (G+C)%. Le coût maximum de 2 a été choisi de manière à augmenter la spécificité des solutions, c'est-à-dire à diminuer le nombre de faux positifs.

Les six pARN à boîte H/ACA du génome de *T. kodakarensis* similaires aux pARN des génomes voisins ont été consignés dans le tableau 7.4, en suivant la nomenclature utilisée dans [TdGSG06]. Un pARN pouvant être composé de plusieurs tiges-boucles, la deuxième colonne renseigne le nombre de tiges-boucles de chaque pARN. La troisième colonne rappelle que DARN ! a trouvé tous ces pARN. La quatrième colonne donne le score avec lequel chaque tige-boucle a été trouvée. La dernière colonne donne les résultats en utilisant BlastN [AGM+90], un outil standard qui recherche les séquences en utilisant la structure primaire uniquement, avec les paramètres par défaut.

DARN ! trouve en tout près de sept mille solutions, mais seules vingt-deux sont localement optimales et présentées à l'utilisateur, ce qui prouve en pratique l'utilité du mécanisme. Parmi les vingt-deux solutions, dix d'entre elles font partie des solutions consignées dans le tableau et sont donc très susceptibles d'être des pARN. Neuf solutions ont un coût nul, et deux d'entre elles ne ressemblent pas aux pARN trouvés dans les autres génomes. Elles ont toutefois toutes les bonnes caractéristiques pour être des pARN à boîte H/ACA. Six solutions ont un coût de un. Parmi elles, une solution fait partie de H/ACA-2 et une autre fait partie de H/ACA-4. Les autres n'appartiennent pas aux pARN connus mais peuvent constituer de bons candidats. Les autres pARN ont un coût de deux. L'un d'entre eux fait partie de H/ACA-5, les autres ne font pas partie des pARN connus, mais peuvent aussi constituer de bons candidats. Ceci montre la précision de notre approche.

```
TOP_VALUE = 3
VARIABLES X_VAR = 1...18, Y_VAR = 1...6

CONSTRAINTS

PATTERN[word="RA",err=0](X7,X8)
PATTERN[word="RUGA",err=0](X9,X10)
PATTERN[word="ANA",err=0](X17,X18)
PATTERN[word="UN",err=0](Y3,Y4)

HELIX[stem=7..10,loop=30..65,err=1](X1,X2,X15,X16)

DUPLEX[err=1](X3,X4,Y1,Y2)
DUPLEX[err=1](X13,X14,Y5,Y6)
COMPOSITION[nucleotides="GC",threshold>=75%](X1,X2)
COMPOSITION[nucleotides="GC",threshold>=70%](X5,X6)
COMPOSITION[nucleotides="GC",threshold>=65%](X11,X12)

SPACER[lenmin=0,lenmax=8](X2,X3)
SPACER[lenmin=3,lenmax=6](X3,X4)
SPACER[lenmin=1,lenmax=1](X4,X5)
SPACER[lenmin=6,lenmax=8](X5,X6)
SPACER[lenmin=1,lenmax=1](X6,X7)
SPACER[lenmin=3,lenmax=30](X8,X9)
SPACER[lenmin=1,lenmax=1](X10,X11)
SPACER[lenmin=4,lenmax=8](X10,X12)
SPACER[lenmin=1,lenmax=1](X12,X13)
SPACER[lenmin=3,lenmax=6](X13,X14)
SPACER[lenmin=0,lenmax=8](X14,X15)
SPACER[lenmin=0,lenmax=1](X16,X17)
SPACER[lenmin=3,lenmax=6](Y1,Y2)
SPACER[lenmin=1,lenmax=1](Y2,Y3)
SPACER[lenmin=1,lenmax=1](Y4,Y5)
SPACER[lenmin=3,lenmax=6](Y5,Y6)
SPACER[lenmin=10,lenmax=12](Y1,Y6)

PREFERENCES

PATTERN[word="GA",err=1](X7,X8)
PATTERN[word="ACA",err=1](X17,X18)
COMPOSITION[nucleotides="GC",threshold>=75..100%,costs=0..2](X1,X2)
COMPOSITION[nucleotides="GC",threshold>=70..100%,costs=0..2](X5,X6)
COMPOSITION[nucleotides="GC",threshold>=65..100%,costs=0..2](X11,X12)
```

FIG. 7.4 – Descripteur du pARN à boîte H/ACA.

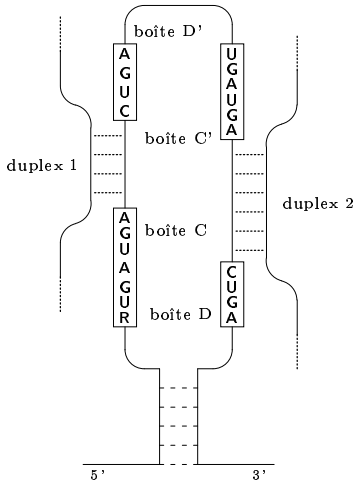


FIG. 7.5 – Une représentation d'un pARN à boîte C/D interagissant avec deux cibles.

7.5 Petits ARN à boîte C/D

Nous avons voulu ensuite compléter le jeu de tests précédent par la recherche de petits ARN à boîte C/D dans différents génomes d'archées. Ces ARN sont, comme les ARN à boîte H/ACA, des guides pour la maturation de l'ARN. Ils aident à sélectionner les sites de 2'–O–méthylation, où un groupement –OH situé sur le désoxyribose est transformé en un groupement $-O-CH_3$. L'ARN à maturer est souvent l'ARN ribosomique, et plus particulièrement les sous-unités 16S et 23S. On trouve aussi des nucléotides méthylés par l'ARN à boîte C/D situés sur les ARN de transfert.

La structure secondaire de ces ARN est la suivante (cf. figure 7.5). L'ARN est composé d'une petite hélice, surmontée d'une grande boucle. Cette boucle contient notamment deux mots relativement bien conservés : le mot RUGAUGA (où R désigne A ou G) appelé boîte C et le mot CUGA appelé boîte D. Deux autres mots sont présents dans cette même boucle, mais moins bien conservés : le mot UGAUGA appelé boîte C' et le mot CUGA appelé boîte D'.

Le fonctionnement de cet ARN est sensiblement le même que celui du pARN à boîte C/D. Cet ARN possède deux sites de méthylation distincts. Le premier est situé entre la boîte C et la boîte D' et le second est situé entre la boîte C' et la boîte D. Les boîtes C/D et C'/D' s'avèrent aussi être des *k-turn* et sont les points d'accroche à la protéine qui réalise la 2'–O–méthylation. Celle-ci s'effectue sur le nucléotide situé à cinq bases en amont de la boîte D ou D'.

Notre but a ici été de connaître la spécificité et la sensibilité de notre approche en se rapportant à une base de données appelée SnoRNADb[2] [OLR+00]. Celle-ci recense un certain nombre de pARN à boîte C/D qui ont été validés expérimentalement chez les archées. Les auteurs de la base de données ont utilisé un logiciel spécifique aux pARN à boîte C/D appelé snoScan [LE99] et parmi les réponses données par cet outil, en ont

[2] Accessible sur http://lowelab.ucsc.edu/snoRNAdb/.

Organisme	avec duplex					sans duplex				
	VP	FP	FN	spé.	sens.	VP	FP	FN	spé.	sens.
A. pernix	13	56	11	19 %	54 %	14	192	10	7 %	58 %
A. fulgidus	4	189	0	2 %	100 %	4	679	0	1 %	100 %
M. jannaschii	8	62	0	11 %	100 %	8	406	0	2 %	100 %
P. horikoshii	48	97	3	33 %	94 %	49	359	2	12 %	96 %
P. furiosus	48	70	4	40 %	92 %	51	404	1	11 %	98 %
P. abyssi	50	56	6	47 %	89 %	53	304	3	15 %	95 %
T. kodakarensis	146					357				

TAB. 7.5 – Résultats

sélectionné certaines pour vérification expérimentale. La base de données n'est donc pas exhaustive.

Dans ce jeu de tests, nous avons créé deux descripteurs de pARN à boîte C/D, accessibles sur la page Web de DARN!. Le premier descripteur contient un duplex (entre la boîte C et la boîte D') alors que le second n'en contient pas. Nous avons autorisé une erreur dans toutes les boîtes ainsi que dans le duplex, sachant que les solutions ne pouvaient contenir plus de trois erreurs au total. Nous avons donné comme séquence principale le génome complet de chaque organisme et comme séquence cible la sous-unité 16S puis la sous-unité 23S. Nous avons ensuite fusionné les deux résultats. Le but de ces expérimentations est de tester la sensibilité de DARN!, c'est-à-dire sa capacité à retrouver les pARN répertoriés, et à en oublier le moins possible. Le nombre de faux positifs, autrement dit le nombre de pARN trouvés mais non répertoriés, est sans doute moins important. En effet, la base de données de référence ne contient pas tous les pARN à boîte C/D. Ces faux positifs peuvent leurs être de véritables pARN pas encore découverts.

Le tableau 7.5 donne donc le nombre de vrais positifs (VP), de faux positifs (FP) et de faux négatifs (FN) ainsi que la spécificité et la sensibilité pour chaque organisme analysé et pour chaque descripteur. Nous avons aussi analysé le génome de *Thermoccocus kodakarensis*, dans lequel le nombre de pARN n'est pas connu. Seules les solutions localement optimales ont été prises en compte.

Certains des pARN qui ne sont pas reconnus par DARN! possèdent deux erreurs dans une boîte C, D, C' ou D'. C'est particulièrement vrai chez *A. pernix* (la seule archée analysée de la branche des crénarchées, alors que les autres sont des euryarchées), où les boîtes C' et D' sont très dégénérées. Pour quelques autres pARN, il ne semble pas exister de duplex près de la boîte D', mais seulement près de la boîte D. Pour d'autres encore, le duplex ne s'apparie ni avec le 16S, ni avec le 23S, mais avec un ARNt ou une région inconnue. Le descripteur avec duplex ne reconnaît donc pas ces candidats, alors que le descripteur sans duplex les reconnaît.

On peut voir que le descripteur avec duplex donne d'assez mauvais résultats sur *A. pernix*, à cause du nombre d'erreurs sur les boîtes. Le nombre de faux négatifs (onze faux négatifs pour treize vrais positifs) est très important, alors qu'il est beaucoup

plus acceptable pour les autres organismes. Pour les autres organismes, la sensibilité va de 89 % pour *P. abyssi* à 100 % pour *A. fulgidus* et *M. jannaschii*, ce qui semble acceptable. Si l'on enlève le duplex, la spécificité diminue (le nombre de solutions est au moins multiplié par trois) alors que la sensibilité n'augmente que peu (au maximum trois candidats de plus sont reconnus).

À titre de comparaison, notre logiciel analyse chaque génome en moins de cinq secondes tandis que SnoScan donne 1183 réponses pour le génome de *A. pernix* à apparier avec ses sous-unité 16S et 23S en deux cent quatre secondes. Parmi les réponses, on compte huit vrai positifs, quinze faux négatifs et mille cinq soixante-quinze faux positifs.

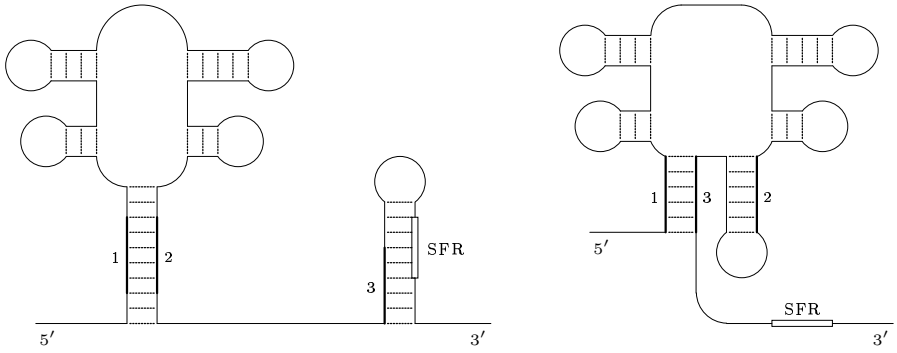
Lors des expérimentations, nous avons aussi remarqué que certains pARN étaient situés à seulement quelques nucléotides les uns des autres. Une question est alors de savoir si notre mécanisme de détection de solutions localement optimales prendrait un des pARN comme une solution localement optimale et l'autre comme une dominée, ou bien si le mécanisme serait capable de donner les deux solutions. Même pour les solutions situées à une dizaines de nucléotides l'une de l'autre, le mécanisme n'oublie pas de solution. C'est par exemple le cas pour sR60 et sR26 dans le génome de *P. horikoshii*. Les deux solutions, situées sur le brin moins, sont respectivement détectées entre les positions 1 328 875 et 1 328 827 avec un coût de deux pour la première, et 1 328 938 et 1 328 888 avec un coût de un pour la seconde. Même avec douze nucléotides séparant les deux pARN, DARN ! a été capable de les donner comme solutions.

Afin d'améliorer la signature, nous souhaiterions effectuer de nouvelles expérimentations où les boîtes conservées C, D, C' et D' seraient modélisées par des interactions non-canoniques. Ceci permettrait d'obtenir des résultats sans doute plus spécifiques pour *A. pernix*.

7.6 *Riboswitches*

Les *riboswitches* [WB03, BCW$^+$04, MB04, VRMG04, WB05] ne sont pas des ARN non-codants. Ce sont des motifs structurés situés sur des régions non codantes des ARN messagers. Ils permettent de réguler la transcription ou la traduction du gène — souvent un gène d'enzyme — près duquel ils sont situés, en fonction de la concentration d'un métabolite. Ils sont donc des acteurs de la régulation au sein du réseau métabolique. Considérons par exemple la figure 7.6(a). Celle-ci représente à gauche le *riboswitch* appelé élément RFN [VRMG02] lorsque la concentration en flavine mononucléique (un dérivé de la vitamine B_2) est suffisamment élevée. À droite du *riboswitch*, on peut observer le site de fixation du ribosome (SFR ou *ribosomal binding site* en anglais). C'est sur cette région que se fixe le ribosome pour amorcer la traduction de l'ARN messager en protéine. Si ce site est pris dans une tige-boucle, comme c'est le cas ici, le ribosome ne peut pas se fixer et la traduction ne se fait pas.

En cas de diminution de la concentration en flavine mononucléique, le *riboswitch* adopte une autre structure tertiaire. Une partie du *riboswitch* s'apparie avec la région 3 (cf. figure 7.6(b)) qui interagissait auparavant avec le site de fixation du ribosome. Le site est donc maintenant découvert et la traduction peut se faire.



(a) Lorsque la concentration en flavine mononucléique est élevée, le site de fixation du ribosome (RBS) est séquestré.

(b) Lorsque la concentration en flavine mononucléique est faible, le RBS est fonctionnel.

FIG. 7.6 – Mécanisme de régulation de la traduction par un *riboswitch*.

Riboswitch	Métabolite
RFN	flavine mono-nucléique (dérivé de la vitamine B_2)
B12	vitamine B_{12}
THI	thiamine pyrophosphate (dérivé de la vitamine B_1)
boîte S, boîte S-II	S-adénosylméthionine (donneur de groupe méthyle)
boîte L	lysine (acide aminé)

TAB. 7.6 – Quelques *riboswitches* et leur métabolite associé.

Il existe plusieurs types de *riboswitches* et chacun a une structure secondaire plus ou moins complexe. Le tableau 7.6 donne un liste de quelques noms des *riboswitches*, ainsi que leur métabolite associé. Les *riboswitches* sont tous très structurés et possèdent au moins trois hélices.

Le but du test que nous avons mis en place est de savoir si le module de génération de signatures était sensible. Pour cela, nous avons téléchargé un alignement pour chaque catégories de *riboswitches* et nous avons opéré de la manière suivante. Nous avons enlevé la première séquence de l'alignement, qui sera la séquence testée. Nous avons ensuite lancé le module de génération de signature sur le reste de l'alignement. Nous avons enfin lancé DARN ! sur le descripteur donné par le module et la séquence de test. Nous avons itéré le processus, en prenant la deuxième séquence comme séquence test, etc. Nous avons enfin lu les résultats donnés. Nous avons recueilli tous nos alignements de RFAM (contenant principalement des bactéries, quelques virus et parfois un eucaryote), sauf l'alignement boîte S-II, qui a été rendu public dans [CBL+02] (contenant exclusivement des séquences bactériennes). Les résultats sont consignés dans le tableau 7.7.

Les résultats permettent de s'assurer que le module de génération de signatures était relativement sensible. La sensibilité va de 72 % pour l'élément RFN à 100 %

Riboswitch	vrais positifs	faux négatifs	nombre de séquences
RFN	23	9	32
B12	128	0	128
THI	173	1	174
boîte S	48	7	55
boîte S-II	73	2	75
boîte L	37	6	43

TAB. 7.7 – Résultats de la recherche de *riboswitches*

pour l'élément B12. Il est à noter que la sensibilité est notamment plus élevée dès lors que les alignements contiennent un grand nombre de séquences. Le plus petit score de sensibilité est donné pour l'alignement contenant le plus petit nombre de séquences (trente-deux séquences pour l'élément RFN), alors que les alignements contenant plus de cent séquences (B12 et THI) ont des scores presque parfaits (zéro ou un seul faux négatif observé).

La raison en est la suivante. Dans notre jeu de tests, les faux négatifs ne sont pas décelés lorsqu'ils possèdent une variabilité orpheline : une erreur dans une hélice alors que toutes les autres séquences n'en contiennent pas, une boucle beaucoup plus courte que l'on ne retrouve pas ailleurs, etc. Lorsque le nombre de séquences dans l'alignement est suffisamment grand, la variabilité dans les structures primaire et secondaire que l'on observe à l'intérieur même d'une famille d'ARN est mieux capturée et le nombre de faux négatifs diminue.

Notons aussi que RFAM contient parfois quelques doublons, c'est-à-dire des séquences strictement identiques, apparaissant plusieurs fois. Cela peut faire augmenter artificiellement le nombre de vrais positifs trouvés.

La figure 7.7 donne les éléments de structure reconnus sur RFN. Dans les première, deuxième et cinquième hélices, toutes les interactions des hélices sont reconnues et conservées dans la signature. Ces hélices contiennent respectivement dix, trois et trois paires de bases. La troisième hélice est peu conservée : dans certaines séquences, elle compte une suppression, et dans d'autres deux mésappariements. L'hélice n'est donc pas retenue. Concernant la quatrième hélice, on observe une longue insertion dans une des séquences de l'alignement. Cette hélice est donc divisée en deux par le générateur de signatures en une hélice de quatre paires de bases et une hélice d'une seule paire de bases. On observe, dans une séquence de l'alignement, que l'hélice à quatre paires de bases contient trois mésappariements. Cette partie est donc éliminée. Seule reste l'hélice à une paire de bases.

Certains mots conservés reconnus par le générateurs sont parfois longs (le mot 1 compte trente-quatre nucléotides). Cela vient tout d'abord du fait que la structure secondaire est très conservée, et donc on observe peu d'insertions et de suppressions dans les hélices. De plus, on observe peu de mutations compensatoires et donc la séquence est relativement conservée. Enfin, les boucles des hélices 2, 4 et 5 sont aussi très conservées et l'on retrouve des structures hyper-stables (le troisième mot est GYNNRA). Cela se

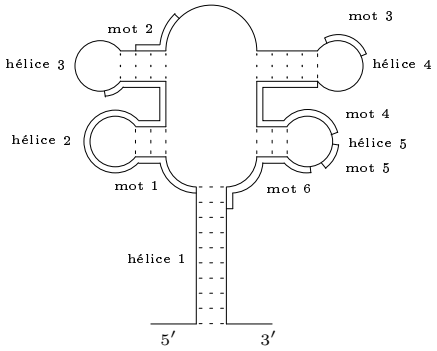


FIG. 7.7 – La signature de la riboflavine générée automatiquement — les liaisons en pointillés ne sont pas retrouvées, alors que les liaisons en traits discontinus le sont, les mots trouvés sont représentés par des doubles lignes.

retrouve dans la signature.

7.7 RNase P

La RNase P [AKT93, PB95, MJW98, KP06] intervient principalement dans le processus de maturation des ARN de transfert (cf. figure 7.1(a)). La RNase P est une des plus longues séquences d'ARN connues. Elle compte plusieurs centaines de nucléotides. Sa structure secondaire est complexe et variable selon les organismes. Néanmoins, on peut parvenir à exhiber un cœur conservé qui a en charge l'activité catalytique. Ce cœur est donné dans la figure 7.8.

La maturation de l'ARNt se fait de la manière suivante. Initialement, le bras accepteur du pré-ARNt (qui est l'hélice située en haut dans la figure 7.1(a)) est prolongé dans le sens $5'$. Le rôle de la RNase P est d'élider cette prolongation de l'ARNt mature. La RNase P reconnaît alors le bras accepteur et le bras T (situé à droite dans la figure) du pré-ARNt. Une des protéines associées à la RNase P enlève alors la partie en amont du bras accepteur.

Dans ce jeu d'expérimentations, nous avons tout d'abord voulu tester la sensibilité et la spécificité de notre module de génération de signatures. Ensuite, nous avons voulu savoir si DARN ! était capable de rechercher des séquences d'ARN longues. Nous avons donc pris de la base de données *the RNase P database*[3] [Bro99] un alignement réalisé par [MJW98] contenant cent soixante-cinq séquences de RNase P de bactéries. Nous avons ensuite généré automatiquement une signature grâce au module de génération de signatures. Celui-ci a créé un descripteur contenant trente-deux variables, six mots conservés, cinq hélices et autorisant un score maximum de cinquante-sept. Ce descripteur est accessible sur le site Web de DARN !.

Nous avons ensuite recherché dans tous les génomes de bactéries contenus sur GenBank [BKML+07] si l'on retrouvait la RNase P, sachant qu'*a priori*, il y en a exactement

[3]Accessible sur http://www.mbio.ncsu.edu/RNaseP/.

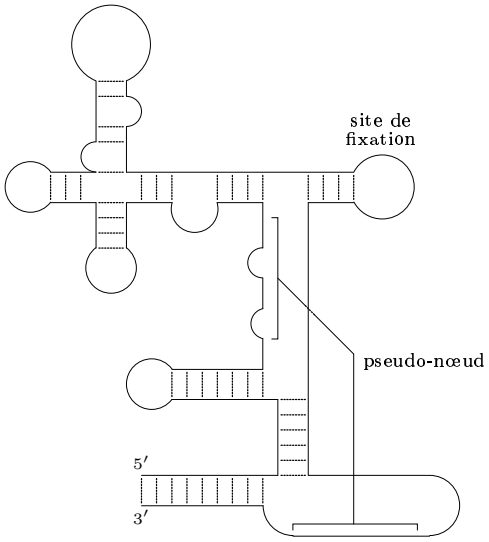


FIG. 7.8 – Le cœur fonctionnel d'une RNase P.

un par organisme.

Nous avons mené nos expérimentations sur la version 160.0 de GenBank. Les résultats ont été consignés dans l'annexe B. On peut retenir les points suivants :

- 508 génomes bactériens recensés,
- 1 génome où l'on trouve deux RNase P (*Salinispora tropica* CNB-440),
- 469 génomes où l'on trouve une RNase P,
- 38 génomes où l'on ne trouve aucune RNase P.

Les recherches ont été menées sur une machine AMD Opteron, cadencée à 2 GHz, et contenant 16 Go de RAM. Le temps d'analyse d'un chromosome dure moins de quatorze secondes. À titre de comparaison, Infernal (avec un préfiltrage utilisant les chaînes de Markov cachées), entraîné sur le même alignement (sans les pseudo-nœuds qu'il ne peut pas prendre en compte) trouve deux solutions sur *B. subtilis* en plus de deux heures sur une machine allant à 2,4 GHz avec 8 Go de mémoire vive.

Nos résultats montrent tout d'abord que l'outil de génération de signatures est particulièrement spécifique. Pour une solution non voulue dans cinq cent huit génomes, le taux de spécificité est de presque 100 %. Ce taux s'explique en partie par le fait que la structure à rechercher est longue, possède de nombreuses hélices et quelques mots conservés. Toutefois, notre recherche a aussi inclus les génomes d'archées et les plasmides des bactéries que, dans notre cas, nous pouvons considérer comme des séquences de test de spécificité car elles ne doivent pas contenir de RNase P bactérienne. Il s'est avéré qu'aucune RNase P n'a été découverte dans ces chromosomes, conformément à nos attentes.

Le taux de sensibilité atteint les 92,5 %, ce qui nous assure que notre outil est relativement sensible. Ce résultat vient aussi du fait que l'alignement d'entrée contient environ un tiers des candidats à retrouver, ce qui facilite beaucoup la tâche de notre programme.

Ce banc de tests montre également que DARN ! est capable de rechercher rapidement des ARN longs. Les bactéries ont des génomes pouvant compter de quelques centaines de milliers de nucléotides à une dizaine de millions. La recherche d'un ARN de plusieurs centaines de nucléotides dans de tels génomes en moins de quatorze secondes prouve que notre outil est bien adapté à ce type de recherches.

7.8 Conclusion

Dans ce chapitre, nous avons testé plusieurs caractéristiques du logiciel de recherche de signatures et du module de génération de signatures que nous avons conçus. Nous avons tout d'abord comparé la cohérence d'arc aux bornes avec la cohérence d'arc et montré que la seconde était en pratique inutilisable pour nos problèmes, à cause de ses complexités temporelle et surtout spatiale.

La possibilité de modéliser des interaction non-canoniques et les interactions entre deux séquences d'ARN s'avère aussi intéressante. Nous avons vu que, sur les *k-turn*, l'ajout d'interactions non-canoniques permettait de diviser par trois le nombre de solutions données par DARN !. Sur des petits ARN à boîte C/D, l'ajout d'un duplex permet de faire augmenter sensiblement la spécificité, sans trop faire descendre la sensibilité.

Notre outil s'avère suffisamment expressif pour rivaliser avec des outils recherchant spécifiquement les membres d'une seule famille donnée, comme c'est le cas pour tRNAscan-SE. Par rapport à ce dernier, notre outil montre de bons résultats en terme de sensibilité, spécifité et rapidité. Il est capable de rechercher des structures longues et complexes contenant un grand nombre d'hélices, comme la RNase P, en quelques secondes.

De plus, le module de génération de signatures, même s'il constitue encore une ébauche, donne de bons résultats en terme de sensibilité (comme on le voit sur l'exemple des *riboswitches*). Sur l'exemple de la RNase P, par la sensibilité et la spécificité des signatures qu'il génère, il montre qu'il peut être un outil intéressant pour le génération de signatures. Celles-ci sont discriminantes et parviennent à retrouver le cœur d'une signature avec un nombre réduit de variables et de fonctions de coût.

Dans le futur, nous souhaiterions mener d'autres expérimentations. Nous voudrions concevoir un descripteur unique reconnaissant l'ARN de transfert sur un grand nombre d'eucaryotes, et connaître la sensibilité et la spécificité des signatures de *riboswitches* générées automatiquement sur des génomes complets.

Conclusion

À l'issue de la présentation de notre recherche, nous aimerions en souligner les apports et faire un recensement non exhaustif des travaux pouvant être menés à sa suite.

Apports

Nous avons développé dans le cadre de nos recherches un outil appelé DARN !, qui recherche les motifs d'ARN grâce aux réseaux de contraintes pondérées. Nous avons vu comment ces derniers permettaient de résoudre élégamment le problème en dissociant efficacement les algorithmes de *pattern-matching* et les algorithmes de recherche de solutions. Ces derniers types d'algorithmes, permettant d'accélerer notablement la résolution des problèmes, ont été largement étudiés dans le domaine de l'intelligence artificielle et il nous a été facile d'en reprendre un certain nombre d'entre eux : branchements binaires et réfutation, backjumping, ordonnancement des variables, etc. Nous avons aussi vu que le formalisme des contraintes pondérées avait l'avantage d'être modulaire, dans la mesure où, pour modéliser un nouvel élément de structure, il suffit d'implanter une fonction de coût. Nous en avons profité pour proposer un grand nombre d'éléments de structure : plusieurs types d'hélices, des mots, mais aussi un (G+C)%, des interactions non-canoniques et surtout des duplex, rarement modélisés avant. De plus, nous avons vu que la gestion des coûts permettait de modéliser finement les signatures, de pouvoir quantifier la qualité des solutions trouvées, et même de supprimer certaines solutions non-pertinentes.

Nous avons également présenté deux nouveaux algorithmes de filtrage utilisables dans les réseaux de contraintes pondérées. Le premier est la cohérence d'arc existencielle, qui s'applique à des réseaux de contraintes pondérées plus « classiques » que ceux utiisés par la recherche d'ARNnc. Cette propriété est sans la meilleure connue à ce jour et elle a permis à TOULBAR [LS03, HLdGZ05], un solveur de contraintes pondérées utilisant la cohérence d'arc existencielle, de gagner la compétition de solveurs de contraintes[4]. Un autre algorithme de filtrage a été proposé pour les réseaux dont dont la taille des domaines est très grande, comme c'est le cas dans les problèmes de recherche de structures d'ARN. Il s'agit de la cohérence d'arc aux bornes, dont nous avons décliné

[4]La compétition a été organisée dans l'atelier *Constraint Propagation And Implementation* en 2006 et les résultats sont accessibles à l'adresse http://www.cril.univ-artois.fr/CPAI06/round2/results/results.php?idev=7.

plusieurs spécialisations dans le cas où la sémantique des fonctions de coût est connue, ou lorsque l'on souhaitait obtenir une propriété de cohérence locale plus forte. Il est clair que cet algorithme de filtrage pourra être utilisé pour n'importe quel réseau de contraintes pondérées possédant, comme dans notre cas, de très grands domaines. On peut notamment penser aux réseaux temporels, où l'horizon de temps peut être long, et aux réseaux numériques, qui recherchent les solutions dans les flottants et donc divisent les domaines en intervalles très petits.

Nous avons aussi implanté un algorithme de *pattern-matching* pour chaque fonction de coût. Ces algorithmes ont pour la plupart été extraits de la littérature et ont été choisis parce qu'ils étaient particulièrement performants, comme dans le cas de la fonction de coût de mot. Mais d'autres ont été spécialement conçus pour cette méthode, comme l'algorithme de recherche de mots approchés dans un tableau de suffixes. Cet algorithme, très général, pourrait d'ailleurs être utilisé avec profit pour n'importe quelle recherche de mot approchée.

Nous avons de plus commencé l'élaboration d'un utilitaire permettant de créer automatiquement des signatures dans un langage compris par DARN!, en se basant sur un alignement de séquences où la structure est renseignée.

D'après les tests que nous avons menés, l'approche suivie semble donner de bons résultats en termes de sensibilité, spécificité et rapidité. Elle est de plus robuste, dans la mesure où elle peut détecter même de longs ARN dans des génomes en un temps raisonnable. Les expérimentations ont aussi montré que l'utilisation d'éléments de structures parfois non modélisables par d'autres approches, comme les interactions non-canoniques ou les duplex, permettent de rendre DARN! beaucoup plus spécifique et constituent donc un avantage de notre outil. Le module d'apprentissage de signature donne également des descripteurs concis et pertinents, qui peuvent être utilisés pour rechercher de nouveaux ARN, mais aussi être modifiés et améliorés par l'utilisateur.

Perspectives

Le travail est certainement loin d'être fini. Nous ne reparlerons pas ici de celui qu'il resterait à faire pour améliorer le générateur de signatures puisque nous l'avons déjà évoqué dans le chapitre six. Considérant DARN! lui-même, tout solide qu'il soit, il pourrait être complété par un certain nombre de fonctionnalités. Tout d'abord, nous avons vu dans le chapitre 6 qu'il est possible d'améliorer le test de dominance, ce qui permettrait de gagner beaucoup de temps dans les cas où l'on a beaucoup de solutions. De plus, il peut être intéressant d'ajouter de nouvelles fonctions de coût. On peut par exemple penser à une fonction de duplex permettant l'interaction entre une séquence principale et une séquence parmi plusieurs séquences cibles. Les petits ARN à boîte H/ACA servent par exemple à maturer l'ARN ribosomique ou un ARN de transfert. Si l'on pouvait entrer un fichier multi-fasta contenant ces ARN, la fonction de coût permettrait de dire avec quelles séquences le pARN peut s'apparier.

Un élément important qui pourraît être ajouté est l'évaluation d'une structure par son énergie libre. Les modalités précises de cet ajout restent à définir, mais on pour-

rait affiner le modèle en choisissant de préférence les structures thermodynamiquement stables. Cela permettrait par exemple de privilégier simplement les hélices les plus longues. Un appel à l'algorithme de Zuker [ZS81] semble indiqué dans ce cas.

On peut aussi modifier l'implantation de certaines fonctions de coûts et améliorer par exemple la fonction de coût d'hélice alternative, dont l'algorithme de propagation n'est sans doute pas excellent. Il est de plus possible d'utiliser les tableaux de suffixes pour la contrainte dure de mot. Si l'on veut aller plus loin, on peut aussi employer un seul tableau de suffixes pour tous les mots d'une signature, en utilisant les algorithmes développés pour Smile [Mar02] ou ses successeurs, Riso [CFOS06] et Risotto [PCMS06].

Nous avons aussi noté que PatScan acceptait des signatures contenant des alternatives. On peut ainsi par exemple spécifier que l'on veut soit le mot ACGU, soit une hélice de taille cinq. Ce n'est pas possible dans DARN ! mais l'étude de mécanismes gérant les alternatives pourraient faire l'objet de recherches ultérieures.

Une autre question est de savoir si DARN ! peut être adapté pour les séquences protéiques. La réponse est affirmative, moyennant très peu de modifications. Il suffirait simplement de changer les matrices de substitutions (utilisant une matrice PAM ou BLOSUM) et l'analyseur de fichiers fasta (qui s'attend à lire des séquences nucléotidiques).

De plus, pour rendre l'outil facilement utilisable, nous pensons que son ergonomie gagnerait à être enrichie. Les points suivants proposent des pistes d'amélioration.

On peut tout d'abord remarquer que la nomenclature des variables dans les descripteurs n'est pas particulièrement simple. L'analyseur de descripteurs s'attend à avoir des variables de type x_1, x_2, ... ayant des numéros qui se suivent. Si l'on veut supprimer une variable, il faut potentiellement renommer toutes les autres, ce qui est parfois fastidieux. Il serait plus pratique d'effectuer une première passe de l'analyseur, qui relève tous les noms des variables (ayant une syntaxe plus libre) afin d'éviter ce problème.

De plus, si l'on n'est pas familier avec le concept de contraintes dures et de fonctions de coût (ce qui est généralement le cas), il est parfois difficile de comprendre la différence entre la partie `CONTRAINTES`, où les fonctions de coût modélisant les contraintes sont données, et la partie `PRÉFÉRENCES`, qui contient les fonctions de coût exprimant les options (cf. paragraphe 3.4.4 pour plus de détails sur cette distinction). Nous voudrions faire en sorte que chaque élément de structure ne soit mentionné qu'une fois et que des mots-clefs spécifient dans quelle classe se trouve l'élément de structure, comme nous l'avons décrit dans le paragraphe 6.2.

Il serait particulièrement intéressant de créer une base de données de signatures, comme dans RFAM [GJMM+05]. Cette base de données permettrait de rechercher simplement des ARNnc dans une séquence proposée par l'utilisateur. Les signatures de la base pourrait aussi être adaptées par l'utilisateur, pour les besoins de sa recherche.

Enfin, nous souhaitons ajouter une fonctionnalité sur le site Web faisant apparaître la structure secondaire des résultats trouvés. Il suffirait de donner chaque solution trouvée par DARN ! à un logiciel de visualisation de structure secondaire comme RNADraw (developpé dans le cadre du programme IRIS [Per04]). D'une manière générale, une représentation sous forme graphique est sans doute préférable à celle d'un descripteur. On pourrait s'inspirer alors des travaux de [RRG07].

Grâce à ces améliorations et aux recherches en informatique que nous avons menées pour lui, DARN ! pourrait devenir, tel est du moins notre souhait, un outil performant et fiable, utilisé par la communauté bio-informaticienne et biologiste de l'ARN.

Annexe A

Guide d'utilisation DARN !

Avertissement 1 *Ceci est le guide d'utilisation en ligne de DARN!. Pour qu'il puisse être utilisé partout dans le monde, il a été écrit en Anglais. Ce manuel décrit précisément la syntaxe actuellement utilisée par DARN!*

This document should tell you how to use DARN!. This tool takes as parameter a descriptor, a DNA/RNA sequence and possibly another DNA/RNA sequence. Its aim is to find in the sequence(s) all the subsequences that match the descriptor.

A.1 The descriptor

The descriptor should describe the general shape of a non-coding RNA (ncRNA) through constraints.

A.1.1 General shape

Here is the skeleton of a descriptor :

```
TOP_VALUE = n
POSITIONS X_VAR = X_min..X_max, Y_VAR = Y_min..Y_max
CONSTRAINTS
# constraints here
PREFERENCES
# preferences here
```

The first line gives the *overall* number of errors the candidate may have, with respect to the signature. This number is n.

On the second line, the `X` positions refer to positions in the main sequence, whereas the `Y` positions refer to positions in the target sequence, if any. Thus, if you do not need any target sequence, drop the `Y_VAR =` $Y\_min..Y\_max$ part. This line creates $X\_max - X\_min + 1$ positions on the main sequence (ranging from $\mathtt{X}_{X\_min}$ to $\mathtt{X}_{X\_max}$), and $Y\_max - Y\_min + 1$ on the target sequence.

A.1.2 Constraints

Here are some examples involving every supported constraint.

```
PATTERN[word="ACGU",errors=1](X1,X2)
```

Look for the word ACGU starting at position `X1` and ending at position `X2`, with one possible error.

```
SPACER[length=10..20](X1,X2)
```

State that the distance between positions `X1` and `X2` should be not less than 10, and not greater than 20. This means that $10 \leq \texttt{X2} - \texttt{X1} \leq 20$, or, put in other words, that the number of nucleotides between `X1` and `X2` should be not less than 9, and not greater than 19.

```
COMPOSITION[nucleotides="CG",threshold>=80](X1,X2)
```

State that the (G+C)% between positions `X1` and `X2` should be not less than 80%. We may express than the (G+C)% should be not greater than 20% by writing `threshold<=20` instead.

```
HELIX[stem=5..7,loop=4..10,errors=1](X1,X2,X3,X4)
```

Set the presence of an helix formed by a stem between positions `X1` and `X2` on the one hand, and a stem between positions `X3` and `X4` on the other hand. The stem should contain not less than 5 base pairings, and not more than 7 base pairings. The loop should contain not less than 4 nucleotides, and not more than 10 nucleotides. One substitution is allowed in the stem. If you also want to allow insertions and deletions, add `indels=yes` to the list of parameters. Only canonic interactions (i.e. A–U and G–C) are allowed. If you want to allow wobble interactions (i.e. G–U), add `wobble=yes` to the list of parameters.

```
REPETITION[size=5..7,distance=4..10,errors=1](X1,X2,X3,X4)
```

Set that the subsequence between positions `X1` and `X2` is repeated between positions `X3` and `X4`. The size parameter give the size of the repeated subsequence, and the distance parameter gives the distance between the two subsequences. The number of allowed differences between the subsequences is given by the parameter error, and the optional parameter `indels` states whether indels are allowed.

```
PAIR[interaction=WATSON_CRICK,interaction=SUGAR,orientation=CIS,family=2](X1,X2)
```

Set a non canonic pairing between the nucleotides at positions `X1` and `X2`. The nucleotide at position `X1` interacts with its Watson-Crick side ; the nucleotide at position `X2` interacts with its Sugar side. This is a *cis* interaction, in the geometric family 2, as defined by *Leontis et al.*. If you wish to accept all geometric families, given two sides and an orientation, write `family=all`.

```
DUPLEX[errors=1](X1,X2,Y1,Y2)
```

Set the presence of a duplex formed by a stem between positions `X1` and `X2` on the main sequence, and a stem between positions `Y1` and `Y2` on the target sequence. The duplex may contain a substitution, an insertion, or a deletion. Notice that it is a time consuming constraint, so set the distance between `X1` and `X2`, and between `Y1` and `Y2`, and do not use high numbers of errors. If you want to allow wobble interactions, add `wobble=yes` to the list of parameters.

A.1.3 Preferences

The `PATTERN`, `HELIX`, `REPETITION`, `DUPLEX` preferences have the same syntax as the constraints. The given cost is the number of errors found. The meaning of the soft `SPACER` and `COMPOSITION` is the following.

```
SPACER[length=15..20..30..35,costs=1..5](X1,X2)
```

If the distance between positions `X1` and `X2` is not less than 20, and not greater than 30, then give a cost of 1. If the distance between positions `X1` and `X2` is less than 15, or greater than 35, then give a cost of 5. Otherwise, it gives a cost between 1 and 5.

```
COMPOSITION[nucleotides="CG",threshold>=60..80,costs=1..5](X1,X2)
```

If the (G+C)% between positions `X1` and `X2` is not less than 80%, then the given cost is 1. If it is not greater than 60%, then the given cost is 5. Otherwise, the cost is between 1 and 5.

The soft `PAIR` constraint has the same syntax as the hard one. The only change is the following. If an interaction is not in the given family, but still allowed by the given sides and orientation, it is accepted, with a penalty.

For the `PATTERN`, `HELIX` and `DUPLEX` preferences, you can also change the costs given by the preference. For example, if you write

```
PATTERN[word="ACGU",errors=1](X1,X2) 0,2
```

then the cost given for no error is 0 (as usual). If one error is found, a cost of 2 is given.

A.1.4 Example

This example detects H/ACA box sRNAs.

```
TOP_VALUE = 3
VARIABLES X_VAR = 1...18, Y_VAR = 1...6

CONSTRAINTS

PATTERN[word="RA",errors=0](X7,X8)
PATTERN[word="RUGA",errors=0](X9,X10)
PATTERN[word="ANA",errors=0](X17,X18)
PATTERN[word="UN",errors=0](Y3,Y4)

HELIX[stem=7..10,loop=30..65,errors=1](X1,X2,X15,X16)

DUPLEX[errors=1](X3,X4,Y1,Y2)
DUPLEX[errors=1](X13,X14,Y5,Y6)
COMPOSITION[nucleotides="GC",threshold>=75%](X1,X2)
COMPOSITION[nucleotides="GC",threshold>=70%](X5,X6)
COMPOSITION[nucleotides="GC",threshold>=65%](X11,X12)

SPACER[lenmin=0,lenmax=8](X2,X3)
SPACER[lenmin=3,lenmax=6](X3,X4)
SPACER[lenmin=1,lenmax=1](X4,X5)
SPACER[lenmin=6,lenmax=8](X5,X6)
SPACER[lenmin=1,lenmax=1](X6,X7)
SPACER[lenmin=3,lenmax=30](X8,X9)
SPACER[lenmin=1,lenmax=1](X10,X11)
SPACER[lenmin=4,lenmax=8](X10,X12)
SPACER[lenmin=1,lenmax=1](X12,X13)
SPACER[lenmin=3,lenmax=6](X13,X14)
SPACER[lenmin=0,lenmax=8](X14,X15)
SPACER[lenmin=0,lenmax=1](X16,X17)
SPACER[lenmin=3,lenmax=6](Y1,Y2)
SPACER[lenmin=1,lenmax=1](Y2,Y3)
SPACER[lenmin=1,lenmax=1](Y4,Y5)
SPACER[lenmin=3,lenmax=6](Y5,Y6)
SPACER[lenmin=10,lenmax=12](Y1,Y6)

PREFERENCES

PATTERN[word="GA",errors=1](X7,X8)
PATTERN[word="ACA",errors=1](X17,X18)
COMPOSITION[nucleotides="GC",threshold>=75..100%,costs=0..2](X1,X2)
COMPOSITION[nucleotides="GC",threshold>=70..100%,costs=0..2](X5,X6)
COMPOSITION[nucleotides="GC",threshold>=65..100%,costs=0..2](X11,X12)
```

A.2 The main sequence

You can choose the main sequence in the database, copy your sequence in the text area, or upload one of you files (with fasta or multi-fasta format).

Alternatively, you can also paste an alignment using the Stockholm format with the secondary structure (use the `#=GC` tag for that) into the second text area. Then click on the "Get descriptor" button, and a descriptor is automatically generated, and written in the first text area. You may also edit this descriptor.

A.3 The target sequence

Choose a target sequence in the list, or upload one of your file. Use it only if you used `Y` positions.

Annexe B

Résultats sur les RNAse P

Dans le paragraphe 7.7, nous avons effectué une recherche sur tous les génomes de bactérie séquencés. Cette recherche a trouvé un candidat dans deux candidat dans un génome, quatre cent soixante-neuf génomes, et aucun candidat dans trente-huit génomes.

Deux candidats ont été trouvés dans Salinispora tropica CNB-440.

Les autres candidats ont été trouvés dans :
Acidiphilium cryptum JF-5
Acidobacteria bacterium Ellin345
Acidothermus cellulolyticus 11B
Acidovorax avenae subsp. citrulli AAC00-1
Acidovorax sp. JS42
Acinetobacter baumannii ATCC 17978
Acinetobacter sp. ADP1
Actinobacillus pleuropneumoniae L20
Actinobacillus succinogenes 130Z
Aeromonas hydrophila subsp. hydrophila ATCC 7966
Aeromonas salmonicida subsp. salmonicida A449
Agrobacterium tumefaciens str. C58
Agrobacterium tumefaciens str. C58
Alcanivorax borkumensis SK2
Alkalilimnicola ehrlichei MLHE-1
Alkaliphilus metalliredigens QYMF
Anabaena variabilis ATCC 29413
Anaeromyxobacter dehalogenans 2CP-C
Anaeromyxobacter sp. Fw109-5
Anaplasma marginale str. St. Maries
Anaplasma phagocytophilum HZ
Arthrobacter aurescens TC1
Arthrobacter sp. FB24
Aster yellows witches'-broom phytoplasma AYWB

Azoarcus sp. BH72
Azoarcus sp. EbN1
Bacillus anthracis str. 'Ames Ancestor'
Bacillus anthracis str. Ames
Bacillus anthracis str. Sterne
Bacillus cereus ATCC 10987
Bacillus cereus ATCC 14579
Bacillus cereus subsp. cytotoxis NVH 391-98
Bacillus cereus E33L
Bacillus clausii KSM-K16
Bacillus halodurans C-125
Bacillus licheniformis ATCC 14580
Bacillus licheniformis ATCC 14580
Bacillus subtilis subsp. subtilis str. 168
Bacillus thuringiensis str. Al Hakam
Bacillus thuringiensis serovar konkukian str. 97-27
Bacteroides fragilis NCTC 9343
Bacteroides fragilis YCH46
Bacteroides thetaiotaomicron VPI-5482
Bacteroides vulgatus ATCC 8482
Bartonella bacilliformis KC583
Bartonella henselae str. Houston-1
Bartonella quintana str. Toulouse
Baumannia cicadellinicola str. Hc (Homalodisca coagulata)
Bdellovibrio bacteriovorus HD100
Bordetella bronchiseptica RB50
Bordetella parapertussis 12822
Bordetella pertussis Tohama I
Borrelia afzelii PKo
Borrelia burgdorferi B31
Borrelia garinii PBi
Bradyrhizobium sp. BTAi1
Bradyrhizobium japonicum USDA 110
Bradyrhizobium sp. ORS278
Brucella abortus biovar 1 str. 9-941
Brucella melitensis biovar Abortus 2308
Brucella melitensis 16M
Brucella ovis ATCC 25840
Brucella suis 1330
Buchnera aphidicola str. Sg (Schizaphis graminum)
Buchnera aphidicola str. APS (Acyrthosiphon pisum)
Burkholderia sp. 383
Burkholderia cenocepacia AU 1054
Burkholderia cenocepacia HI2424

Burkholderia cepacia AMMD
Burkholderia mallei ATCC 23344
Burkholderia mallei NCTC 10229
Burkholderia mallei NCTC 10247
Burkholderia mallei SAVP1
Burkholderia pseudomallei 1106a
Burkholderia pseudomallei 1710b
Burkholderia pseudomallei 668
Burkholderia pseudomallei K96243
Burkholderia thailandensis E264
Burkholderia vietnamiensis G4
Burkholderia xenovorans LB400
Caldicellulosiruptor saccharolyticus DSM 8903
Campylobacter fetus subsp. fetus 82-40
Campylobacter jejuni subsp. jejuni 81-176
Campylobacter jejuni subsp. jejuni NCTC 11168
Campylobacter jejuni RM1221
Candidatus Blochmannia floridanus
Candidatus Blochmannia pennsylvanicus str. BPEN
Candidatus Pelagibacter ubique HTCC1062
Candidatus Ruthia magnifica str. Cm (Calyptogena magnifica)
Candidatus Vesicomyosocius okutanii HA
Carboxydothermus hydrogenoformans Z-2901
Caulobacter crescentus CB15
Chlamydia muridarum Nigg
Chlamydia trachomatis A/HAR-13
Chlamydia trachomatis D/UW-3/CX
Chlorobium phaeobacteroides DSM 266
Chlorobium tepidum TLS
Chromobacterium violaceum ATCC 12472
Chromohalobacter salexigens DSM 3043
Clavibacter michiganensis subsp. michiganensis NCPPB 382
Clostridium acetobutylicum ATCC 824
Clostridium beijerinckii NCIMB 8052
Clostridium botulinum A str. ATCC 3502
Clostridium novyi NT
Clostridium perfringens ATCC 13124
Clostridium perfringens str. 13
Clostridium perfringens SM101
Clostridium thermocellum ATCC 27405
Colwellia psychrerythraea 34H
Corynebacterium diphtheriae NCTC 13129
Corynebacterium glutamicum ATCC 13032
Corynebacterium glutamicum ATCC 13032

Corynebacterium glutamicum R
Coxiella burnetii RSA 493
Synechococcus sp. JA-3-3Ab
Synechococcus sp. JA-2-3B'a(2-13)
Cytophaga hutchinsonii ATCC 33406
Dechloromonas aromatica RCB
Dehalococcoides sp. BAV1
Dehalococcoides sp. CBDB1
Dehalococcoides ethenogenes 195
Deinococcus geothermalis DSM 11300
Deinococcus radiodurans R1
Desulfitobacterium hafniense Y51
Desulfovibrio desulfuricans G20
Desulfovibrio vulgaris subsp. vulgaris DP4
Desulfovibrio vulgaris subsp. vulgaris str. Hildenborough
Dichelobacter nodosus VCS1703A
Ehrlichia canis str. Jake
Ehrlichia chaffeensis str. Arkansas
Ehrlichia ruminantium str. Gardel
Ehrlichia ruminantium str. Welgevonden
Ehrlichia ruminantium str. Welgevonden
Enterobacter sp. 638
Enterococcus faecalis V583
Erwinia carotovora subsp. atroseptica SCRI1043
Erythrobacter litoralis HTCC2594
Escherichia coli 536
Escherichia coli APEC O1
Escherichia coli CFT073
Escherichia coli K12
Escherichia coli O157 :H7 EDL933
Escherichia coli O157 :H7 str. Sakai
Escherichia coli UTI89
Escherichia coli W3110 DNA
Flavobacterium johnsoniae UW101
Flavobacterium psychrophilum JIP02/86
Francisella tularensis subsp. tularensis FSC 198
Francisella tularensis subsp. holarctica
Francisella tularensis subsp. holarctica OSU18
Francisella tularensis subsp. novicida U112
Francisella tularensis subsp. tularensis Schu 4
Francisella tularensis subsp. tularensis WY96-3418
Frankia alni ACN14a
Frankia sp. CcI3
Fusobacterium nucleatum subsp. nucleatum ATCC 25586

Geobacillus kaustophilus HTA426
Geobacillus thermodenitrificans NG80-2
Geobacter metallireducens GS-15
Geobacter sulfurreducens PCA
Geobacter uraniumreducens Rf4
Gloeobacter violaceus PCC 7421
Gluconobacter oxydans 621H
Gramella forsetii KT0803
Granulibacter bethesdensis CGDNIH1
Haemophilus ducreyi 35000HP
Haemophilus influenzae 86-028NP
Haemophilus influenzae Rd KW20
Haemophilus influenzae PittEE
Haemophilus influenzae PittGG
Haemophilus somnus 129PT
Hahella chejuensis KCTC 2396
Halorhodospira halophila SL1
Helicobacter acinonychis str. Sheeba
Helicobacter hepaticus ATCC 51449
Helicobacter pylori 26695
Helicobacter pylori HPAG1
Helicobacter pylori J99
Herminiimonas arsenicoxydans
Hyphomonas neptunium ATCC 15444
Idiomarina loihiensis L2TR
Jannaschia sp. CCS1
Janthinobacterium sp. Marseille
Kineococcus radiotolerans SRS30216
Klebsiella pneumoniae subsp. pneumoniae MGH 78578
Lactobacillus acidophilus NCFM
Lactobacillus brevis ATCC 367
Lactobacillus casei ATCC 334
Lactobacillus delbrueckii subsp. bulgaricus ATCC BAA-365
Lactobacillus delbrueckii subsp. bulgaricus ATCC 11842
Lactobacillus gasseri ATCC 33323
Lactobacillus johnsonii NCC 533
Lactobacillus plantarum WCFS1
Lactobacillus reuteri F275
Lactobacillus sakei subsp. sakei 23K
Lactobacillus salivarius subsp. salivarius UCC118
Lactococcus lactis subsp. cremoris MG1363
Lactococcus lactis subsp. cremoris SK11
Lactococcus lactis subsp. lactis Il1403
Lawsonia intracellularis PHE/MN1-00

Legionella pneumophila str. Corby
Legionella pneumophila str. Lens
Legionella pneumophila str. Paris
Legionella pneumophila subsp. pneumophila str. Philadelphia 1
Leifsonia xyli subsp. xyli str. CTCB07
Leptospira borgpetersenii serovar Hardjo-bovis JB197
Leptospira borgpetersenii serovar Hardjo-bovis L550
Leptospira interrogans serovar Copenhageni str. Fiocruz L1-130
Leptospira interrogans serovar Lai str. 56601
Leuconostoc mesenteroides subsp. mesenteroides ATCC 8293
Listeria innocua Clip11262
Listeria monocytogenes str. 4b F2365
Listeria monocytogenes EGD-e
Listeria welshimeri serovar 6b str. SLCC5334
Magnetococcus sp. MC-1
Magnetospirillum magneticum AMB-1
Mannheimia succiniciproducens MBEL55E
Maricaulis maris MCS10
Marinobacter aquaeolei VT8
Marinomonas sp. MWYL1
Mesoplasma florum L1
Mesorhizobium sp. BNC1
Mesorhizobium loti MAFF303099
Methylibium petroleiphilum PM1
Methylobacillus flagellatus KT
Methylococcus capsulatus str. Bath
Moorella thermoacetica ATCC 39073
Mycobacterium avium 104
Mycobacterium avium subsp. paratuberculosis K-10
Mycobacterium bovis BCG str. Pasteur 1173P2
Mycobacterium bovis AF2122/97
Mycobacterium gilvum PYR-GCK
Mycobacterium sp. JLS
Mycobacterium sp. KMS
Mycobacterium leprae TN
Mycobacterium sp. MCS
Mycobacterium smegmatis str. MC2 155
Mycobacterium tuberculosis CDC1551
Mycobacterium tuberculosis F11
Mycobacterium tuberculosis H37Ra
Mycobacterium tuberculosis H37Rv
Mycobacterium ulcerans Agy99
Mycobacterium vanbaalenii PYR-1
Mycoplasma agalactiae PG2

Mycoplasma capricolum subsp. capricolum ATCC 27343
Mycoplasma gallisepticum R
Mycoplasma genitalium G37
Mycoplasma hyopneumoniae 232
Mycoplasma hyopneumoniae 7448
Mycoplasma hyopneumoniae J
Mycoplasma mobile 163K
Mycoplasma mycoides subsp. mycoides SC str. PG1
Mycoplasma penetrans HF-2
Mycoplasma pneumoniae M129
Mycoplasma synoviae 53
Myxococcus xanthus DK 1622
Neisseria gonorrhoeae FA 1090
Neisseria meningitidis FAM18
Neisseria meningitidis MC58
Neisseria meningitidis Z2491
Neorickettsia sennetsu str. Miyayama
Nitratiruptor sp. SB155-2
Nitrobacter hamburgensis X14
Nitrobacter winogradskyi Nb-255
Nitrosococcus oceani ATCC 19707
Nitrosomonas europaea ATCC 19718
Nitrosomonas eutropha C91
Nitrosospira multiformis ATCC 25196
Nocardia farcinica IFM 10152
Nocardioides sp. JS614
Nostoc sp. PCC 7120
Novosphingobium aromaticivorans DSM 12444
Oceanobacillus iheyensis HTE831
Ochrobactrum anthropi ATCC 49188
Onion yellows phytoplasma OY-M
Orientia tsutsugamushi Boryong
Parabacteroides distasonis ATCC 8503
Paracoccus denitrificans PD1222
Pasteurella multocida subsp. multocida str. Pm70
Pediococcus pentosaceus ATCC 25745
Pelobacter carbinolicus DSM 2380
Pelobacter propionicus DSM 2379
Pelodictyon luteolum DSM 273
Pelotomaculum thermopropionicum SI
Photobacterium profundum SS9
Photorhabdus luminescens subsp. laumondii TTO1
Rhodopirellula baltica SH 1
Polaromonas sp. JS666

Polaromonas naphthalenivorans CJ2
Polynucleobacter sp. QLW-P1DMWA-1
Porphyromonas gingivalis W83
Prochlorococcus marinus str. AS9601
Prochlorococcus marinus subsp. marinus str. CCMP1375
Prochlorococcus marinus subsp. pastoris str. CCMP1986
Prochlorococcus marinus str. MIT 9301
Prochlorococcus marinus str. MIT 9303
Prochlorococcus marinus str. MIT 9312
Prochlorococcus marinus str. MIT 9313
Prochlorococcus marinus str. MIT 9515
Prochlorococcus marinus str. NATL1A
Prochlorococcus marinus str. NATL2A
Propionibacterium acnes KPA171202
Prosthecochloris vibrioformis DSM 265
Pseudoalteromonas atlantica T6c
Pseudoalteromonas haloplanktis TAC125
Pseudomonas aeruginosa PAO1
Pseudomonas aeruginosa PA7
Pseudomonas aeruginosa UCBPP-PA14
Pseudomonas entomophila L48
Pseudomonas fluorescens Pf-5
Pseudomonas fluorescens PfO-1
Pseudomonas mendocina ymp
Pseudomonas putida F1
Pseudomonas putida KT2440
Pseudomonas stutzeri A1501
Pseudomonas syringae pv. phaseolicola 1448A
Pseudomonas syringae pv. syringae B728a
Pseudomonas syringae pv. tomato str. DC3000
Psychrobacter arcticus 273-4
Psychrobacter cryohalolentis K5
Psychrobacter sp. PRwf-1
Psychromonas ingrahamii 37
Ralstonia eutropha H16
Ralstonia eutropha JMP134
Ralstonia metallidurans CH34
Ralstonia solanacearum GMI1000
Rhizobium etli CFN 42
Rhizobium leguminosarum bv. viciae 3841
Rhodobacter sphaeroides 2.4.1
Rhodobacter sphaeroides ATCC 17025
Rhodobacter sphaeroides ATCC 17029
Rhodococcus sp. RHA1

Rhodoferax ferrireducens T118
Rhodospirillum rubrum ATCC 11170
Rickettsia bellii RML369-C
Rickettsia prowazekii str. Madrid E
Rickettsia typhi str. Wilmington
Roseobacter denitrificans OCh 114
Rubrobacter xylanophilus DSM 9941
Saccharophagus degradans 2-40
Saccharopolyspora erythraea NRRL 2338
Salmonella enterica subsp. enterica serovar Paratyphi A str. ATCC 9150
Salmonella typhimurium LT2
Salmonella enterica subsp. enterica serovar Typhi str. CT18
Salmonella enterica subsp. enterica serovar Typhi Ty2
Shewanella amazonensis SB2B
Shewanella sp. ANA-3
Shewanella baltica OS155
Shewanella baltica OS185
Shewanella denitrificans OS217
Shewanella frigidimarina NCIMB 400
Shewanella loihica PV-4
Shewanella sp. MR-4
Shewanella sp. MR-7
Shewanella oneidensis MR-1
Shewanella putrefaciens CN-32
Shewanella sp. W3-18-1
Shigella boydii Sb227
Shigella dysenteriae Sd197
Shigella flexneri 2a str. 2457T
Shigella flexneri 2a str. 301
Shigella flexneri 5 str. 8401
Shigella sonnei Ss046
Silicibacter pomeroyi DSS-3
Silicibacter sp. TM1040
Sinorhizobium medicae WSM419
Sinorhizobium meliloti 1021
Sodalis glossinidius str. 'morsitans'
Sphingomonas wittichii RW1
Sphingopyxis alaskensis RB2256
Staphylococcus aureus subsp. aureus MRSA252
Staphylococcus aureus subsp. aureus MSSA476
Staphylococcus aureus subsp. aureus COL
Staphylococcus aureus subsp. aureus JH1
Staphylococcus aureus subsp. aureus JH9
Staphylococcus aureus subsp. aureus Mu50

Staphylococcus aureus subsp. aureus MW2
Staphylococcus aureus subsp. aureus N315
Staphylococcus aureus subsp. aureus NCTC 8325
Staphylococcus aureus subsp. aureus str. Newman
Staphylococcus aureus RF122
Staphylococcus aureus subsp. aureus USA300
Staphylococcus epidermidis ATCC 12228
Staphylococcus epidermidis RP62A
Staphylococcus haemolyticus JCSC1435
Staphylococcus saprophyticus subsp. saprophyticus ATCC 15305
Streptococcus agalactiae 2603V/R
Streptococcus agalactiae A909
Streptococcus agalactiae NEM316
Streptococcus mutans UA159
Streptococcus pneumoniae D39
Streptococcus pneumoniae R6
Streptococcus pneumoniae TIGR4
Streptococcus pyogenes M1 GAS
Streptococcus pyogenes str. Manfredo
Streptococcus pyogenes MGAS10270
Streptococcus pyogenes MGAS10394
Streptococcus pyogenes MGAS10750
Streptococcus pyogenes MGAS2096
Streptococcus pyogenes MGAS315
Streptococcus pyogenes MGAS5005
Streptococcus pyogenes MGAS6180
Streptococcus pyogenes MGAS8232
Streptococcus pyogenes MGAS9429
Streptococcus pyogenes SSI-1
Streptococcus sanguinis SK36
Streptococcus suis 05ZYH33
Streptococcus suis 98HAH33
Streptococcus thermophilus CNRZ1066
Streptococcus thermophilus LMD-9
Streptococcus thermophilus LMG 18311
Streptomyces avermitilis MA-4680
Streptomyces coelicolor A3(2)
Sulfurovum sp. NBC37-1
Synechococcus sp. CC9311
Synechococcus sp. CC9605
Synechococcus sp. CC9902
Synechococcus elongatus PCC 6301
Synechococcus elongatus PCC 7942
Synechococcus sp. RCC307

Synechococcus sp. WH 8102
Synechococcus sp. WH 7803
Synechocystis sp. PCC 6803
Syntrophobacter fumaroxidans MPOB
Syntrophus aciditrophicus SB
Thermoanaerobacter tengcongensis MB4
Thermobifida fusca YX
Thermosipho melanesiensis BI429
Thermosynechococcus elongatus BP-1
Thermotoga maritima MSB8
Thermotoga petrophila RKU-1
Thermus thermophilus HB27
Thermus thermophilus HB8
Thiobacillus denitrificans ATCC 25259
Thiomicrospira crunogena XCL-2
Treponema denticola ATCC 35405
Treponema pallidum subsp. pallidum str. Nichols
Trichodesmium erythraeum IMS101
Tropheryma whipplei TW08/27
Tropheryma whipplei str. Twist
Ureaplasma parvum serovar 3 str. ATCC 700970
Verminephrobacter eiseniae EF01-2
Vibrio cholerae O1 biovar eltor str. N16961
Vibrio cholerae O395
Vibrio fischeri ES114
Vibrio parahaemolyticus RIMD 2210633
Vibrio vulnificus CMCP6
Vibrio vulnificus YJ016
Wigglesworthia glossinidia endosymbiont of Glossina brevipalpis
Wolbachia endosymbiont strain TRS of Brugia malayi
Wolbachia endosymbiont of Drosophila melanogaster
Wolinella succinogenes DSM 1740
Xanthomonas campestris pv. campestris str. 8004
Xanthomonas campestris pv. campestris str. ATCC 33913
Xanthomonas campestris pv. vesicatoria str. 85-10
Xanthomonas axonopodis pv. citri str. 306
Xanthomonas oryzae pv. oryzae KACC10331
Xanthomonas oryzae pv. oryzae MAFF 311018
Xylella fastidiosa 9a5c
Xylella fastidiosa Temecula1
Yersinia enterocolitica subsp. enterocolitica 8081
Yersinia pestis Antiqua
Yersinia pestis biovar Microtus str. 91001
Yersinia pestis CO92

Yersinia pestis KIM
Yersinia pestis Nepal516
Yersinia pestis Pestoides F
Yersinia pseudotuberculosis IP 32953
Zymomonas mobilis subsp. mobilis ZM4

Aucun candidat n'a été trouvé dans :
Aquifex aeolicus VF5
Bifidobacterium adolescentis ATCC 15703
Bifidobacterium longum NCC2705
Buchnera aphidicola str. Cc (Cinara cedri)
Buchnera aphidicola str. Bp (Baizongia pistaciae)
Candidatus Carsonella ruddii PV
Carboxydothermus hydrogenoformans Z-2901
Chlamydophila abortus S26/3
Chlamydophila caviae GPIC
Chlamydophila felis Fe/C-56
Chlamydophila pneumoniae AR39
Chlamydophila pneumoniae CWL029
Chlamydophila pneumoniae J138
Chlamydophila pneumoniae TW-183
Chlorobium chlorochromatii CaD3
Clostridium difficile 630
Clostridium perfringens phage phiSM101
Clostridium tetani E88
Corynebacterium efficiens YS-314
Corynebacterium jeikeium K411
Desulfotalea psychrophila LSv54
Desulfotomaculum reducens MI-1
Mycoplasma pulmonis UAB CTIP
Oenococcus oeni PSU-1
Candidatus Protochlamydia amoebophila UWE25
Rhodopseudomonas palustris BisA53
Rhodopseudomonas palustris BisB18
Rhodopseudomonas palustris BisB5
Rhodopseudomonas palustris CGA009
Rhodopseudomonas palustris HaA2
Rickettsia conorii str. Malish 7
Rickettsia felis URRWXCal2
Roseiflexus sp. RS-1
Salinibacter ruber DSM 13855
Solibacter usitatus Ellin6076
Symbiobacterium thermophilum IAM 14863
Syntrophomonas wolfei subsp. wolfei str. Goettingen

Thiomicrospira denitrificans ATCC 33889

Bibliographie

[AGM+90] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215 :403–410, 1990.

[AHV+01] L. Argaman, R. Hershberg, J. Vogel, G. Bejerano, E. Wagner, H. Margalit, and S. Altuvia. Novel small RNA-encoding genes in the intergenic regions of Escherichia coli. *Current Biology*, 11 :941–950, 2001.

[AKO02] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its application to genome analysis. In *Second Workshop in Algorithms in Bioinformatics*, 2002.

[AKO04] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2 :53–86, 2004.

[AKT93] S. Altman, L. Kirsebom, and S. Talbot. Recent studies of ribonuclease P. *The FASEB Journal*, 7 :7–14, 1993.

[ASBG02] J.-M. Alliot, T. Schiex, P. Brisset, and F. Garcia. *Intelligence artificielle et informatique théorique*. Cépaduès, 2002.

[BCFO04] C. Bessière, R. Coletta, E. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *CP'04*, pages 123–137, 2004.

[BCH02] J.-P. Bachellerie, J. Cavaillé, and A. Hüttenhofer. The expanding snoRNA world. *Biochimie*, 84(8) :775–790, 2002.

[BCKO05] C. Bessière, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *ECML'05*, pages 23–24, 2005.

[BCOP07] C. Bessière, R. Coletta, B O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In *IJCAI'07*, pages 44–49, 2007.

[BCP05] C. Bessière, R. Coletta, and T. Petit. Acquiring parameters of implied global constraints. In *CP'05*, pages 747–751, 2005.

[BCP07] C. Bessière, R. Coletta, and T. Petit. Learning implied global constraints. In *IJCAI'07*, pages 50–55, 2007.

[BCW+04] J. Barrick, K. Corbino, W. Winkler, A. Nahvi, M. Mandal, J. Collins, M. Lee, A. Roth, N. Sudarsan, I. Jona, J. Wickiser, and R. Breaker. New RNA motifs suggest an expanded scope for riboswitches in bacterial genetic control. *PNAS*, 101(17) :6421–6426, 2004.

[BE67] L. Baum and J. Egon. An inequality with applications to statistical estimation for probabilistic functions of a Markov process and to a model for ecology. *Bulletin of the American Meteorological Society*, 73 :360–363, 1967.

[Bes92] C. Bessière. *Systèmes à contraintes évolutifs en Intelligence Artificielle*. PhD thesis, Université des Sciences et Techniques du Languedoc, 1992.

[BFM+99] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs : Frameworks, Properties and Comparison. *Constraints*, 4 :199–240, 1999.

[BG95] A. Brazma and D. Gilbert. A pattern language for molecular biology. Technical report, City University – Department of Computer Science, 1995.

[BKML+07] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and D. Wheeler. Genbank. *Nucleic Acids Research*, 35 :D21–D25, 2007.

[BKV96] B. Billoud, M. Kontic, and A. Viari. Palingol : a declarative programming language to describe nucleic acids' secondary structures and to scan sequence database. *Nucleic Acids Research*, 24(8) :1395–1403, 1996.

[BMR95] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *IJCAI'95*, pages 624–630, 1995.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of the ACM*, 44(2) :201–236, 1997.

[BP66] L. Baum and T. Petrie. Statistical inference for probabilistic functions of state Markov chains. *Annals of Mathematical Statistics*, 37 :1554–1563, 1966.

[BPSW70] L. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Ann. Math. Statist.*, 41(1) :164–171, 1970.

[Bro99] J. Brown. The ribonuclease P database. *Nucleic Acids Research*, 27(1) :314–314, 1999.

[BRYZ05] C. Bessière, J.-C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 2(165) :165–185, 2005.

[BSF96] A. Balakin, L. Smith, and M. Fournier. The RNA world of the nucleolus : two major families of small RNAs defined by different box elements with related functions. *Cell*, 86 :823–834, 1996.

[BWRvdP04] E. Bonnet, J. Wuyts, P. Rouzé, and Y. van de Peer. Evidence that microRNA precursors, unlike other non-coding RNAs, have lower folding free energies than random sequences. *Bioinformatics*, 20(17) :2911–2917, 2004.

[CBL+02] K. Corbino, J. Barrick, J. Lim, R. Welz, B. Tucker, I. Puskarz, M. Mandal, N. Rudnick, and R. Breaker. Evidence for a second class of S-adenosylmethionine riboswitches and other regulatory RNA motifs in alpha-proteobacteria. *Genome Biology*, 6 :R70, 2002.

[CBO+03] R. Coletta, C. Bessière, B. O'Sullivan, E. Freuder, S. O'Connell, and J. Quinqueton. Constraint acquisition as semi-automatic modelling. In *AAAI-03*, pages 111–124, 2003.

[CdGS07] M. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *IJCAI'07*, pages 68–73, 2007.

[CDH01] R. Carter, I. Dubchak1, and S. Holbrook. A computational approach to identify genes for functional RNAs in genomic sequences. *Nucleic Acids Research*, 29(19) :3928–3938, 2001.

[CFOS06] A. Carvalho, A. Freitas, A. Oliveira, and M.-F. Sagot. An efficient algorithm for the identification of structured motifs in DNA promoter sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3 :126–140, 2006.

[CMW03] L. Cai, R. Malmberg, and Y. Wu. Stochastic modeling of RNA pseudoknotted structures : a grammatical approach. *Bioinformatics*, 19 :i66–i73, 2003.

[CNB96] J. Cavaillé, M. Nicoloso, and J.-P. Bachellerie. Targeted ribose methylation of RNA in vivo directed by tailored antisense RNA guides. *Nature*, 383(6602) :732–735, 1996.

[CNC83] E. Comay, R. Nussinov, and O. Comay. An accelerated algorithm for calculating the secondary structure of single stranded RNAs. *Nucleic Acids Research*, 12(1) :53–66, 1983.

[Coo03] M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3) :311–342, 2003.

[Cou02] J. Couzin. Breakthrough of the year : Small RNAs make big splash. *Science*, 298 :2296–2297, 2002.

[CS70] J. Cocke and J. Schwartz. Programming languages and their compilers : Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[CS04] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154 :199–227, 2004.

[DC93] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In *ICLP'93*, pages 774–790, 1993.

[dCSG+06] E. de Castro, C. Sigrist, A. Gattiker, V. Bulliard, P. Langendijk-Genevaux, E. Gasteigner, A. Bairoch, and N. Hulo. ScanProsite : detection of PROSITE signature matches and ProRule-associated functional and structural residues in proteins. *Nucleic Acids Research*, 34 :W362–W365, 2006.

[Dec90] R. Dechter. Enhancement schemes for constraint processing : backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[DEKM98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis : probabilistic models of proteins and nucleic acids*. Cambridge Univ. Press, 1998.

[DJ89] R. Dubes and A. Jain. Random field models in image analysis. *Journal of Applied Statistics*, 16(2) :131–164, 1989.

[DLO97] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *Trends in Genetics*, 13(12), 1997.

[ED94] S. Eddy and R. Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Research*, 22(11) :2079–88, 1994.

[Edd01] S. Eddy. Non-coding RNA genes and the modern RNA world. *Nature Reviews*, 2, 2001.

[EGJ+01] I. Eidhammer, D. Gilbert, I. Jonassen, M. Ratnayake, and S. Grindhaug. A constraint based structure description language for biosequences. *Constraints*, 6(2–3) :173–200, 2001.

[EKW+00] M. Ermolaeva, H. Khalak, O. White, O. Smith, and S. Salzberg. Prediction of transcription terminators in bacterial genomes. *Journal of Molecular Biology*, 301(1) :27–33, 2000.

[EML96] N. El-Mabrouk and F. Lisacek. Very fast identification of RNA motifs in genomic DNA. application to tRNA search in the yeast genome. *Journal of Molecular Biology*, 264 :46–55, 1996.

[Erl78] D. Erlenkotter. A dual-based procedure for uncapacitated facility location. *Operations Research*, 26(6) :992–1009, 1978.

[FL93] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems : a probalistic approach. In *ECSQARU 1993*, pages 97–104, 1993.

[Fon05] A. Fontaine. Détection d'ARN non-codants. Master's thesis, Université de Sciences et Technologies de Lille, 2005.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[GB65] S. Golomb and L. Baumert. Backtrack programming. *Journal of the ACM*, 12(4) :516–524, 1965.

[GBMS95] C. Gaspin, C. Bessière, A. Moisan, and T. Schiex. Satisfaction de contraintes et biologie moléculaire. *Revue d'Intelligence Artificielle*, 9(3) :355–381, 1995.

[GG84] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the bayesian restoration of images. *Transactions on Pattern Analysis and Machine Intelligence*, 6(6) :721–741, 1984.

[GG04] P. Gardner and R. Giegerich. A comprehensive comparison of comparative RNA structure prediction approaches. *BMC Bioinformatics*, 5(1) :140, 2004.

[GHC90] D. Gautheret, L. Heyer, and R. Cedergren. Pattern searching/alignment with RNA primary and secondary structures : an effective descriptor for tRNA. *Computer Applications in the BioSciences*, 6 :325–331, 1990.

[GHS97] J. Gorodkin, L. Heyer, and G. Stormo. Finding the most significant common sequence and structure motifs in a set of RNA sequences. *Nucleic Acids Research*, 25(18) :3724–3732, 1997.

[GJBM+03] S. Griffiths-Jones, A. Bateman, M. Marshall, A. Khanna, and S. Eddy. RFAM : an RNA family database. *Nucleic Acids Research*, 31(1) :439–441, 2003.

[GJMM+05] S. Griffiths-Jones, S. Moxon, M. Marshall, A. Khanna, S. Eddy, and A. Bateman. RFAM : annotating non-coding RNAs in complete genomes. *Nucleic Acids Research*, 33 :D121–D124, 2005.

[GL01] D. Gautheret and A. Lambert. Direct RNA motif definition and identification from multiple sequence alignments using secondary structure profiles. *Journal of Molecular Biology*, 313 :1003–1011, 2001.

[GS93] W. Gish and D. States. Identification of protein coding regions by database similarity search. *Nature Genetics*, 3(3) :266–272, 1993.

[Hay94] S. Haykin. *Neural Networks : a comprehensive Foundation*. IEEE Press, 1994.

[HDT92] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3) :291–321, 1992.

[HE80] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[HFS+94] I. Hofacker, W. Fontana, P. Stadler, L. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatschefte für Chemie / Chemical Monthly*, 125(2) :167–188, 1994.

[HFS02] I. Hofacker, M. Fekete, and P. Stadler. Secondary structure prediction for aligned RNA sequences. *Journal of Molecular Biology*, 319(5) :1059–1066, 2002.

[Hir75] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6) :341–343, 1975.

[HLdGZ05] F. Heras, J. Larrosa, S. de Givry, and M. Zytnicki. Existential arc consistency : getting closer to full arc consistency in weighted CSPs. In *IJCAI'05*, pages 84–89, 2005.

[HSD95] P. Van Hentenryck, V. Saraswat, and Y. Deville. The design, implementation, and evaluation of the constraint language cc(FD). In *Constraint Programming : Basics and Trends*, pages 293–316. Springer Verlag, 1995.

[HTGK03] M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz. Local similarity of RNA secondary structures. In *Proceedings of the IEEE Bioinformatics Conference*, pages 159–168, 2003.

[JCH95] I. Jonassen, J. Collins, and D. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4 :1587–1595, 1995.

[Jon97] I. Jonassen. Efficient discovery of conserved patterns using a pattern graph. *Computer Applications in the BioSciences*, 13(5) :509–522, 1997.

[KAA+07] T. Kulikova, R. Akhtar, P. Aldebert, N. Althorpe, M. Andersson, A. Baldwin, K. Bates, S. Bhattacharyya, L. Bower, P. Browne, M. Castro, G. Cochrane, K. Duggan, R. Eberhardt, N. Faruque, G. Hoad, C. Kanz, C. Lee, R. Leinonen, Q. Lin, V. Lombard, R. Lopez, D. Lorenc, H. McWilliam, G. Mukherjee, F. Nardone, M. Garcia-Pastor, S. Plaister, S. Sobhany, P. Stoehr, R. Vaughan, D. Wu, W. Zhu, and R. Apweiler. EMBL nucleotide sequence database in 2006. *Nucleic Acids Research*, 35 :D16–D20, 2007.

[Kas65] T. Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, 1965.

[Kay93] S. Kay. *Fundamentals of Statistical Signal Processing : Estimation Theory*. Prentice Hall, 1993.

[KB02] L. Krippahl and P. Barahona. PSICO : Solving protein structures with constraint programming and optimization. *Constraints*, 7 :317–331, 2002.

[KE03] R. Klein and S. Eddy. RSEARCH : Finding homologs of single structured RNA sequences. *BMC Bioinformatics*, 4(44), 2003.

[Ken02] J. Kent. BLAT—the BLAST-like alignment tool. *Genome Research*, 12 :656–664, 2002.

[KH99] B. Knudsen and J. Hein. RNA secondary structure prediction using stochastic context-free grammars and evolutionary history. *Bioinformatics*, 15(6) :446–454, 1999.

[KMMR01] L. Khatib, P. Morris, R. Morris, and F. Rossi. Temporal constraint reasoning with preferences. In *IJCAI'01*, 2001.

[Kör89] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 39 :157–173, 1989.

[KP06] A. Kazantsev and N. Pace. Bacterial RNAse P : a new view of an ancient enzyme. *Nature Reviews Microbiology*, 4 :729–740, 2006.

[KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53 :918–936, 2006.

[KSMS01] D. Klein, T. Schmeing, P. Moore, and T. Steitz. The kink-turn : a new RNA secondary structure motif. *EMBO Journal*, 20(15) :4214–4221, 2001.

[KTFL01] J. Kratica, D. Tosic, V. Filipovic, and I. Ljubic. Solving the simple plant location problems by genetic algorithm. *RAIRO Operations Research*, 35 :127–142, 2001.

[Lab00] F. Laburthe. CHOCO : implementing a CP kernel. In *TRICS-2000*, 2000.

[Laf93] A. Laferrière. *Rnamot v2.0*, 1993.

[Lar02] J. Larrosa. On arc and node consistency in weighted CSP. In *Proc. AAAI'02*, 2002.

[LBLW06] J. Livny, A. Brencic, S. Lory, and M. Waldor. Identification of 17 *pseudomonas aeruginosa* sRNAs and prediction of sRNA-encoding genes in 10 diverse pathogens using the bioinformatic tool sRNAPredict2. *Nucleic Acids Research*, 34(12) :3484–3493, 2006.

[LCB+88] S.-Y. Le, J.-H. Chen, M. Braun, M. Gonda, and J. Maizel. Stability of RNA stem-loop structure and distribution of non-random structure in the human immunodeficiency virus (HIV-1). *Nucleic Acids Research*, 16(11) :5153–5168, 1988.

[LCCM98] S. Lemieux, P. Chartrand, R. Cedergren, and F. Major. Modeling active RNA structures using the intersection of conformational space : application to the lead-activated ribozyme. *RNA*, 4 :739–749, 1998.

[LDM94] F. Lisacek, Y. Diaz, and F. Michel. Automatic identification of group I intron cores in genomic DNA sequences. *Journal of Molecular Biology*, 235 :1206–1217, 1994.

[LE97] T. Lowe and S. Eddy. tRNAscan-SE : a program for improved detection of transfer RNA genes in genomic sequence. *Nucleic Acids Research*, 25(5) :955–964, 1997.

[LE99] T. Lowe and S. Eddy. A computational screen for methylation guide snoRNAs in yeast. *Science*, 283, 1999.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10 :707–710, 1966.

[Lew97] B. Lewin. *Genes VI*. Oxford University Press, 1997.

[LFA99] R. Lee, R. Feinbaum, and V. Ambros. The *c. elegans* heterochronic gene lin-4 encodes small RNAs with antisense complementarity to lin-14. *Cell*, 75 :843–854, 199.

[LFDW05] J. Livny, M. Fogel, B. Davis, and M. Waldor. sRNAPredict : an integrative computational approach to identify sRNAs in bacterial genomes. *Nucleic Acids Research*, 33(13) :4096–4105, 2005.

[LFL+04] A. Lambert, J.-F. Fontaine, M. Legendre, F. Leclerc, E. Permal, F. Major, H. Putzer, O. Delfour, B. Michot, and D. Gautheret. The ERPIN server : an interface to profile-based RNA motif identification. *Nucleic Acids Research*, 32 :W160–W165, 2004.

[LG03] M. Legendre and D. Gautheret. Sequence determinants in human polyadenylation site selection. *BMC Genomics*, 4(7), 2003.

[LGC94] A. Laferrière, D. Gautheret, and R. Cedergren. An RNA pattern matching program with enhanced performance and portability. *Computer Applications in the BioSciences*, 10(2) :211–212, 1994.

[Lho93] O. Lhomme. Consistency techniques for numeric CSPs. In *IJCAI'93*, pages 232–238, 1993.

[Lho03] O. Lhomme. Efficient filtering algorithm for disjunction of constraints. In *CP'03*, pages 904–908, 2003.

[LLFG05] A. Lambert, M. Legendre, J.-F. Fontaine, and D. Gautheret. Computing expectation values for RNA motifs using discrete convolutions. *BMC Bioinformatics*, 6(118), 2005.

[LLG05] M. Legendre, A. Lambert, and D. Gautheret. Profile-based detection of microRNA precursors in animal genomes. *Bioinformatics*, 21(7) :841–845, 2005.

[LLMW05] A. Lescoute, L. Leontis, C. Massire, and E. Westhof. Recurrent structural RNA motifs, isostericity matrices and sequence alignments. *Nucleic Acids Research*, 33(8) :2395–2409, 2005.

[LS03] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. In *IJCAI'03*, 2003.

[LSW02] N. Leontis, J. Stombaugh, and E. Westhof. The non-Watson-Crick base pairs and their associated isostericity matrices. *Nucleic Acids Research*, 30(16) :3497–3531, 2002.

[LY90] K. Lari and S. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4 :35–56, 1990.

[Mac77] A. Mackworth. Consistency in networks of constraints. *Artificial Intelligence*, 8 :99–118, 1977.

[Mar02] L. Marsan. *Inférence de motifs structurés : algorithmes et outils appliqués à la détection de sites de fixation dans les séquences génomiques.* PhD thesis, Université de Marne-La-Vallée, 2002.

[MB04] M. Mandal and R. Breaker. Gene regulation by riboswitches. *Nature Reviews Molecular Cellular Biology*, 5 :451–463, 2004.

[MC01] T. Macke and D. Case. *RNAMotif Users' Manual*, 2001.

[MEG+01] T. Macke, D. Ecker, R. Gutell, D. Gautheret, D. Case, and R. Sampath. Rnamotif, an RNA secondary structure definition and search algorithm. *Nucleic Acids Research*, 29(22) :4724–4735, 2001.

[MGC93] F. Major, D. Gautheret, and R. Cedergren. Reproducting the three-dimensional structure of a tRNA molecule from structural constraints. *PNAS*, 90 :9408–9412, 1993.

[MGEW93] G. Muller, C. Gaspin, A. Etienne, and E. Westhof. Automatic display of RNA secondary structures. *Computer Applications in the BioSciences*, 9(5) :551–561, 1993.

[MH04] L. Michel and P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal on Operations Research*, 157(3) :576–591, 2004.

[Mit82] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18 :203–226, 1982.

[MJW98] C. Massire, L. Jaeger, and E. Westhof. Derivation of the three-dimensional architecture of bacterial ribonuclease P RNAs from comparative sequence analysis. *Journal of Molecular Biology*, 279(4) :773–793, 1998.

[MM93] G. Mehldau and G. Myers. A system for pattern matching applications on biosequences. *Computer Applications in the BioSciences*, 9(3) :299–314, 1993.

[Mon74] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Sciences*, 7 :95–132, 1974.

[MSS05] H. Matsui, K. Sato, and Y. Sakakibara. Pair stochastic tree adjoining grammars for aligning and predicting pseudoknot RNA structures. *Bioinformatics*, 21(11), 2005.

[MT02] D. Mathews and D. Turner. Dynalign : an algorithm for finding the secondary structure common to two RNA sequences. *Journal of Molecular Biology*, 317(2) :171–323, 2002.

[MTG+91] F. Major, M. Turcotte, D. Gautheret, G. Lapalme, E. Fillion, and R. Cedergren. The combination of symbolic and numerical computation for three-dimensional modeling of RNA. *Science*, 253(5025) :1255–1560, 1991.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings — Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[NSF02] P. Nicodème, B. Salvy, and P. Flajolet. Motif statistics. *Theoretical Computer Science*, 287 :593–617, 2002.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3) :443–453, 1970.

[OLR+00] A. Omer, T. Lowe, A. Russell, H. Ebhardt, S. Eddy, and P. Dennis. Homologs of small nucleolar RNAs in Archaea. *Science*, 288(5465) :517–522, 2000.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[PB95] N. Pace and J. Brown. Evolutionay perspective on the structure and function of ribonuclase P, a ribozyme. *Journal of Bacteriology*, 177(8) :1919–1928, 1995.

[PCMS06] N. Pisanti, A. Carvalho, L. Marsan, and M.-F. Sagot. Risotto : Fast extraction of motifs with mismatches. In *LATIN'06*, pages 757–768, 2006.

[PDF04] A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5 :186, 2004.

[Per04] D. Pervouchine. IRIS : intermolecular RNA interaction search. *Genome Informatics*, 15 :92–101, 2004.

[PL88] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. *PNAS*, 85 :2444–2448, 1988.

[Ple94] C. Pleij. RNA pseudoknots. *Current Opinion in Structural Biology*, 4 :337–344, 1994.

[PRB00] T. Petit, J.-C. Régin, and C. Bessière. Meta-constraints on violations for over constrained problems. In *ECTAI'00*, pages 358–365, 2000.

[RBW06] F. Rossi, P. Van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[RE99] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285(5) :2053–2068, 1999.

[RE01] E. Rivas and S. Eddy. Noncoding RNA gene detection using comparative sequence analysis. *BMC Bioinformatics*, 2(1) :19, 2001.

[Rég95] J.-C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* PhD thesis, Université Montpellier II, 1995.

[Rég96] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI-96*, pages 209–215, 1996.

[RHZ76] A. Rosenfeld, R. Hummel, and S. Zucker. Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man and Cybernetics*, 6(6) :173–184, 1976.

[RKJE01] E. Rivas, R. Klein, T. Jones, and S. Eddy. Computational identification of noncoding RNAs in E. coli by comparative genomics. *Current Biology*, 11 :1369–1373, 2001.

[RRG07] J. Reeder, J. Reeder, and R. Giegerich. Locomotif : from graphical motif description to RNA motif search. *Bioinformatics*, 23(13) :i392–i400, 2007.

[RS04] F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4), 2004.

[RSV+02] F. Rossi, A. Sperduti, K. Venable, L. Khatib, P. Morris, and R. Morris. Learning and solving soft temporal constrains : An experimental study. In *CP'02*, 2002.

[Ruv01] G. Ruvkun. Glimpses of a tiny RNA world. *Science*, 294(5543) :797–799, 2001.

[SAB+77] F. Sanger, G. Air, B. Barrell, N. Brown, A. Coulson, C. Fiddes, C. Hutchison, P. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage Φ X174 DNA. *Nature*, 265 :687–698, 1977.

[San85] D. Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics*, 45(5) :810–825, 1985.

[SB05] S. Siebert and R. Backofen. MARNA : multiple alignment and consensus structure prediction of RNAs based on sequence structure comparisons. *Bioinformatics*, 21(16) :3352–3359, 2005.

[SBH+94] Y. Sakakibara, M. Brown, R. Hughey, I. Mian, K. Sjölander, R. Underwood, and D. Haussler. Recent methods for RNA modeling using stochastic context-free grammars. In *CPM'94*, pages 289–306, 1994.

[Sch92] T. Schiex. Possibilistic constraint satisfaction problems or "how to handle soft contraints ?". In *UAI'92*, pages 268–275, 1992.

[Sch00] T. Schiex. Arc consistency for soft constraints. In *CP'00*, pages 411–424, 2000.

[Sch02a] P. Schattner. Searching for RNA genes using base-composition statistics. *Nucleic Acids Research*, 30(9) :2076–2082, 2002.

[SCH+02b] C. Sigrist, L. Cerutti, N. Hulo, A. Gattiker, L. Falquet, M. Pagni, A. Bairoch, and P. Bucher. PROSITE : A documented database using patterns and profiles as motifs descriptors. *Briefings in Bioinformatics*, 3(3) :265–274, 2002.

[Sch03] P. Schattner. *Computational gene-finding for non-coding RNAs*, pages 33–48. Landes Bioscience, 2003.

[SFV95] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : hard and easy problems. In *IJCAI'95*, pages 631–637, 1995.

[SG94] D. States and W. Gish. Combined use of sequence similarity and codon bias for coding region identification. *Journal of Computational Biology*, 1(1) :39–50, 1994.

[TdGSG05] P. Thébault, S. de Givry, T. Schiex, and C. Gaspin. Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences. In *5th workshop on modelling and solving problems with constraints*, 2005.

[TdGSG06] P. Thébault, S. de Givry, T. Schiex, and C. Gaspin. Searching RNA motifs and their intermolecular contacts with constraint networks. *Bioinformatics*, 22(17) :2074–2080, 2006.

[Thé04] P. Thébault. *Formalisme CSP et localisation de motifs structurés dans les textes génomiques*. PhD thesis, Université Toulouse III, 2004.

[THG94] J. Thompson, D. Higgins, and T. Gibson. 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(22) :4673–4680, 1994.

[Tou05] H. Touzet. Analyse comparative pour l'étude des gènes d'ARN. In *Journées ARéNa 2005*, Toulouse, 2005.

[TP04] H. Touzet and O. Perriquet. CARNAC : folding families of related RNAs. *Nucleic Acids Research*, 32 :142–145, 2004.

[Ukk83] E. Ukkonen. On approximate string matching. In *International Conference on Foundations of Computation Theory*, pages 487–495, 1983.

[Via04] S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theoretical Computer Science*, 312(2-3) :223–249, 2004.

[Vit67] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2) :260–269, 1967.

[VRMG02] A. Vitreschak, D. Rodionov, A. Mironov, and M. Gelfand. Regulation of riboflavin biosynthesis and transport genes in bacteria by transcriptional and translational attenuation. *Nucleic Acids Research*, 30(14) :3141–3151, 2002.

[VRMG04] A. Vitreschak, D. Rodionov, A. Mironov, and M. Gelfand. Riboswitches : the oldest mechanism for the regulation of gene expression ? *Trends in Genetics*, 20(1) :44–50, 2004.

[Wal96] M. Wallace. Practical applications of constraint programming. *Constraints*, 1 :139–168, 1996.

[WB03] W. Winkler and R. Breaker. Genetic control by metabolite-binding riboswitches. *ChemBioChem*, 4 :1024–1032, 2003.

[WB05] W. Winkler and R. Breaker. Regulation of bacterial gene expression by riboswitches. *Annual Review of Microbiology*, 59 :487–517, 2005.

[WH07] M. Wieland and J. Hartig. RNA quadruplex-based modulation of gene expression. *Chemistry & Biology*, 14 :757–763, 2007.

[WHS05] S. Washietl, I. Hofacker, and P. Stadler. Fast and reliable prediction of noncoding RNAs. *PNAS*, 102(7) :2454–2459, 2005.

[WOHN06] I. Wallace, O. O'Sullivan, D. Higgins, and C. Notredame. M-Coffee : combining multiple sequence alignment methods with T-Coffee. *Nucleic Acids Research*, 34(6) :1692–1699, 2006.

[You67] D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2) :189–208, 1967.

[ZGS06a] M. Zytnicki, C. Gaspin, and T. Schiex. A new local consistency for weighted CSP dedicated to long domains. In *SAC'06 : Proceedings of the 2006 ACM symposium on Applied computing*, pages 394–398, 2006.

[ZGS06b] M. Zytnicki, C. Gaspin, and T. Schiex. Suffix arrays and weighted csps. In *WCB'06*, 2006.

[ZGS07] M. Zytnicki, C. Gaspin, and T. Schiex. DARN ! a soft constraint solver for RNA motif localisation. In *JOBIM 2007*, pages 51–56, 2007.

[ZM07] R. Zivan and A. Meisels. Conflict directed backjumping for Max-CSPs. In *IJCAI'07*, pages 198–204, 2007.

[ZS81] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1) :133–148, 1981.

Résumé

La recherche d'ARN non-codants (ARNnc) a reçu un regain d'intérêt suite à la découverte de nouveaux types d'ARNnc aux fonctions multiples. De nombreuses techniques ont été développées pour localiser ces ARN dans des séquences génomiques. Nous utilisons ici une approche supposant la connaissance d'un ensemble d'éléments de structure discriminant une famille d'ARNnc appelé signature.

Dans cette approche, nous combinons plusieurs techniques de *pattern-matching* avec le formalismes des réseaux de contraintes pondérées afin de modéliser simplement le problème, de décrire finement les signatures et d'attribuer un coût à chaque solution. Nos travaux nous ont conduit à élaborer plusieurs techniques de filtrage ainsi que des algorithmes de *pattern-matching* originaux que nous présentons ici.

Nous avons de plus conçu un logiciel, appelé DARN !, qui implante notre approche, ainsi qu'un module de génération de signatures. Ceux-ci permettent de rechercher efficacement de nouveaux ARNnc.