La.M.I. - Équipe Bio-Info

Laboratoire de Méthodes Informatiques

C.N.R.S. U.M.R. 8042

Rapport de stage de D.E.A.

Application des Mathématiques et de l'Informatique à la Biologie

Mécanisme d'annotation par recherche de K plus courts chemins

Matthieu MANCENY

Encadrant : Franck DELAPLACE, Maître de conférence, Université d'Évry-Val d'Essonne

26 juin 2003

LaMI - Université d'Évry-Val d'Essonne
Tour Evry 2 / 2ème étage
523 place des Terrasses
91025 Evry - France

Remerciements

Je tiens tout d'abord à remercier mon maître de stage, Franck Delaplace, pour sa grande disponibilité, ses idées et suggestions, son assistance pour le langage *OCaml*, et de manière générale, pour toute l'aide qu'il a pu m'apporter durant le stage.

Je remercie également toute l'équipe de stagiaires, et en particulier Céline, Christophe et Adrien pour avoir répondu à mes questions ainsi qu'Antoine pour les moments de détente.

Merci enfin à toute l'équipe bio-info du La.M.I. pour m'avoir permis d'effectuer ce stage dans une ambiance agréable.

Résumé

Le séquençage de l'ADN, c'est-à-dire la lecture des « lettres » constituant l'ADN, a débuté dans les années 1970. Aujourd'hui les avancées technologiques et l'industrialisation du séquençage permettent celui de millions de bases d'ADN. C'est ainsi que l'année 2003 a vu le séquençage complet du génome humain.

Mais ces données sont des donnés brutes, de quelle utilité sont-elles si on ne peut les comprendre ? L'interprétation des données issues du séquençage, appelée *annotation*, est donc l'etape suivante à la compréhension du génome humain.

C'est dans le cadre de l'annotation qu'a été développé le langage TAGCC.

Basé sur des méthodes de recherche de gènes se fondant sur des données statistiques et sur la structure grammaticale de l'ADN, TAGCC segmente la séquence d'ADN traitée en plaçant des marques, des balises le long de cette séquence. Le placement de ces balises permet de définir différentes solutions, différentes annotations qui définissent elles-mêmes autant de gènes possibles dans la séquence.

Mais l'ensemble des solutions obtenues est un espace combinatoire, on ne peut donc toutes les rechercher. De plus certains phénomènes biologiques font que la recherche de la meilleure solution n'est pas suffisante. C'est pourquoi la recherche de K solutions est nécessaire, ce qui est un problème NP-Complet.

Pour résoudre ce problème, les solutions obtenues avec TAGCC sont représentées sous forme de graphe, le graphe *Marques-Positions*, où chaque sommet représente le fait de mettre une balise à une certaine position de la séquence, et les arcs les possibilités de passages entre ces balises. La recherche de K solutions revient alors à rechercher K chemins dans ce graphe.

En relâchant les contraintes imposées aux chemins cherchés, et du fait de l'annotation et de TAGCC, le graphe *Marques-Positions* est acyclique, ce qui nous permet d'appliquer l'algorithme *MS*, développé par Martins et dos Santos, pour la recherche des K plus courts chemins (Martins, E.d.Q.V. et dos Santos, J.L.E. (1999). A new shortest paths ranking algorithm. *Investigação Operational*). Cette recherche s'effectue en un temps $O(K.n+m)$, où n est le nombre de sommets du graphe *Marques-Positions* et m le nombre d'arcs.

Le stage a permis d'une part de créer une librairie générique pour traiter les graphes en *OCaml*, et d'autre part d'implémenter cette librairie dans TAGCC, afin d'effectuer la recherche de K solutions.

Table des matières

Introduction

L'analyse des génomes représente un enjeu scientifique d'importance dont on espère de nombreuses retombées, notamment dans les domaines médicaux, pharmaceutiques, agroalimentaires et biotechnologiques. La systématisation de cette analyse, sa complexité et la taille des données à analyser concourent à donner à l'informatique une place centrale pour un traitement « à grande échelle » des séquences biologiques.

Le *séquençage* constitue la première étape de ce traitement. Débuté dans les années 1970, celui-ci est entré dans les années 1990 dans une phase industrielle de numérisation massive, mettant ainsi à disposition de la communauté scientifique un nombre croissant de séquences. En effet, aujourd'hui les avancées technologiques dans le domaine du séquençage permettent de réaliser chaque jour celui de millions de bases d'ADN provenant de différents organismes.

Mais de quelle utilité seraient ces données si nous étions incapables de les comprendre ?

Dans les années 2000, l'« industrialisation » du séquençage met à présent l'éclairage sur l'interprétation de ces données ouvrant l'ère de la post-génomique. Le séquençage complet du génome humain en 2003 laisse présager des avancées phénoménales ; une telle perspective doit enthousiasmer et inciter à déployer toutes les méthodes pour tenter d'interpréter ces données brutes.

Cette interprétation implique de repérer les régions *significatives* d'un génome, qui se déterminent par rapport aux fonctions biologiques qu'elles assument. L'attribution d'une fonction à une région ou, inversement, la recherche d'une région possédant une fonction biologique particulière, se nomme *l'annotation* (voir le chapitre I.1). Il s'agit d'interpréter et d'inventorier systématiquement les différentes fonctions des régions d'un génome. Dans cet inventaire, la recherche de gènes possède une place centrale.

TAGCC est un langage dédié à l'annotation par des méthodes *ab-initio* (voir le chapitre I.2). Ces méthodes découvrent de nouveaux gènes par des critères heuristiques, basés sur des statistiques et sur des connaissances biologiques de la structure du gène. Elles se fondent sur la recherche de *signaux* qui correspondent à des zones biologiquement fonctionnelles à l'intérieur de la séquence d'ADN. L'ensemble de ces signaux permet de définir une structure pour la séquence.

TAGCC utilise une méthode de *segmentation* qui consiste à découper la séquence en y indiquant des points stratégiques. Ces points stratégiques s'identifient par des marques, des balises qui repèrent les signaux.

Mais la segmentation ne fournit pas une solution unique. En effet, un signal est défini par des critères heuristiques, il peut donc se situer à divers endroits de la séquence. À partir d'un programme écrit en TAGCC, il est possible d'obtenir l'ensemble des solutions compatibles avec la structure de la séquence.

Se pose alors la question de comparer ces solutions entre elles.

Usuellement en informatique, la comparaison se fonde sur une fonction objective de coût. Pratiquement, ce coût est un score de défiance en l'annotation : plus le score est petit, meilleure est l'annotation. Les coûts des solutions sont déterminés par des critères biologiques ou statistiques. et permettent de comparer les solutions entre elles.

Toutefois la recherche d'une unique solution, la meilleure soit-elle, ne suffit pas. En effet, l'annotation est fondées sur des heuristiques et nécessite donc la recherche de plusieurs solutions, qui devront être soumises à l'expertise d'un biologiste. De plus, la recherche d'une unique solution ne prend pas en compte le phénomène de l'épissage alternatif. Lors de l'épissage (durant la phase de maturation, voir le chapitre I.1) l'excision des introns n'est pas unique et fournit des solutions distinctes qui mèneront à la synthèse de protéines différentes. Ces solutions ne peuvent pas être comparées entre elles, bien qu'elles proviennent de la même séquence. Pour prendre en compte ce phénomène, il est nécessaire non seulement de rechercher la meilleure solution, mais également les suivantes.

Une méthode algorithmique permettant d'énumérer les K meilleures solutions doit alors être trouvée.

Une séquence annotée, segmentée, correspond finalement à un ensemble de marques, de balises, placées à différentes positions. Le couple marque-position identifie le placement de la balise. Ce couple constitue une partie de la solution en servant de pivot à la segmentation de la séquence.

Si on considère l'ensemble de toutes les marques-positions possibles pour une séquence, il est alors nécessaire, pour représenter une annotation particulière, de choisir un sous-ensemble de ces marques. Une manière de représenter ce choix, est de relier entre elles les marques-positions composant l'annotation.

L'espace des solutions est ainsi modélisé par un graphe, appelé graphe *Marques-Positions*, où un sommet représente la possibilité de mettre une certaine marque à une certaine position, et un arc, le passage entre ces marques/positions. Les arcs sont naturellement valués par la fonction de coût représentant la défiance en l'annotation.

L'énumération des K meilleures solutions revient à rechercher les K plus courts chemins dans le graphe Marques-Positions. Dans le cas général, ce problème est NP-Complet. Autrement dit, il n'existe pas d'algorithme polynomial pour le résoudre (voir le chapitre II.1).

L'objectif de ce stage était double. D'une part réaliser une librairie en *OCaml* afin de gérer les graphes et d'autre part, mettre au point un algorithme de recherche des K plus courts chemins et l'implémenter dans le logiciel TAGCC.

Le rapport se découpe de la manière suivante.

La partie I concerne les mécanismes d'annotation.
Le chapitre 1 s'intéresse à l'annotation (sa définition, les méthodes existantes), à l'ADN (sa structure, les différentes étapes de l'expression d'un gène).
Le chapitre 2 présente le logiciel TAGCC, dédié à l'annotation par des méthodes *ab-initio*.

La partie II présente un état de l'art sur la recherche des K plus courts chemins dans un graphe.

La partie III concerne les spécificités de l'annotation dans l'optique de la recherche des K plus courts chemins.
Le chapitre 1 présente le graphe *Marques-Positions*, sa topologie, comment le construire.
Le chapitre 2 s'intéresse quant à lui à un algorithme de recherche des K plus courts chemins applicable sur le graphe *Marques-Positions* : l'algorithme *MS*.

Enfin la partie IV concerne les tests réalisés afin de valider l'application.

Le rapport se conclut sur le travail effectué et son utilité.

En annexe sont présentés les algorithmes implémentés ainsi que la librairie de graphe.

Première partie

Mécanisme d'annotation

Chapitre 1

ADN et annotation

Ce chapitre s'intéresse à l'annotation. Nous en voyons les principes et les différentes méthodes avant d'en venir aux signaux permettant les mécanismes d'expression des gènes.

1.1 Principes de l'annotation

Le séquençage d'un génome produit des données de séquences brutes ; c'est-à-dire, des données correspondant à la transcription en informatique des suites de bases A T G C composant l'ADN.

Ces données brutes ne sont pas exploitables directement, il est nécessaire de les interpréter, autrement dit de repérer les différentes régions constituant la séquence. Ces régions se distinguent les unes des autres par les différentes fonctions biologiques qu'elles assument.

L'étude et la recherche des régions du génome se nomment *annotation* :

- l'*annotation syntaxique* consiste à déterminer et identifier la structure des gènes,
- l'*annotation fonctionnelle* s'intéresse à déterminer et identifier la fonction des gènes,
- les relations entre les entités biologiques relatives au génome sont étudiées par l'*annotation relationnelle*.

On distingue deux grandes classes de méthodes algorithmiques pour la recherche et la prédiction de gènes :

- les méthodes par homologie,
- les méthodes *ab-initio*.

1.1.1 Méthodes par homologie

Les méthodes par homologie sont considérées comme des standards pour la prédiction de gènes. Elles identifient les gènes par similarité avec d'autres gènes déjà découverts dans des génomes différents de celui étudié. Ces méthodes se fondent sur deux hypothèses :

- on considère que des gènes ayant une stucture similaire codent pour des protéines ayant des fonctions proches, l' homologie correspond à une analogie ;
- on considère que les fonctions dont dépendent un organisme vivant se retrouvent d'une espèce à l'autre, on peut donc ainsi rapprocher et comparer la structure de gènes entre différents génomes.

Le succès et la fiabilité d'une recherche par homologie repose donc sur l'existence d'un gène de référence dans un génome. Ceci exclut de fait la découverte de nouveaux gènes, des gènes ne possédant pas d'homologues connus tels que ceux codants pour des protéines intervenant dans des maladies rares.

1.1.2 Méthodes *ab-initio*

Les méthodes *ab-initio* découvrent des gènes par des critères permettant de distinguer les régions codantes des régions non-codantes. Ces critères se déterminent un calcul de biais statistique sur les gènes et sur des connaissances biologiques de la structure d'un gène.

Méthodes par contenu

Afin de distinguer les régions codantes des régions non codantes, les mesures *ab-initio* par contenu se fondent sur l'existence d'un biais statistique dans la partie exonique, qui n'existe pas dans la partie intergénique, ou intergénique + introns (voir la section 1.2).

Certaines mesures dépendent d'un modèle d'ADN codant. Elles se fondent sur une modèle probabilistique décrivant une séquence codante type. Plus leur évaluation est précise, plus on peut avoir confiance dans la prédiction codant/non-codant.

D'autres mesures se basent sur des caractéristiques intrasèque de l'ADN. Elles reposent sur l'hypothèse que les séquences non-codantes sont homogènes, c'est-à-dire qu'il y a équiprobabilité d'apparition pour les nucléotides aux différentes positions. Par opposition, les séquences codantes sont hétérogènes et présentent un biais sur les positions.

Méthodes par signaux

Les mesure *ab-initio* par signaux se fondent sur la détection de courtes séquences qui jouent un rôle particulier, et parfois essentiel, dans le processus de transcription ou de traduction.

Schématiquement, un gène possède une structure « grammaticale » qui est représentative des opérations de production d'une protéine. La reconnaissance des signaux permet la reconnaissance de cette grammaire dans la séquence afin d'en isoler les régions codantes (voir la section 1.2).

On distingue deux catégories de signaux :

- les signaux dits « *durs* », pour lesquels il existe un nombre fini de variations, par exemple les signaux de début et de fin de séquences traduites, les codons `START` et `STOP` ;
- les signaux dits « *mous* », pour lesquels il existe un nombre de variations non fixé, par exemple les sous-séquences du promoteur. Ces signaux sont définis par leurs fonctions et non par leur structure.

La détection de ces deux type de signaux est donc différente, puisque dans le cas des signaux durs, on peut les identifier par une grammaire, alors que dans le cas des signaux mous, ce n'est pas possible.

Les signaux mous peuvent être identifiés par une *méthode de matrice de poids*. Cette méthode s'appuie sur l'étude des fréquences d'apparition des nucléotides en fonction des positions. Plus précisément, à partir d'un échantillon de séquences représentant le même signal (c'est-à-dire ayant la même fonction) on procède à leur alignement (supposé de longueur n), puis on relève la probabilité $p(b, i)$ d'apparition de la base b (pour chaque b dans {A, T, G, C}) en position $i \in [1, n]$.

La reconnaissance d'un signal consensus par une méthode du ratio de log-vraisemblance consiste à discriminer un signal en examinant le rapport entre la mesure statistique d'appartenir à un modèle et celle de son contre modèle.

Étant donnée une séquence S, on détermine $P(S|M)$ la probabilité qu'une séquence observée correspond au modèle M (à un promoteur par exemple). $P(S|\bar{M})$ correspond à la probabilité de ne pas appartenir au modèle M.

On note par $p_{(b,i)}$ la probabilité d'occurrence d'un nucléotide b en position i dans le signal et par p_b la probabilité d'apparition d'un nucléotide b dans le génome de l'organisme étudié.

Celui-ci peut dévier significativement d'une mesure d'équiprobalité (0.25).

L'identification du signal par la méthode du ratio de log-vraisemblance correspond à la fraction

$$r = \log \frac{P(S|M)}{P(S|\bar{M})} = \sum_{i=1}^{|S|} \log \frac{p_{(b,i)}}{p_b}$$

En posant $\beta_{b,i} = log\frac{p_{(b,i)}}{p_b}$, on a $r = \sum_{i=1}^{|S|} \beta_{i,b}$. si r est positif le signal fait partie du modèle. β correspond à une matrice de poids.

1.2 De l'ADN à la protéine : la recherche des signaux

Pour qu'un gène s'exprime, une protéine doit être synthétisée. Le mécanisme de synthèse est relativement complexe. Celui-ci débute par la transcription de l'ADN en ARN pré-messager, qui subit alors une maturation pour devenir de l'ARN messager, avant d'être traduit en protéine. Des marques tout au long de l'ADN, les signaux, permettent les différents processus moléculaires réalisant ces étapes. Ces signaux peuvent également permettre aux biologistes de se repérer, et de pouvoir distinguer les gènes des régions non codantes, il est donc indipensable de bien les identifier. Mais si à partir d'une fonction on peut mettre en évidence des signaux, la réciproque est fausse, en effet la présence de signaux ne suffit pas à identifier la fonction d'une région de l'ADN.

1.2.1 De l'ADN à l'ARN pré-messager : transcription

L'expression d'un gène inscrit dans l'ADN en protéine nécessite la synthèse préalable d'un ARN par le mécanisme de transcription. La protéine responsable de cette synthèse, l'ARN polymérase, utilise des signaux (*i.e.* des séquences particulières) sur la séquence d'ADN pour déclencher le processus de transcription conduisant à la fabrication d'un ARN pré-messager. Différents signaux ou boîtes (GC, CAAT, TATA) composent le *promoteur* qui déterminent le début de la transcription. Ces signaux sont mous, ils sont définis par leur fonction, et non par leur syntaxe ; il peut donc être délicat de les identifier.

1.2.2 De l'ARN pré-messager à l'ARN messager : maturation

Lors de la maturation l'ARN pré-messager voit ses extrémités modifiées, et subit un épissage, qui correspond à l'excision des introns et à la réunion des exons.

Modification des extrémités

Les extrémités de l'ARN pré-messager sont modifiées, on obtient ainsi deux régions UTR (UnTranslated Regions) qui ne seront pas traduites.

Épissage, exons et introns

Les exons sont les parties de l'ARN pré-messager qui vont effectivement coder pour une partie de la protéine. Les introns quant à eux sont les parties non codantes, on ne sait pas aujourd'hui à quoi ils servent.

L'ARN pré-messager est une succession d'exons et d'introns. Des signaux les séparent. L'épissage est le fait de réunir les exons en supprimant, excisant, les introns. Ainsi, entre un exon et un intron, le signal est un site donneur d'épissage, `SDE`, alors qu'entre un intron et un exon, on parle de site accepteur d'épissage, `SAE`. Le site donneur correpond à la succession des nucléotides G T, les site accepteur aux nucléotides A G. Ces signaux sont des signaux durs.

Épissage alternatif

Pour un même gène et dans un même organisme, l'élimination des introns peut être différente selon la cellule concernée. Ainsi, pour un même gène, la combinaison des exons sera différente, l'ARN messager sera différent et donnera naissance à une protéine différente. Ce phénomène est un phénomène d'économie pour la cellule, qui est omniprésent chez tous les eucaryotes pluricellulaires. Aussi, la recherche de la meilleure annotation, quel que soit le critère, ne peut suffire dans le cas de l'épissage alternatif, car les différentes solutions ne sont pas toutes comparables entre elles.

On estime que plus que 50% des gènes humains sont épissés de façon alternative. Comme le nombre de variants d'épissage produits à partir d'un seul gène varie entre deux et plusieurs milliers on évalue que l'épissage alternatif pourrait être impliqué dans la génèse de près de 95% des protéines humaines ([Infobiogen, 2003]).

1.2.3 De l'ARN messager à la protéine : traduction

Cette phase correspond à la traduction de l'ARN messager en protéine. Les processus de traduction opèrent par groupe de trois nucléotides. Le codon `START` (ATG) marque le début de la traduction, et le codon `STOP` (TAA, TGA ou TAG) la fin. La séquence située entre ces deux codons est appelée CDS pour CoDing Sequence. Ces codons sont des signaux durs.

La figure 1.1 résume l'ensemble de ce mécanisme et de ces signaux.

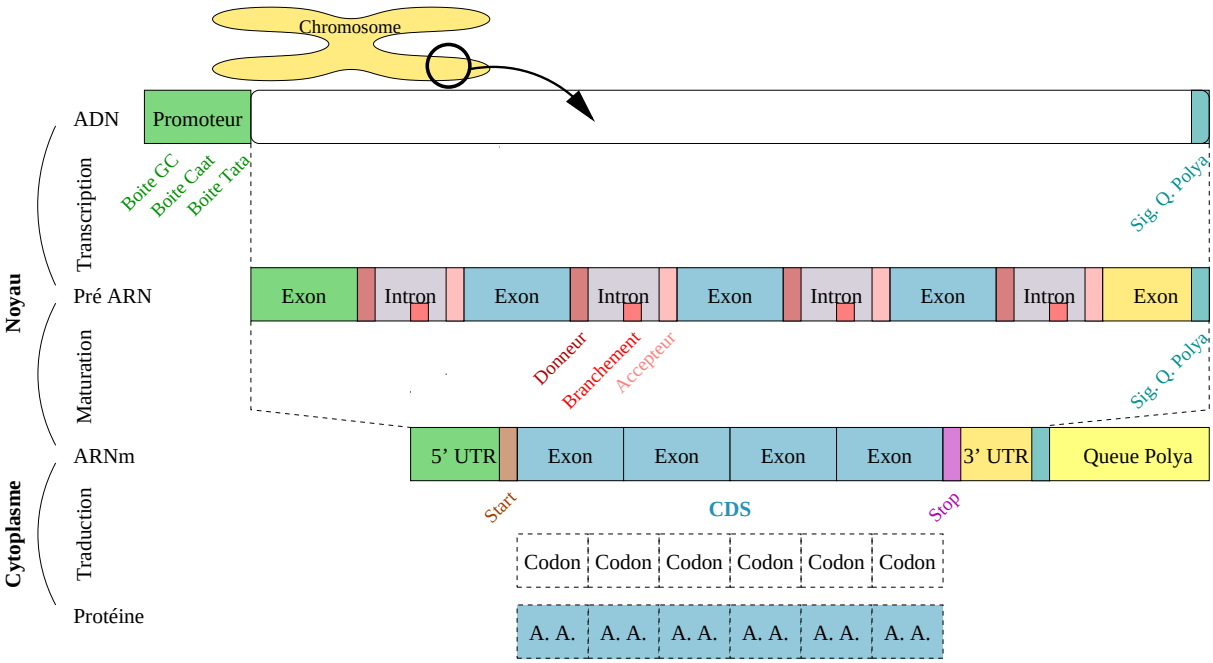

FIG. 1.1: Expression d'un gène et signaux correspondants

Chapitre 2

Un langage dédié à l'annotation :TAGCC

Nous nous intéressons ici au langage TAGCC dédié à l'annotation par des méthodes *ab-initio*. Dans un premier temps nous étudierons l'architecture de TAGCC avant d'en venir au langage en lui-même et de finir par les mécanismes permettant l'annotation d'une séquence.

Tout au long de ce chapitre, nous utiliserons la recherche de CDS afin d'illustrer nos propos. Pour rappel, un CDS commence par le signal ATG, codon `START` et se termine par TAA, TGA ou TAG, codon `STOP`. Entre ces deux signaux se trouve un certain nombre de codons, c'est-à-dire un nombre de nucléoides qui doit être multiple de trois.

2.1 Présentation

2.1.1 Généralités

TAGCC, acronyme de Transducteur pour l'Annotation des Génomes par des méthodes fondées sur des Contraintes et des Coûts, est un langage spécialisé au domaine de l'annotation dont l'objet est de décrire des structures de gènes comme se décrivent les structures grammaticales des langages ([Delaplace, 2002a], [Delaplace, 2002b]). Développé par Franck Delaplace en *OCaml*, TAGCC offre un cadre de programmation pour la résolution de la prédiction des gènes. Le mécanisme interne permettant de combiner reconnaissance et calcul se base sur l'emploi d'automates pondérés dont l'efficacité a été démontrée dans les domaines de la linguistique. Ce mécanisme permet tout en offrant une expressivité certaine une résolution très efficace.

L'originalité technique de ce langage concerne la définition d'un mécanisme unique permettant d'annoter la séquence des régions fonctionnelles d'intérêts, de gérer les solutions alternatives et enfin de prendre en compte les conditions.

Ce mécanisme se fonde sur la notion de marque avec contrainte. Une marque sera reconnue uniquement si la condition associée est satisfaite. Dans ce cas la séquence se verra marquée à la position où cette reconnaissance s'applique. Ce marquage offre un procédé pour *souligner* les zones présentant un intérêt fonctionnel.

Les solutions alternatives sont trouvées en combinant différemment les marques. Bien qu'une marque puisse être reconnue, le programme considérera à chaque fois que cela est possible la transition conduisant au marquage et celle conduisant à la lecture classique d'un caractère.

2.1.2 Architecture logicielle

Du point de vue de l'architecture, TAGCC est un langage permettant de réaliser des programmes d'annotation. On distingue deux étapes principales dans cette réalisation :

1. la compilation du programme écrit en TAGCC,
2. l'exécution de ce programme.

Compilation du programme

Un programme d'annotation est un fichier d'extension `.tag`, écrit en langage TAGCC (voir la section 2.2). TAGCC permet de compiler ce fichier pour créer un fichier *OCaml*, qui est lui même compilé par ocamlopt, le compilateur de *OCaml*.

Plus précisément, en considérant le fichier `cds.tag` qui décrit la recherche de CDS, TAGCC crée trois fichiers :

- un fichier `.tagcc.tmp` lié au fonctionnement interne de TAGCC,
- un fichier `cds.ml` qui correspond à la source en langage *OCaml* du programme obtenu,
- un fichier exécutable `cds`, qui est le programme d'annotation à proprement parler.

La figure 2.1 résume l'ensemble de ce mécanisme.

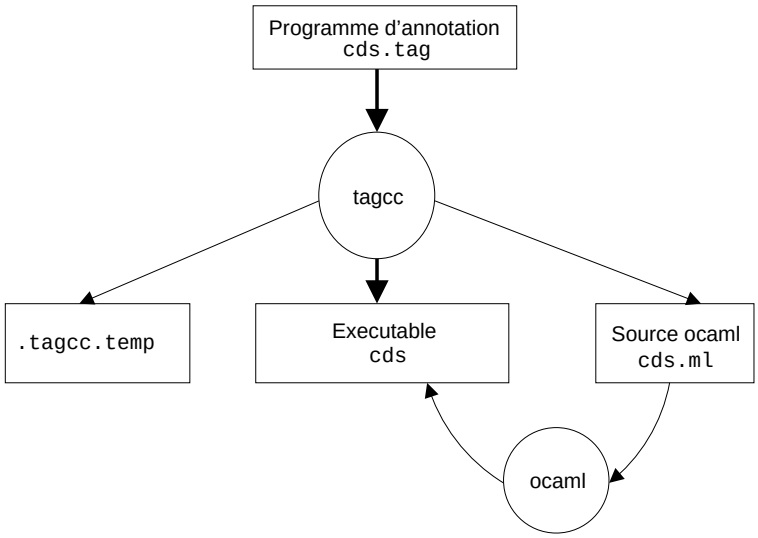

FIG. 2.1: Architecture logiciel de TAGCC : compilation

Exécution du programme

La deuxième étape consiste à exécuter le programme d'annotation obtenu par la compilation sur un fichier contenant la séquence à annoter. Ce fichier peut être de différent format (`.fasta`, `.dat`, etc.), le résultat obtenu sera une trace de l'exécution au format HTML. La trace d'annotation correspond à un découpage de la séquence en différents signaux dont la logique est définie dans le programme écrit en langage TAGCC.

La figure 2.2 résume cette étape.

2.2 le langage TAGCC

Cette section décrit le langage TAGCC au travers de l'exemple du programme `cds.tag`. Le langage TAGCC est un langage *déclaratif*. Un programme écrit en TAGCC correspond à la déclaration d'un *système d'équations*. Dans ce type de langage, on décrit uniquement les contraintes d'un problème que les solutions doivent respecter. En quelque sorte, les programmes des langages déclaratifs se réduisent à l'énoncé du problème, charge au système de résolution de

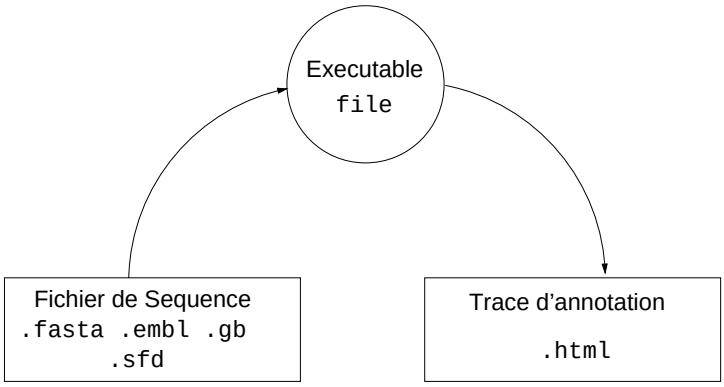

FIG. 2.2: Architecture logiciel de TAGCC : exécution

trouver une solution.

2.2.1 Recherche de CDS

Le programme de la figure 2.3 fournit un premier énoncé de la recherche des CDS.

```
// PROGRAMME DE RECHERCHE D'UN CDS
N =[A T G C] ;
Start = A T G ;
Stop  = (T A A) | (T A G) | (T G A) ;
cds = N* /{BEGIN_CDS} Start N+ Stop /{END_CDS} N* ;
cds ;
```

FIG. 2.3: Reconnaissance des CDS en TAGCC

Ce programme est constitué de cinq équations :
- `N =[A T G C] ;` définit N comme l'un des quatre nucléotides Adénine, Thymine, Guanine et Cytosine.
- `Start = A T G ;` attribue au signal `START` le codon ATG.
- `Stop  = (T A A) | (T A G) | (T G A) ;` attribue au signal `STOP` l'un des trois codons possibles TAA, TAG, TGA.
- `cds = N* /{BEGIN_CDS} Start N+ Stop /{END_CDS} N* ;` définit le CDS (une séquence de N nucléotides située entre un codon `START` et un codon `STOP`).
  `/{BEGIN_CDS}` et `/{END_CDS}` sont des marques qui s'insèrent dans la séquence pour en montrer la découpe dans le fichier résultat.
- La dernière ligne du programme (`cds ;`) indique au programme quel signal il doit utiliser pour l'annotation.

Cet exemple montre la simplicité et la concision des programmes écrit en langage TAGCC. Le nombre de lignes constituant le programme est réduit au minimum, c'est-à-dire à la modélisation du problème.

2.2.2 Syntaxe

Alphabet

L'alphabet de TAGCC correspond aux 4 nucléotides. Il s'agit des lettres A, T, G, C auxquelles s'ajoutent une cinquième, la lettre X. Cette dernière indique une erreur de séquençage.

Expressions Régulières

Les équations composant les programmes TAGCC sont définis par des *expressions régulières* suivant la syntaxe :

$$< signal > = < definition >;$$

Ces équations permettent de définir une équation principale qui sera déclarée à la fin du programme. C'est cette *équation principale* qui est utilisée pour l'annotation. Sa déclaration se fait suivant la syntaxe :

$$< equation\ principale >;$$

Il est possible de déclarer plusieurs équations principales dans un même programme.

Opérateurs

Différents opérateurs sont utilisés dans la définitions des expressions régulières. Ici sont présentés les plus fréquents.

- **L'opérateur de concaténation**
  Représenté par `[espace]`, il décrit la succession d'expressions régulières. Il permet de définir des signaux. Par exemple, la définition du codon `START` sera donnée par :
  `Start = A T G ;`
- **L'alternative**
  Elle permet de définir un choix pour la reconnaissance d'expressions régulières. Il existe deux notations dont les résultats sont identiques :
  - la notation ensembliste : $< signal >= [< expression_1 > \cdots < expression_n >]$,
  - la notation $< signal >=< expression_1 > | \cdots | < expression_n >$.

  Cela permet la déclaration de signaux possédant plusieurs définitions. Par exemple :
  `N = [A T C G] ;` ou encore `N = A|T|C|G ;`
- **Répétitions infinies**
  Deux notations sont possibles :
  - $< expression > *$ pour des répétitions allant de 0 à ∞,
  - $< expression > +$ pour les répétions allant de 1 à ∞.

  On utilise ces répétitions lorsque la longueur d'un signal n'est pas connue.

2.2.3 Calcul sur les séquences

La reconnaissance de motifs à l'aide des expressions régulières n'est pas suffisante à elle seule pour permettre l'annotation d'un génome. En effet, la définition des modèles pour l'annotation fait non seulement appel à des données qualitatives, mais aussi quantitatives.

Certaines zones à annoter sont en effet correctement définies non par une syntaxe mais par leur fonction. Ces zones sont alors représentées par une séquence consensus correspondant aux bases nucléotidiques les plus souvent observées. Néanmoins, chacune de ces bases présente une variabilité pouvant être déterminée sur le plan statistique (voir le chapitre I.1 pour les calculs par matrice de poids).

La reconnaissance de signaux par les programmes d'annotation écrits en TAGCC peut s'accompagner de calculs de valeurs. Ces calculs se font par l'intermédiaire de variables dont l'utilisation nécessite deux étapes :

1. **Déclaration de la variable**
   Le type des variables dans TAGCC correspond à un couple (domaine de définition, opération) On trouve ainsi par exemple, les entiers naturels, ou les réels munis de l'addition, ou encore les nucléotides munis de la concaténation.
   La déclaration des variables se fait classiquement sous la forme

   $$< type > \quad < variable_1 >, \cdots, < variable_n > ;$$

2. **Écriture du calcul fait à partir des variables**
   A chaque expression régulière peut être associée une *valeur élémentaire* pour chaque variable déclarée. Cette valeur correspond au poids de la variable pour cette expression. L'attribution du poids pour cette expression se fait par :

   $$< expr. > : \{< variable_1 >=< expr.num._1 > \cdots < variable_n >=< expr.num._n >\}$$

   Lors de la reconnaissance de l'expression régulière, un calcul s'effectue sur chaque variable déclarée. L'opération effectuée est celle définie par le type de la variable. Si une variable n'est pas associée à une expression, la reconnaissance de cette expression n'entraîne aucun calcul sur cette variable.

L'utilisation de variables permet de réaliser des programmes qui rendent une information plus riche. Ainsi, le programme `cds.tag` de la figure 2.3 peut être amélioré pour nous rendre la longueur du cds. Le résultat est présenté figure 2.4

```
nat len ;
N =[A T G C] ;
Start = A T G ;
Stop  = (T A A) | (T A G) | (T G A) ;
cds = N* /{BEGIN_CDS} Start (N:{len=1})+ Stop /{END_CDS} N* ;
cds ;
```

FIG. 2.4: Reconnaissance et calcul de la longueur des CDS en TAGCC

Deux lignes ont changé par rapport au programme de la figure 2.3 :
- `nat len ;` définit `len` comme une variable de type entier, l'opérateur d'addition lui est associé.
- `cds = N* /{BEGIN_CDS} Start (N:{len=1})+ Stop /{END_CDS} N* ;` ajoute 1 à la variable `len` à chaque fois qu'un nucléotide situé entre le codon `START` et le codon `STOP`) est lu.

2.2.4 Marquage simple et conditionnel

L'utilisation des marques dans les équations des programmes écrits en TAGCC permet d'identifier des points importants de la séquence. On distingue deux types de marquage : le marquage simple et le marquage conditionnel.

Marquage simple

On peut introduire une marque simple dans une équation en utilisant la syntaxe suivante :

$$/\ \{ < marque > \}$$

Ces marques vont être insérées à des points importants de la séquence définis par l'équation dans laquelle elles se trouvent. Par exemple, l'équation `Signal = N* T /M N*` appliquée à la séquence `AATATATA` va conduire aux marquages suivants, où `<M>` désigne la position d'insertion de la marque :

```
AAT <M> ATATA
AATAT <M> ATA
AATATAT <M> A
```

Le fichier résultat va contenir tous les marquages possibles.

Ces marques ont le même emploi que la ponctuation dans la langue française. Elles permettent d'interpréter les résultats d'une certaine manière et d'ôter toute ambiguïté. Par exemple, pour la recherche des CDS (voir le programme de la figure 2.4), si la marque `BEGIN_CDS` est posée, la séquence ATG suivant cette marque pourra être interprétée comme le début d'un gène potentiel. Par contre, si la marque n'est pas posée, le même ATG sera considéré comme faisant partie d'une région intergénique. Dans cet exemple, la marque permet de lever l'ambiguïté.

L'introduction de marques correspond non seulement à une mise en valeur de parties de la séquence à analyser mais aussi à une nécessité liée au calcul de *l'automate pondéré* constituant le coeur de la compilation (voir la section 2.3).

Marquage conditionnel

Le Marquage conditionnel permet d'imposer des conditions pour le choix de lecture ou de marquage. Ces conditions s'écrivent de la facon suivante :

$$/\ \{ < cond_1 > , \cdots , < cond_n > \ -> < marque > \}$$

Le marquage ne se fera que si toutes les conditions sont vérifiées.

Ainsi dans le cas de recherche de CDS, le nombre de nucléotides situés entre les codons `START` et `STOP` doit être un multiple de trois pour correspondre à un nombre entier de codons. La marque `END_CDS` ne peut être mise que sous cette condition. C'est ce que montre le programme de la figure 2.5.

```
nat len ;
N =[A T G C] ;
Start = A T G ;
Stop  = (T A A) | (T A G) | (T G A) ;
cds = N* /{BEGIN_CDS} Start (N:{len=1})+ Stop /{ (len % 3) == 0 -> END_CDS} N* ;
cds ;
```

FIG. 2.5: Reconnaissance des CDS dont la longueur est un multiple de trois en TAGCC

Actions associées

Des actions peuvent être associées aux marques. Il s'agit par exemple de remettre une variable à zéro dans le cas où l'on pose la marque, ou bien encore de ne pas lire une certaine partie de la séquence.

Fonction de coût

Il est intéressant de pouvoir donner un coût aux différentes solutions trouvées afin de pouvoir les comparer. Dans TAGCC, une fonction de coût peut être associée à chaque marque. Cela s'effectue par la commande

$$/\ \{ < marque > [< cout >] \}$$

TAGCC impose que la fonction de coût ne dépende que de la marque/position précédente.

Dans notre exemple, on peut ainsi vouloir rechercher les solutions avec le plus court CDS possible. Il suffit alors d'associer la longueur du CDS à la marque de fin `END_CSD`, ce qui s'obtient par :

```
cds = N*/{BEGIN_CDS} Start (N:{len=1})+ Stop/{(len%3)==0 -> END_CDS[len]} N* ;
```

Cet exemple n'est pas des plus pertinent, mais à l'avantage d'être simple.

2.3 Mécanismes

Dans cette section nous nous intéressons aux mécanismes de fonctionnement de TAGCC.

Lorsque TAGCC génère le fichier *OCaml*, il y définit différentes structures ; les deux principales sont un automate fini déterministe pondéré, et un ensemble de contraintes, conditions et formules de calcul à appliquer pour évaluer les marques.

Considérons le programme `cds.tag` de la figure 2.5. TAGCC génère un fichier `cds.ml` dont les deux principales structures sont représentées figures 2.6 et 2.8.

2.3.1 Automate

Les équation du programme `.tag`, dans notre exemple `cds.tag`, sont représentées en parties par une structure contenant, entre autres, la définition d'un automate fini déterministe pondéré. Cette structure pour `cds.tag` est décrite figure 2.6.

Les deux principaux champs sont

- `tagcc_cds.transducer` qui représente l'automate,
- `tagcc_cds.markdef` qui permet d'associer à un état de l'automate une marque (voir la section 2.3.2).

Dans notre exemple, le cœur du programme se situe dans la ligne

```
cds = N*/{BEGIN_CDS} Start (N:{len=1})+ Stop/{(len%3)=0 -> END_CDS} N* ;
```

L'automate correspondant à cette ligne ne prend pas en compte la condition, il est représenté sur la figure 2.7

Les états de l'automate sont réprésentés par des cercles, les flèches représentent les transitions. Lorsqu'une flèche est accompagnée d'un texte, cela peut signifier plusieurs choses :

- s'il s'agit d'une lettre A, T, G ou C, cela indique le nucléotide à lire dans la séquence traitée afin de franchir la transition,
- si un nombre accompagne ces lettres, cela indique de quelle quantitée doit être augmentée la variable représentant la longueur du CDS,
- enfin, le texte peut également représenté un nom de marque (`BEGIN_CDS` par exemple), dans ce cas, la transition n'est franchie que si la marque est posée sur la séquence.

```
let tagcc_cds =
{
  beg = 0;
  term = 10;
  transducer = Array.make 11 [||];
  lasttrans = [|Model.Null|];
  markdef = [|1;0;0;0;0;0;0;0;0;2;0|];
};;

tagcc_cds.transducer.(0)<-[|(0,Model.Null);
    (0,Model.Null);
    (0,Model.Null);
    (0,Model.Null);
    (-1,Model.Null);
    (1,Model.Null)|];;(* 0 *)

( ... )

tagcc_cds.transducer.(9)<-[|(5,Model.Data {nat=Model.Data [|4;|]; });
    (6,Model.Data {nat=Model.Data [|3;|]; });
    (5,Model.Data {nat=Model.Data [|4;|]; });
    (5,Model.Data {nat=Model.Data [|4;|]; });
    (-1,Model.Null);
    (10,Model.Null)|];;(* 9 *)

tagcc_cds.transducer.(10)<-[|(10,Model.Null);
     (10,Model.Null);
     (10,Model.Null);
     (10,Model.Null);
     (-1,Model.Null);
     (-1,Model.Null)|];;(* 10 *)
```

FIG. 2.6: Définition de la structure de l'automate pour la recherche de CDS par TAGCC

2.3.2 Famille

La structure `families` sert à représenter les contraintes et les marques. La figure 2.8 définit les familles de marques pour le programme `cds.tag`.

Pour chaque famille, plusieurs champs sont disponibles :

- `pconstr` modélise la contrainte à respecter pour poser la marque,
- `marker` est le nom de la marque
- `format` détermine le format de sortie des marques, il indique la manière dont seront écrites les marques dans le fichier résultat,
- `fcost` correspond au coût de la marque,
- `fmark` correspond aux actions associées à une marque.

Dans notre exemple, si l'on considère l'état 9 de l'automate (juste avant de mettre la marque `END_CDS`), le champ `tagcc_cds.markdef` nous indique qu'il faut considérer la famille 2. Autrement dit, la contrainte associée à l'état 9 de l'automate est

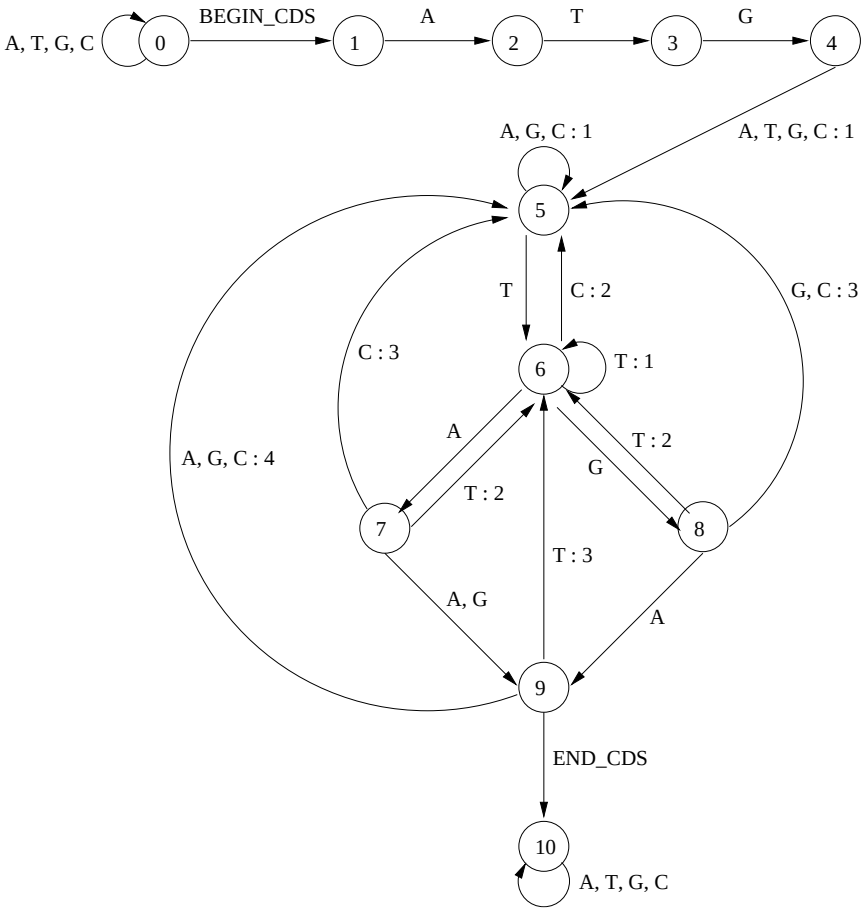

FIG. 2.7: Automate pour la recherche des CDS généré par TAGCC

```
pconstr=(fun r ->(Nat.get (Main.forget r).nat 0) mod 3=0);
```

La marque ne sera donc mise que si la taille du CDS est un multiple de 3.

2.3.3 Recherche de solutions

La recherche des solutions correspond à la recherche des sous-séquences de la séquence ADN à traiter, qui respectent la structure grammaticale définie par l'automate, et les contraintes modélisées par les familles.

Le principe de recherche est le suivant. le programme va lire la séquence ADN tout en avançant dans l'automate. À chaque lettre de la séquence est donc associé un état de l'automate. A chaque fois que le programme trouve un emplacement où il est possible de poser une marque, il doit considérer deux cas :

- soit la marque est posée,
- soit la marque sera posée plus tard, et dans ce cas la lettre est lue comme étant un nucléotide classique.

Considérons la recherche de CDS sur la séquence suivante

A T G C A T G C C C T A A C C T G A

L'automate permet de placer les marques suivantes

```
let families =
[|
  { pconstr=(fun r ->true);
    marker="!";
    format= "_";
    fcost=(fun r ->0.);
    fmark = (fun r ->());
  };
  { pconstr=(fun r ->true);
    marker="BEGIN_CDS";
    format= "";
    fcost=(fun r ->0.);
    fmark = (fun r ->());
  };
  { pconstr=(fun r ->(Nat.get (Main.forget r).nat 0) mod 3=0);
    marker="END_CDS";
    format= "";
    fcost=(fun r ->0.);
    fmark = (fun r ->());
  }
|];;
```

FIG. 2.8: Définition des familles pour la recherche de CDS par TAGCC

- `BEGIN_CDS` en position 0 et 4,
- `END_CDS` en position 12 et 17.

C'est-à-dire les quatre solutions suivante :

1. `<BEGIN_CS> A T G C A T G C C C T A A <END_CDS> C C T G A`
2. `<BEGIN_CS> A T G C A T G C C C T A A C C T G A <END_CDS>`
3. `A T G C <BEGIN_CS> A T G C C C T A A <END_CDS> C C T G A`
4. `A T G C <BEGIN_CS> A T G C C C T A A C C T G A <END_CDS>`

Le programme va alors relire ces solutions pour vérifier qu'elles respectent bien les contraintes. Dans notre cas, les familles de contraintes invalident les solutions 1 et 4 qui ne respectent pas que la longueur du CDS doit être un multiple de trois.

Deuxième partie

Recherche des K plus courts chemins

Chapitre 1

État de l'art sur la recherche des K plus courts chemins dans un graphe

Ce chapitre a pour but la présentation de l'état de l'art sur la recherche des K plus courts chemins.

Dans un premier temps nous nous intéresserons aux notations employées puis nous définirons de manière formelle le problème de recherche des K plus courts chemins avant de donner quelques applications de cette recherche. Nous en viendrons alors aux différentes familles de solutions. et étudierons les différentes classes de problèmes.

1.1 Notations et définitions

Nous voyons ici les notations employées par la suite, et définissons de manière formelle le problème de recherche des K plus courts chemins.

1.1.1 Graphe orienté

Soit $\mathcal{G}$ un graphe orienté. Alors $\mathcal{G} \equiv (\mathcal{N}, \mathcal{A})$ où
- $\mathcal{N} = \{v_1, ..., v_n\}$ est l'ensemble des sommets de $\mathcal{G}$,
- $\mathcal{A} = \{a_1, ..., a_m\} \subseteq \mathcal{N} \times \mathcal{N}$ est l'ensemble des arcs de $\mathcal{G}$; chaque arc $a_k \in \mathcal{A}$ s'écrit sous la forme d'une paire (i, j) de sommets.

Le problème des K plus courts chemins peut s'appliquer indifféremment sur un graphe G orienté ou non. Dans la suite du problème, on considérera que le graphe est orienté (le cas échéant, le graphe peut être orienté en remplaçant chacun des arcs (i, j) non orientés par deux arcs (i, j) et (j, i) orientés).

Tête et queue d'un arc, arc inverse

Soit $a_k \equiv (i, j) \in \mathcal{A}$, un arc de $\mathcal{G}$. Alors,
- les sommets i et j sont appelés *extrémités* de l'arc a_k
- le sommet i est appelé *queue* de a_k,
- le sommet j est appelé *tête* de a_k,
- l'arc (j, i) est appelé *arc inverse* de a_k et noté $inv(a_k)$.

Prédécesseurs et successeurs

On note par $w^-(j)$ (resp. $w^+(j)$) les *prédécesseurs* (resp. *successeurs*) du sommet $j \in \mathcal{N}$, c'est-à-dire $w^-(j) = \{i \in \mathcal{N} | (i, j) \in \mathcal{A}\}$ (resp. $w^+(j) = \{k \in \mathcal{N} | (j, k) \in \mathcal{A}\}$).

Les arcs $\{(i, j) \in \mathcal{A}\}$ (resp. $\{(j, k) \in \mathcal{A}\}$) sont dits arcs *entrants* (resp. *sortants*) de j.

1.1.2 Chaîne, chemin, circuit et cycle

Soit $n_1 \in \mathcal{N}$ et $n_2 \in \mathcal{N}$ deux sommets de $\mathcal{G}$.

Chaîne et chemin

Une *chaîne* de n_1 à n_2 dans $\mathcal{G}$ est une suite d'arcs $\{a_1, a_2, ..., a_l\}$ telle que
- $a_i \in \mathcal{A}$ ou $inv(a_i) \in \mathcal{A}$, pour tout $i \in [2, l-1]$,
- n_1 est la queue de a_1,
- n_2 est la tête de a_l.

Les *sommets d'une chaîne* sont l'ensemble des extrémités des arcs composant cette chaîne.

On appelle *longueur* d'une chaîne, le nombre d'arcs la composant.

Quand tous les sommets d'une chaîne sont différents (mais n_1 et n_2 peuvent coïncider) la chaîne est dite simple ou *élémentaire*.

Deux chaînes n'ayant aucun arc (resp. sommet) en commun sont dites *disjointes* par arcs (resp. par sommets). Deux chaîne disjointes par arcs sont disjointes par sommets ; la réciproque est fausse.

Si pour tout $i \in [1, l], a_i \in \mathcal{A}$, on parle alors de *chemin*.

On note $\mathcal{P}_{ij}$ l'ensemble des chemins entre deux sommets i et j.

Circuit et cycle

Une chaîne (resp. un chemin) élémentaire depuis un sommet i, jusqu'à ce même sommet i est appelée *circuit* (resp. *cycle*).

Un graphe ne contenant pas de cycle est dit *acyclique*.

Sous-chemin

Étant donnés deux sommets i et j d'un chemin p, le *sous-chemin* de p de i à j est simplement le chemin de i à j qui coïncide avec p entre ces deux sommets.

Sommet initial ou source, sommet terminal ou puit

Ces deux notions recouvrent des sommets particuliers d'un graphe.
- un sommet s d'un graphe $\mathcal{G}$, est dit *sommet initial* ou *source* de $\mathcal{G}$, s'il existe un chemin de s à n'importe quel autre sommet de $\mathcal{G}$, et s'il n'existe pas de chemin allant en s
- un sommet t d'un graphe $\mathcal{G}$, est dit *sommet terminal* ou *puit* de $\mathcal{G}$ s'il existe un chemin de n'importe quel sommet de $\mathcal{G}$ à t, et s'il n'existe pas de chemin partant de t.

Autrement dit $\mathcal{P}_{si} \neq \emptyset$ et $\mathcal{P}_{is} = \emptyset$, $\mathcal{P}_{it} \neq \emptyset$ et $\mathcal{P}_{ti} = \emptyset$ pour tout $i \in \mathcal{N}$. Si de tels sommets existent, ils sont uniques.

Dans la suite, on fera référence à $\mathcal{P}_{st}$, l'ensemble des chemins de s à t dans $\mathcal{G}$, par $\mathcal{P}$.

1.1.3 Ordre topologique

Dans un graphe acyclique, les arcs traduisent une relation d'ordre, partielle, entre les sommets appelée ordre topologique. Un *ordre topologique* T sur un graphe $\mathcal{G} \equiv (\mathcal{N}, \mathcal{A})$ orienté, est tel que si $(i, j) \in \mathcal{A}$ alors $T(i) < T(j)$ pour tout $i, j \in \mathcal{N}$.

L'ordre est dit partiel si on peut avoir $i, j \in \mathcal{N}$, tel que $T(i) = T(j)$.

L'ensemble des sommets d'un graphe ayant le même ordre topologique définit une *classe topologique*.

1.1.4 Arbre

Un *arbre* est un graphe sans circuit, et où il n'existe qu'une unique chaine élémentaire entre deux sommets.

Dans le cas où il existe un sommet initial s, on dit que l'arbre est *enraciné* en s.

Un *arbre de recouvrement* de $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ est un arbre dont l'ensemble des sommets est $\mathcal{N}$, et l'ensemble des arcs est un sous-ensemble de $\mathcal{A}$.

1.1.5 Distance

Distance d'un arc

À chaque arc $a_k \equiv (i,j) \in \mathcal{A}$ on associe un nombre réel d_{ij}, la *distance* (coût, temps, défiance, ...) de a_k ; on considère généralement que cette distance est un entier relatif, pour tous les arcs $(i,j) \in \mathcal{A}$, ce qui ne rend pas le problème plus spécifique.

Un graphe dont les arcs ont une distance est dit *valué*.

Distance d'un chemin

Soit p un chemin de s à t dans $\mathcal{G}$, $p \in \mathcal{P}$. On définit la fonction distance d de la manière suivante

$$\begin{array}{rcl} d : \mathcal{P} & \longrightarrow & \mathbb{Z} \\ p & \longmapsto & d(p) = \displaystyle\sum_{(i,j)\in p} d_{ij} \end{array}$$

$d(p)$ est dite distance du chemin p.

Cycle absorbant

Un cycle *absorbant* est un cycle de distance négative ou nulle. Par abus de langage, on confondra circuit absorbant et cycle absorbant.

1.2 Formulation du problème

Intéressons-nous à la définition du problème de recherche des K plus courts chemins. Rappelons tout d'abord la définition du problème de recherche du plus court chemin, puis nous généraliserons ce problème pour en venir à la recherche de K chemins.

Recherche du plus court chemin

Dans le problème, classique, de recherche du plus court chemin, l'objectif est de déterminer un chemin p^* depuis un sommet donné s jusqu'à un sommet donné t, dans un graphe donné $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, possédant n sommets et m arcs orientés. Le graphe est valué par une distance d. Le chemin cherché p^* est tel que que $d(p^*)$ est minimal sur $\mathcal{P}$, *i.e.* $d(p^*) \leq d(p)$, pour tout $p \in \mathcal{P}$.

Des algorithmes efficaces existent pour résoudre ce problème :

- pour un graphe quelconque, l'algorithme de Floyd-Warshall résout le problème en $\mathcal{O}(n^3)$;
- pour un graphe sans circuit absorbant, l'algorithme de Ford résout le problème en $\mathcal{O}(m.n)$;
- pour un graphe dont toutes les distances sont positives (d est à valeur dans $\mathbb{R}^+$ ou, par exemple, à valeur dans $\mathbb{N}$), l'algorithme de Dijkstra résout le problème en $\mathcal{O}(m+n.log(n))$;
- pour un graphe sans circuit, l'algorithme de Bellman résout le problème en $\mathcal{O}(m)$.

Le tableau de la figure 1.1 récapitule l'ensemble de ces solutions.

Recherche du plus court chemin

Cas général	Pas de circuit absorbant	Distances positives	Pas de circuit
Floyd-Warshall	Ford	Dijkstra	Bellman
$\mathcal{O}(n^3)$	$\mathcal{O}(m.n)$	$\mathcal{O}(m + n.log(n))$	$\mathcal{O}(m)$

FIG. 1.1: Tableau récapitulatif des complexités des algorithmes utilisés pour la recherche du plus court chemin dans un graphe à n sommets et m arcs

Recherche des K plus courts chemins

Étant donné un entier $K > 1$, le problème des K plus courts chemins peut être considéré comme une généralisation du problème précédent où, en plus d'avoir à déterminer le plus court chemin dans $\mathcal{P}$, il faut aussi déterminer le second plus court, ..., jusqu'au $K^{\text{ème}}$ plus court chemin dans $\mathcal{P}$. C'est à dire, en notant $p_i, i \in (N)$, le $i^{\text{ème}}$ plus court chemin de s à t dans $\mathcal{G}$, on recherche un ensemble de chemins $\mathcal{P}_K = \{p_1, ..., p_K\} \subseteq \mathcal{P}$, tel que

1. p_i est déterminé avant p_{i+1}, pour $i = 1, ..., K-1$;
2. $d(p_i) \leq d(p_{i+1})$, pour $i = 1, ..., K-1$;
3. $d(p_K) \leq d(p)$, quelque soit $p \in \mathcal{P} - \mathcal{P}_K$.

Ce problème, dans le cas général, est NP-Complet, c'est-à-dire qu'il n'existe pas d'algorithme polynomial pour le résoudre.

1.3 Applications

Les applications du calcul des K plus courts chemins sont nombreuses. Elles comprennent des situations telles que la recherche de chemin pour un robot, pour les autoroutes, pour la création de réseaux informatiques ([El-Amin and Al-Ghamdi, 1993]) ou dans des domaines tels que la génomique ([Byers and Waterman, 1984]). Des problèmes d'optimisation, comme par exemple le sac-à-dos quadratique ([Day et al., 1993]) ou les problèmes d'alignement de séquences ([Naor and Brutlag, 1994]), qui sont résolus par la programmation dynamique, ou par d'autres méthodes de recherche plus compliquées, sont autant de problèmes qui peuvent être exprimés sous forme de recherche de K plus courts chemins.

On trouve de nombreuses applications au problème des K plus courts chemins car il possède une grande puissance de modélisation.

Contraintes additionnelles.

On peut vouloir trouver un chemin qui satisfait un ensemble de contraintes, comme avoir une distance minimale, et dont certaines peuvent être mal définies, ou difficiles à optimiser. Une solution classique est de calculer plusieurs plus courts chemins, au sens des critères aisément modélisables, et d'en choisir un parmi ceux-ci en considérant les autres critères.

Évaluation d'un modèle.

Les chemins peuvent être utilisés pour modéliser des problèmes dont on connaît les solutions, indépendemment de la formulation du problème sous forme de chemins. Par exemple, dans le cas d'un modèle, sous forme de K plus courts chemins, pour la traduction automatique entre

deux langages, la traduction correcte peut être trouvée par un expert humain. En recherchant les chemins jusqu'à ce que la solution connue apparaisse, on peut déterminer avec quelle finesse le modèle permet de représenter le problème, en terme de nombre de chemins non corrects trouvés avant d'obtenir la bonne solution. Cette information peut être utilisée aussi bien pour améliorer le modèle que pour déterminer le nombre de chemins devant être générés lorsqu'on applique des contraintes additionnelles pour rechercher la bonne solution.

Analyse de la sensibilité.

En calculant plus d'un seul chemin, on peut déterminer la sensibilité de la solution optimale aux différentes variations des paramètres du problème.

Compréhension du problème.

Il peut être utile d'examiner non seulement la meilleure solution mais également un panel plus important de solutions afin de mieux comprendre le problème.

1.4 Familles de solutions

Nous nous intéressons ici aux différentes solutions pouvant être apportées au problème de recherche des K plus courts chemins.

On distingue en général deux familles de solutions :
- les solutions basées sur la déviation de chemins,
- les solutions basées sur le principe d'optimalité de Bellman.

1.4.1 Algorithmes de déviation

Les algorithmes de déviation de chemins (*deviation paths algorithms*) sont basés sur la construction d'un « *pseudo-arbre de recouvrement* » qui permet de calculer la *déviation* d'un chemin par rapport à un chemin référence. Pour être pratiquement utilisés, ces algorithmes nécessitent la construction d'un « *super-arbre* » qui permet de trouver les candidats au rang de $k^{\text{ème}}$ plus court chemin ([Eppstein, 1997], [Martins et al., 1999a]).

pseudo-arbre de recouvrement

Le pseudo-arbre de recouvrement se rapproche d'un arbre de recouvrement au sens classique du terme. Les sommets du pseudo-arbre de recouvrement sont les sommets et les arcs du graphe recouvert, mais, contrairement à un arbre de recouvrement classique, les sommets peuvent être répétés plusieurs fois. Cependant, un sommet n'est répété que s'il appartient à deux chemins différents.

L'exemple suivant permet de mieux comprendre le concept de pseudo-arbre de recouvrement ; considérons le graphe représenté sur la figure 1.2, où la distance entre deux sommets est indiquée sur l'arc reliant ces sommets.

On recherche les 3 plus courts chemins entre les sommet s et t de ce graphe. Ces chemins sont notés $q1$, $q2$ et $q3$. Le pseudo-arbre de recouvrement correspondant à cette situation est représenté sur la figure 1.3. Les sommets sont marqués par la distance de s au sommet considéré dans le chemin correspondant.

Déviation

Intuitivement, la déviation d'un chemin par rapport à un autre chemin est le sommet à partir duquel les deux chemins ne sont plus confondus.

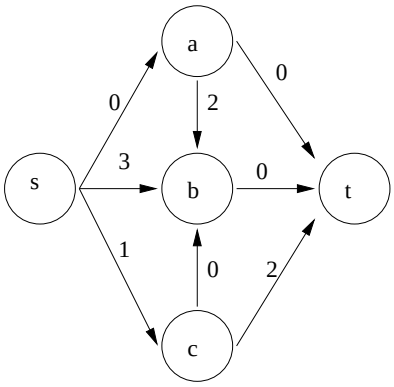

FIG. 1.2: Graphe servant pour le calcul du pseudo-arbre de recouvrement de la figure 1.3

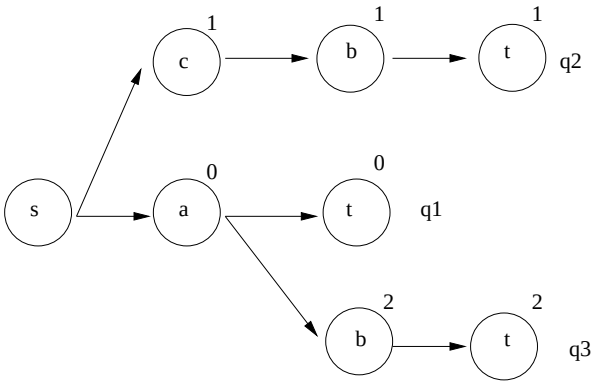

FIG. 1.3: Pseudo-arbre de recouvrement pour la recherche de 3 plus courts chemins dans le graphe de la figure 1.2

Formellement, la déviation d'un chemin est définie de la manière suivante : soit p_k le $k^{\text{ème}}$ plus court chemin d'un graphe $\mathcal{G}$, on appelle déviation du chemin p_k, le sommet y de p_k tel que le sous-chemin de p_k de s à y corresponde à un sous-chemin d'un des $(k-1)^{\text{ème}}$ plus courts chemins précédents. Dans l'exemple précédent, les trois chemins trouvés sont $q1 = \{s, a, t\}$, $q2 = \{s, c, b, t\}$ et $q3 = \{s, a, b, t\}$. La déviation de $q2$ est s, la déviation de $q3$ est a.

Super-arbre

Les algorithmes de déviation tentent de construire un pseudo-arbre de recouvrement qui contiennent les K plus courts chemins recherchés. La plupart du temps, ces algorithmes construisent un super-arbre, c'est à dire un arbre dont les sommets représentent un chemin du pseudo-arbre de recouvremnt. Pour déterminer le pseudo-arbre de recouvrement, les algorithmes de déviation utilisent un ensemble X des chemins candidats à être le prochain plus court chemin et recherchent parmi ceux-ci lequel est le meilleur. Une fois celui-ci trouvé, l'ensemble X est mis-à jour. Seule la partie des chemins comprise entre la déviation du chemin et le sommet terminal doit être recalculée.

La difficulté de cette méthode vient de la construction à proprement parler du super-arbre, ainsi que des traitements effectués pour déterminer l'ensemble X.

1.4.2 Principe d'optimalité de Bellman

La deuxième famille de solutions s'appuie sur le principe d'optimalité de Bellman qui, de manière générale, s'exprime : *« Toute sous-politique d'une politique optimale est une politique optimale »*. Considérant le problème de recherche du plus court chemin, le principe d'optimalité s'énonce : *« Tout sous-chemin, de s à y, du plus court chemin de s à t, est le plus court chemin de s à y »*.

Cet énoncé peut être généralisé au problème des K plus courts chemins. Il devient alors : *« Tout sous-chemin, de s à y, de p_k le $k^{ème}$ plus court chemin de s à t, est un $j^{ème}$ plus court chemin de s à y, $1 \leq j \leq k$ »*. Il est prouvé que le problème des K plus courts chemins satisfait au principe d'optimalité de Bellman *si et seulement si il n'y a pas de cycle absorbant* (*cf.* [de Azevado et al., 1990]).

Deux types d'algorithmes sont basés sur le principe d'optimalité de Bellman. Il s'agit des algorithmes de marquage de sommets (*labeling*) et de suppression de chemins (*deletion path*).

Marquage des sommets

Un algorithme type de marquage de sommet est l'algorithme de Bellman de recherche du plus court chemin dans le cadre d'un graphe acylique. À chaque sommet j est associé une marque, un *label*, constitué d'un coût π_j, et d'un prédécesseur ξ_j. Le coût π_j correspond à la distance du plus court chemin depuis le sommet initial s jusqu'au sommet j ; le prédécesseur ξ_j est le sommet précédent j dans le plus court chemin de s à j. L'ensemble des sommets est marqué petit à petit, en les parcourant dans l'ordre topologique du graphe.

On peut alors d'appliquer cette méthode pour la recherche des K plus courts chemins, en marquant chaque sommet par les K chemins les plus courts ([Martins et al., 1999c]).

Suppression de chemin

Ces algorithmes sont basés sur la suppression successives des chemins du graphe. On considère que p_2, le second plus court chemin de s à t dans $\mathcal{G}$ est le plus court chemin de s à t dans le graphe $\mathcal{G}_\infty$, où $\mathcal{G}_\infty$ est équivalent à $\mathcal{G}$, excepté que p_1, le plus court chemin de s à t dans $\mathcal{G}$, n'existe pas $\mathcal{G}_\infty$.

En répétant cette procédure pour $k > 1$, le $k^{\text{ème}}$ plus court chemin de s à t dans $\mathcal{G}$, p_k, est le plus court chemin de s à t dans le graphe, du moment que les $(k-1)$ premiers plus courts chemins, $p_1, ..., p_{k-1}$, n'existent plus. Donc en notant $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{A}_1) \equiv \mathcal{G}$ le graphe de départ, et $\mathcal{G}_k = (\mathcal{N}_k, \mathcal{A}_k)$ le graphe obtenu à partir de $\mathcal{G}$, en supprimant les $k-1$ premiers plus court chemins, le $k^{\text{ème}}$ plus court chemin de s à t dans $\mathcal{G}$ est le plus court chemin de s à t dans $\mathcal{G}_k$.

Toute la difficulté de ces algorithmes revient à trouver comment invalider un, et un seul, chemin dans un graphe.

1.5 Classes de problèmes

On distingue deux classes de problèmes, les problèmes dits contraints et les problèmes non contraints. Bien entendu, la topologie du graphe influe sur les algorithmes que l'on peut appliquer sur ces problèmes.

Nous allons tout d'abord voir quelles sont les différentes topologies de graphe rencontrées, puis nous nous intéresserons aux problèmes non contraints pour finir par les problèmes contraints.

1.5.1 Topologie du graphe

On distingue, comme dans le cas du problème de recherche du plus court chemin, quatre classes de graphe :

- *Graphe quelconque*
  Dans le cas général, le graphe est quelconque, il peut admettre des cycles (absorbants ou non), les arcs peuvent être valués positivement ou négativement.
- *Graphe sans circuit absorbant*
  Le graphe sans circuit absorbant est un cas particulier du précédent. Les arcs peuvent être valués positivement ou négativement, mais les cycles sont tous de distance strictement positive.
- *Graphe valué positivement*
  Le graphe valué positivement est un cas particulier du graphe sans circuit absorbant. Dans ce graphe, tous les arcs ont une distance positive ou nulle. Les cycles sont tous de distance strictement positive.
- *Graphe sans circuit*
  Le graphe sans circuit est un cas particulier du graphe sans circuit absorbant. Les arcs ont une distance quelconque. Il n'y a aucun cycle.

1.5.2 Problèmes non contraints

Dans cette classe de problèmes, aucune restriction n'est imposée aux chemins. Tout chemin trouvé est valide. On parle alors de recherche de chemins généraux. Cette classe de problèmes peut servir lorsqu'on recherche le plus court chemin, mais que les contraintes imposées sont fortes, non modélisables directement, ou difficiles à modéliser. On recherche alors les plus courts chemins généraux, la solution aux problème est le chemin le plus court qui respecte les contraintes désirées.

- Dans le cas où le graphe est *quelconque* ce problème est NP-Complet.
- Dans le cadre d'un graphe *sans cycle absorbant*, Martins et dos Santos ont développé l'algorithme *MS* (*cf.* [Martins and dos Santos, 1999]), un algorithme de suppression de chemins, qui permet de résoudre le problème en $\mathcal{O}(K.m)$, une fois déterminé le plus court chemin de s, sommet initial, à chacun des autres sommets du graphe. Donc, dans le cas d'un graphe sans cyle absorbant, et en utilisant l'algorithme de Ford de recherche du plus court chemin, la solution est obtenue en $\mathcal{O}(K.m + n.m)$.
- Dans le cadre d'un graphe *valué positivement*, l'algorithme d'Eppstein est applicable ([Eppstein, 1994], [Eppstein, 1997]). Cet algorithme basé sur la déviation de chemins, ainsi que sur la représentation implicite de ces derniers, permet de résoudre le problème en $\mathcal{O}(K + m + n)$, une fois déterminé le plus court chemin de s, sommet initial, à chacun des autres sommets du graphe ; soit, dans le cas d'un graphe valué positivement en $\mathcal{O}(K + m + n.log(n))$. Comparativement, l'algorithme *MS* résout le problème en $\mathcal{O}(K.m + n.log(n))$.
- Enfin, dans le cas d'un graphe *sans circuit*, l'algorithme *MS* résout le problème en $\mathcal{O}(K.m)$; si le graphe est de plus valué positivement, l'algorithme d'Eppstein résout le problème en $\mathcal{O}(K + m + n)$.

1.5.3 Problèmes contraints

Dans cette classe de problèmes, des restrictions sont imposées aux chemins cherchés, c'est-à-dire que seuls certains des chemins existants sont considérés comme valides, et que seuls ces chemins sont recherchés. Les contraintes imposées font que ces problèmes sont plus difficiles à résoudre que les problèmes non contraints.

Deux restrictions sont particulièrement utilisées :

- recherche de chemins disjoints (par arcs ou par sommets),
- recherche de chemins élémentaires (*i.e.* sans circuit).

D'autres restrictions existent mais sont peu utilisées. Généralement, le problème reste NP-Complet dans ces cas là.

Chemins disjoints par arcs ou par sommets

Ce problème est très fréquent, en particulier dans les réseaux d'information (*cf.* [El-Amin and Al-Ghamdi, 1993]). Ainsi, on peut désirer modéliser un réseau par un graphe, où les sommets sont les postes d'information, et les arcs les lignes par lesquelles l'information transite. Afin de minimiser le coût de construction du réseau, on cherche à relier deux postes donnés, par le plus court chemin entre ceux-ci. Mais, un réseau d'information doit également être "robuste", au sens où si une ligne ne permet plus de transmettre l'information, cette dernière doit quand même parvenir à destination. Pour cela, il faut que les deux postes soient reliés non seulement par le plus court chemin, mais aussi par le second plus court chemin au cas où le premier deviendrait non valide. De plus, ces deux chemins ne doivent avoir aucune ligne, aucun arc en commun car si cette ligne devenait non valide, l'information ne pourrait plus passer par aucun des chemins. Les deux chemins doivent donc être disjoints par arcs.

D'autre part, si on se contente de chercher le plus court chemin, puis de supprimer du graphe les arcs le composant et enfin de chercher le nouveau plus court chemin, il se peut que le deuxième chemin trouvé ait une distance très grande. Il ne faut donc pas seulement minimiser la distance des chemins au fur et à mesure, mais bien de manière globale.

Plus formellement, ce problème revient à chercher K chemins disjoints dans un graphe, de telle manière que la somme des distances des chemins soit minimale.

Ce problème est référencé dans le compendium de Garey et Jonhson ([Garey and Johnson, 1979]) sous le sigle *ND40*.

- Dans le cas où le graphe est *quelconque* ce problème est NP-Complet.
- Dans le cas où le graphe est *sans circuit absorbant*, un algorithme a été développé par Gummadi Krishna ([Krishna, 2001]) qui permet de résoudre le problème en $\mathcal{O}(K.m.n)$ où m est le nombre d'arcs du graphe, n le nombre de sommets et K le nombre de chemins cherchés.

Chemins élémentaires

La recherche de K plus courts chemins élémentaires (*i.e.* sans circuit) dans un graphe ne respecte pas le principe d'optimalité de Bellman ([Martins et al., 1997]).

En effet, considérons le graphe de la figure 1.4 comme contre-exemple.

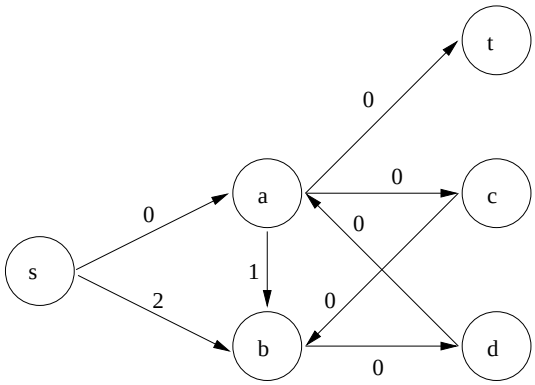

FIG. 1.4: Graphe ne respectant pas le principe d'optimalité de Bellman dans le cas de la recherche des K plus courts chemin élémentaires

On cherche les plus courts chemins de s à t qui soient sans circuit, il en existe deux :
- $p_1 = \{s, a, t\}$, de distance $d_{p_1} = 0$;
- $p_2 = \{s, b, d, a, t\}$, de distance $d_{p_2} = 2$.

D'après le principe d'optimalité de Bellman, pour $y \in p_2$, tout sous-chemin de p_2, de s à y, est un $j^{\text{ème}}$ plus court chemin de s à y, $j \leq 2$ car p_2 est le deuxième plus court chemin de s à t. Or, considérons les chemins de s à b :
- $q_1 = \{s, a, c, b\}$, de distance $d_{q_1} = 0$;
- $q_2 = \{s, a, b\}$, de distance $d_{q_2} = 1$;
- $q_3 = \{s, b\}$, de distance $d_{q_3} = 2$.

Il apparait que $\{s, b\}$ est le troisième plus court chemin de s à b, ce qui contredit le principe d'optimalité de Bellman.

Les solutions apportées au problème des K chemins élémentaires sont donc basées sur la déviation de chemins.

La majorité des algorithme dérivent d'algorithmes de recherche pour les chemins généraux, c'est-à-dire sans contraintes.
- Dans le cas où le graphe est *quelconque* ce problème est NP-Complet.
- Dans le cas où le graphe n'admet *pas de circuit absorbant*, Martins et Pascoal ont développé un algorithme permettant de résoudre le problème en $\mathcal{O}(K.n.(m + n.log(n)))$ ([Martins and Pascoal, 2000a]).
- Dans le cas où le graphe est *sans circuit*, tout chemin trouvé sera nécessairement élémentaire. Le problème de recherche de chemins généraux et celui de recherche de chemins élémentaires deviennent donc équivalents. Dans ce cas, le principe d'optimalité de Bellman est bien entendu respecté. Les algorithmes de recherche de chemins non contraints peuvent donc être utilisés, en particulier celui de Martins et dos Santos ([Martins and dos Santos, 1999]) qui résout le problème en $\mathcal{O}(K.m)$ et celui d'Eppstein ([Eppstein, 1997]) qui résout le problème en $\mathcal{O}(m + n + K)$ si le graphe est, de plus, valué positivement.

Le tableau de la figure 1.5 résume ces résultats.

Recherche des K plus courts chemins

<table>
<tr><th></th><th>Problèmes
non contraints</th><th colspan="2">Problèmes contraints</th></tr>
<tr><th></th><th></th><th>Chemins élémentaires</th><th>Chemins disjoints</th></tr>
<tr><td>Cas
général</td><td colspan="3">NP-Complet
(1)</td></tr>
<tr><td>Pas de circuit
absorbant</td><td>$\mathcal{O}(K.m + m.n)$
(2)</td><td rowspan="2">$\mathcal{O}(K.n.(m + n.log(n)))$
(4)</td><td rowspan="3">$\mathcal{O}(K.m.n)$
(5)</td></tr>
<tr><td>Distances
positives</td><td>$\mathcal{O}(m + n.log(n) + K)$
(3)</td></tr>
<tr><td>Pas de
circuit</td><td colspan="2">$\mathcal{O}(K.m)$ (2)
$\mathcal{O}(m + n + K)$ (3, si distances positives)</td></tr>
</table>

(1) : [Garey and Johnson, 1979]
(2) : [Martins and dos Santos, 1999]
(3) : [Eppstein, 1997]
(4) : [Martins and Pascoal, 2000a]
(5) : [Krishna, 2001]

FIG. 1.5: Tableau récapitulatif des complexités des algorithmes utilisés pour la recherche des K plus courts chemins dans un graphe à n sommets et m arcs

Troisième partie

Spécificités du problème

Chapitre 1

Le graphe *Marques-Positions*

Nous nous intéressons dans ce chapitre au graphe *Marques-Positions* qui permet de représenter l'ensemble des solutions à l'annotation. Nous voyons dans un premier temps la structure de ce graphe, puis comment le construire à partir des données fournies par TAGCC. Ce chapitre se termine sur la recherche de solutions dans ce graphe.

1.1 Structure du graphe

Une solution à l'annotation est définie par un ensemble de marques, mises à certaines positions, sur la séquence à annoter. Certaines marques ont un coût, qui ne dépend que de la marque précédente.

Une solution pour modéliser ce problème est de le représenter sous la forme d'un graphe tel que les sommets représentent les marques aux différentes positions possibles, on parle alors de marque/position, et tel que les arcs représentent les transitions entre ces marques/positions. Étant donné que le coût des marques est lié à la marque/position elle-même ainsi qu'à la marque/position précédente, il semble naturel de mettre ce coût sur les arcs du graphe.

Un tel graphe est dit graphe *Marques-Positions*.

1.1.1 Exemple : la recherche des CDS

Nous revenons ici sur l'exemple détaillé dans le chapitre I.2.

Pour rappel, la recherche des CDS a été modélisée avec TAGCC par la structure

```
cds = N*/{BEGIN_CDS} Start (N:{len=1})+ Stop/{(len%3)=0 -> END_CDS} N* ;
```

La séquence sur laquelle la recherche est effectuée est

```
A T G C A T G C C C T A A C C T G A
```

L'automate permettant de placer les marques/positions `BEGIN_CDS` en position 0 et 4, et `END_CDS` en position 12 et 17, on obtient le graphe *Marques-Positions* de la figure 1.1, où la longueur de la séquence est représentée sur les arcs (cette fonction de coût n'est pas pertinente, elle permet de trouver le CDS le plus petit, mais permet d'illustrer notre propos).

On peut d'ores et déjà remarquer que le graphe *Marques-Positions* ne prend pas en compte les contraintes imposées aux solutions (ici que la longueur du CDS doit être un multiple de 3). Ce problème est traité plus en détails dans la section 1.3.

1.1.2 Topologie

Le graphe *Marques-Positions* est considéré comme étant "*complet par classe topologique*" ; c'est-à-dire que le graphe *Marques-Positions* est tel que, si on calcule un ordre topologique sur le

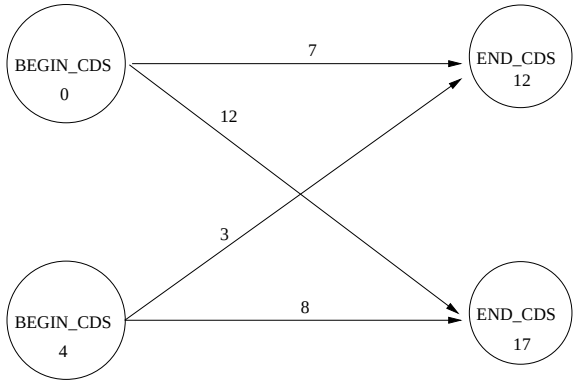

FIG. 1.1: Graphe *Marques-Positions* pour la recherche de CDS dans la séquence `ATGCATGCCCTAACCTGA`

graphe, et en considèrant deux classes topologiques C_1 et C_2, alors s'il existe un arc d'un sommet de C_1 vers un sommet de C_2, chaque sommet de C_1 est relié à chacun des sommets de C_2. Nous obtenons ainsi un graphe similaire à celui de la figure 1.2.

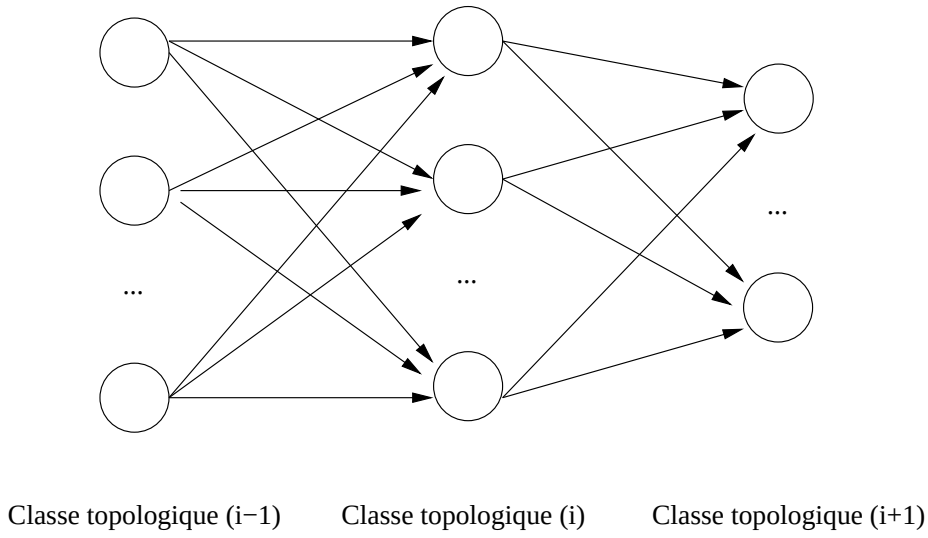

FIG. 1.2: Forme type d'un graphe *Marque-Position*

Schématiquement, chaque classe topologique correspond à l'ensemble des positions que peut prendre une marque.

Cette structure n'est pas une généralisation mais seulement un cas fréquent des graphes rencontrés. Certains graphes peuvent avoir des arcs reliant deux sommets appartenant à des classes topologiques non voisines, ou bien encore deux sommets de classes topologiques voisines peuvent ne pas être reliés entre eux. Mais les conséquences restent les mêmes.

1.1.3 Conséquences

Le graphe *Marques-Positions* est bien entendu un graphe orienté. De plus, du fait de sa construction, il est *acyclique*.

1.2 Construction du graphe

TAGCC, pour annoter une séquence, utilise un automate pondéré (voir le chapitre I.2). De cet automate, et à partir de la séquence à annoter, il est possible de construire le graphe *Marques-Positions*. En fait à plus proprement parler, il ne s'agit pas tant d'un graphe *Marques-Positions* que d'un graphe *États-Positions*. En effet, chaque sommet du graphe correspond non pas à une marque, mais à un état de l'automate.

L'idée de construction est la suivante.

Supposons qu'une marque $M1$ ait déjà été posée à la position $P1$ et qu'elle corresponde à l'état $E1$ de l'automate. Un sommet $S1$ correspondant au couple $(E1, P1)$ a été créé dans le graphe.

On recherche alors toutes les marques atteignables à partir de cette position $P1$ et de cet état $E1$ de l'automate. Pour ce faire, on parcourt la séquence à partir de la position $P1$, en partant de l'état $E1$. Lorsqu'on atteint une position $P2$ où la marque $M2$ peut être placée, deux choix sont possibles. Soit on place la marque, soit on continue de lire la séquence en considérant que la marque sera placée plus tard. Le but étant de trouver l'ensemble des positions atteignables, les deux solutions sont envisagées. De fait, on pose la marque $M2$, c'est-à-dire on crée dans le graphe *États-Positions* un sommet $S2 \equiv (E2, P2)$ ainsi que l'arc $(S1, S2)$ dont la distance ne dépend que de $S1$ et $S2$. On continue alors de lire la séquence à partir de $P2$, comme si l'on n'avait pas posé $M2$. La marque suivante trouvée, $M3$, sera traitée de la même manière, un sommet $S3$ sera créé, ainsi que l'arc $(S1, S3)$. On s'arrête lorsque toute la séquence a été lue.

Une fois toutes les marques atteignables à partir de $M1$ trouvées, on recommence en cherchant toutes les marques atteignables à partir de $M2$, puis de $M3$...

Lorsque l'ensemble des marques a été traité, il reste encore à vérifier que les chemins ainsi créés sont valides. En effet, le fait de poser les marques au dernier moment, peut empêcher de poser toutes les marques nécessaires à l'annotation. Par exemple, si l'annotation demande de poser les marques $M1$, $M2$ et $M3$ les unes après les autres, le fait de poser $M2$ à la toute fin de la séquence empêche de poser $M3$. De fait, les chemins ne permettant pas d'atteindre une marque terminale sont supprimés.

C'est dans le graphe obtenu que sont recherchés les K plus courts chemins.

1.3 Recherche de plus courts chemins dans le graphe

1.3.1 Sommets initial et terminal

Le logiciel TAGCC nous permet, finalement, d'obtenir après traitement d'une séquence d'ADN, un graphe *Marque-Position* acyclique et orienté (ou *DAG* pour Directed Acyclic Graph). Le graphe est également valué, chaque arc portant une distance représentant la défiance que l'on a dans l'annotation lui correspondant. Ce graphe a de plus la particularité d'être *complet par classe topologique*.

Le problème des K plus courts chemins tel que traité dans le chapitre II.1 recherche des chemins entre deux sommets donnés, un sommet initial et un sommet terminal. C'est pourquoi ces deux sommets particuliers sont ajoutés au graphe obtenu. Le sommet initial permet d'atteindre chacune des positions de départ possibles, toutes les positions de fin admissibles sont reliées au sommet terminal. À chaque fois, le coût de l'arc est nul.

Pour en revenir à l'exemple des CDS, on obtient le graphe de la figure 1.3

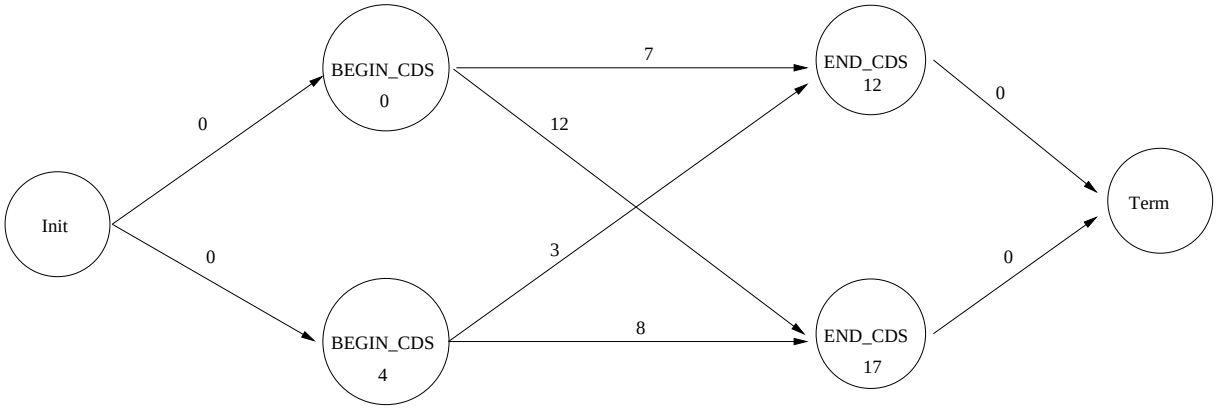

FIG. 1.3: Forme type d'un graphe *Marque-Position*

1.3.2 Contraintes

Les chemins recherchés sont soumis à de nombreuses contraintes, d'une part liées à la forme du chemin, d'autre part liées aux contraintes imposées par l'utilisateur.

Forme du chemin

Les chemins recherchés sont élémentaires, sans répétition de sommets. En effet du fait de l'enchaînement des marques/positions, il n'est pas possible de revenir sur une marque/position déjà posée (si on revient sur une marque, la position sera différente). On pourrait alors considérer, si on se réfère aux définitions du chapitre II.1, que nous sommes dans le cas d'un problème contraint, à savoir la recherche de chemins élémentaires. Or, puisque le graphe est acyclique, tout chemin trouvé sera nécessairement élémentaire.

On peut donc considérer que nous sommes dans un problèmes non contraint en ce qui concerne la forme des chemins cherchés.

Contraintes imposées par l'utilisateur

L'utilisateur est susceptible d'imposer des contraintes multiples et variées aux chemins afin d'être considérés comme valides. Par exemple, dans le cas de la recherche de CDS, la longueur de celui ci doit être un multiple de 3.

Ces contraintes ne sont pas toutes facilement modélisables. C'est pourquoi il est plus aisé de rechercher les chemins généraux (sans contraintes) dans un premier temps, et de faire un tri parmi ceux-ci afin de ne garder que ceux qui vérifient les contraintes. C'est la méthode qui a été employée et qui nous autorise à appliquer l'algorithme *MS* de Martins et dos Santos ([de Azevado et al., 1990], [Martins and dos Santos, 1999] et [Martins et al., 2000]).

De plus, l'algorithme *MS* est fait pour traiter les graphes avec cycles, notre graphe étant acyclique, la complexité de l'algorithme diminue par rapport à celle proposée par Martins et dos Santos. L'algorithme *MS* est détaillé dans le chapitre III.2.

Chapitre 2

Algorithme *MS* de recherche des K plus courts chemins

L'algorithme *MS* permet de rechercher les K plus courts chemins généraux dans un graphe ne contenant pas de circuit absorbant, et possédant un sommet initial et un sommet terminal, ce qui est le cas du graphe *Marque-Position*.

Après une présentation de l'algorithme, puis une analyse de celui-ci, nous étudierons sa complexité, et finirons par un exemple de recherche de K plus courts chemins, sur un graphe du type de ceux utilisés dans notre problème.

2.1 Présentation

L'algorithme *MS* a été développé à l'origine par Martins, puis amélioré par Martins, Pascoal, dos Santos, Azevedo, Madeira, et Pires ([de Azevado et al., 1990], [Martins and dos Santos, 1999], [Martins et al., 2000]).

Cet algorithme se fonde sur le principe d'optimalité de Bellman, et plus particulièrement sur le concept de suppression de chemins (voir le chapitre II.1).

Le principe de l'algorithme est le suivant.

1. Recherche du plus court chemin du graphe.
2. Construction d'un *graphe augmenté* à partir du graphe précédent et du chemin trouvé. C'est-à-dire, un graphe similaire au graphe de départ, mais où des sommets ont été rajoutés, et des arcs ont été rajoutés ou supprimés. En particulier, un nouveau sommet « terminal » est ajouté à chaque étape. Chaque sommet ajouté représente la meilleure alternative au plus court chemin pour aller du sommet initial au sommet considéré.
3. Recherche du plus court chemin depuis le sommet initial jusqu'au nouveau sommet terminal dans le graphe augmenté.
4. Si tous les chemins voulus ont été trouvés, alors on a fini sinon, retour en 2.

2.2 Analyse de l'algorithme

On recherche les K plus courts chemins du graphe $\mathcal{G}$, entre les sommets s et t (voir le chapitre II.1).

2.2.1 Arbre des plus courts chemins

La première étape de l'algorithme consiste à déterminer le plus court chemin de s à t. Cela est fait en déterminant un *arbre des plus courts chemins*, c'est à dire un arbre de recouvrement du

graphe $\mathcal{G}$, enraciné en s, où le plus court chemin de s à chacun des autres sommets est déterminé. Cet arbre, en plus de donner le plus court chemin entre s et t, permet dans la suite de l'algorithme de trouver rapidement le plus court chemin entre s et n'importe quel sommet du graphe.

Soit $p_1 = \{s \equiv n_0, n_1, ..., n_k, ..., n_{r-1}, n_r \equiv t\}$ le plus court chemin de s à t dans $\mathcal{G}$. L'objectif immédiat est de déterminer la meilleure alternative à p_1.

2.2.2 Alternatives

Pour tout sommet $x \in \mathcal{G}$, on note
- π_x la distance du plus court chemin de s à x,
- ξ_x le prédécesseur de x dans le plus court chemin de s à x,
- $w^-(x) = \{i \in \mathcal{N} | (i, x) \in \mathcal{A}\}$, l'ensemble des prédécesseurs de x.

On suppose qu'un algorithme de marquage a été utilisé pour trouver l'arbre des plus courts chemins, c'est-à-dire que pour chaque sommet x du graphe $\mathcal{G}$, on connait π_x et ξ_x.

Ce qui nous intéresse est de trouver p_2, la meilleure alternative au chemin p_1, et pour cela, nous allons déterminer quel est le prédécesseur de t dans p_2.

Deux solutions s'offrent à nous, soit p_2 passe par n_{r-1}, soit il n'y passe pas.
- Dans le cas où p_2 ne passe pas par n_{r-1}, cela signifie que p_2 est le plus court chemin de s à x concaténé à l'arc (x, t), où $x \in w^-(t) - \{n_{r-1}\}$. Autrement dit, le prédécesseur de t dans p_2 est le sommet qui minimise $\{\pi_x + d_{x,t} | x \in w^-(t) - \{n_{r-1}\}\}$.
- Dans le cas où p_2 passe par n_{r-1}, alors p_2 est composé de la meilleure alternative au sous-chemin de p_1 entre les sommets s et n_{r-1}, concaténée à l'arc (n_{r-1}, t).

Dans l'exemple de la figure 2.1, le plus court chemin entre s et t est représenté en pointillé, et passe par le sommet w. Le second plus court chemin entre s et t soit passe par w soit ne passe pas par w. S'il passe par w, alors il est le second plus court chemin de s à w plus l'arc de w à t ; s'il ne passe pas par w, alors il passe soit par u soit par v, en fait, par celui des deux qui minimise la distance de s à x (u ou v) plus la distance de x à t.

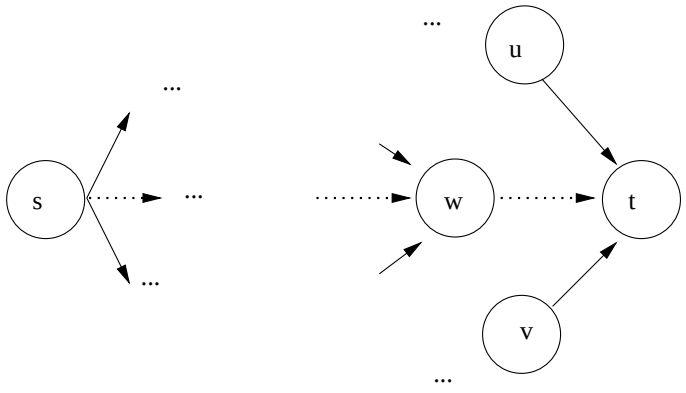

FIG. 2.1: Alternative au plus court chemin de s à t.

On peut conclure, que la détermination de la meilleure solution alternative à p_1 implique la détermination successive des meilleures alternatives à tous les sous-chemin de p_1, $\{n_0, n_1\}, ..., \{n_0, n_1, ..., n_{r-1}\}$ (dans l'exemple de la figure 2.1, pour déterminer l'alternative de s à t, il nous faut l'alternative de s à w).

Donc, une fois que p_1 est calculé, l'étape suivante est de trouver la meilleure alternative au plus court chemin depuis le sommet initial jusqu'à tous les autres sommet de p_1, suivant leur ordre d'apparition dans p_1. En fait l'algorithme commence avec le premier sommet de p_1 qui admet une alternative, c'est-à-dire le premier sommet de p_1 qui a plus d'un antécédent. En effet, la non existence d'une alternative à un chemin de s à un sommet n_k, $1 \leq k \leq r$, de p_1 implique la non existence d'alternatives aux chemins de s à n_j, $0 < j < k$ (s'il n'y a qu'une seule manière d'aller de A à C et qu'elle passe par B, alors il n'y a qu'une seule solution pour aller de A à B).

Soit $n_k, 1 \leq k \leq r$ le premier sommet de p_1 qui admet un chemin alternatif. On peut noter que, dans ce cas, n_k est le premier sommet de p_1 qui a plus d'un seul prédécesseur.

La meilleure alternative à $\{n_0, n_1, ..., n_k\}$ est associée à un nouveau sommet, noté n'_k. On considère que les prédécesseurs de n'_k sont les prédécesseurs de n_k, excepté n_{k-1}, le prédécesseur de n_k dans p_1. Autrement dit,

$$w^-(n'_k) = w^-(n_k) - \{n_{k-1}\}.$$

On conclut aisément en disant que

$$\pi_{n'_k} = min_{x \in w^-(n'_k)}\{\pi_x + d_{x,n'_k}\}$$

et $\xi_{n'_k}$ le sommet qui minimise la valeur précédente,

$$\xi_{n'_k} = argmin_{x \in w^-(n'_k)}\{\pi_x + d_{x,n'_k}\}.$$

Une fois que la meilleure alternative au sous-chemin $\{n_0, n_1, ..., n_k\}$ a été déterminée, la meilleure alternative à $\{n_0, n_1, ..., n_{k+1}\}$ peut être déterminée de la même manière. Il faut juste noter que

$$w^-(n'_j) = w^-(n_j) - \{n_{j-1}\} \cup \{n'_{j-1}\}, \text{ pour tout } n_j \in \{n_{k+1}, ..., n_r\}.$$

Il faut noter qu'avec cette méthode le second plus court chemin de s à t se trouve être le plus court chemin de s à t' dans le graphe augmenté. La correspondance entre les deux chemins est immédiate, il suffit de retirer les $'$ de tous les sommets de s à t'.

2.2.3 Généralisation

On note $x^{(\kappa)'}$ le sommet x avec κ symboles $'$, c'est-à-dire le sommet représentant la $\kappa^{\text{ème}}$ alternative au plus court chemin de s à x, *i.e.* le $(\kappa+1)^{\text{ème}}$ plus court chemin de s à x.

Soit $p = \{s \equiv \tilde{n}_0, \tilde{n}_1, ..., \tilde{n}_k, ..., \tilde{n}_i, ..., \tilde{n}_u \equiv t\}$ le plus court chemin de s à $t^{(\kappa-2)'}$ dans un graphe augmenté, et qui correspond au $(\kappa-1)^{\text{ème}}$ plus court chemin dans le graphe de départ. On suppose que p a été déterminé avec l'algorithme *MS*, et que $\tilde{n}_k$ est le premier sommet de p ayant plus d'un antécédent.

Alors la meilleure alternative à $\{s \equiv \tilde{n}_0, \tilde{n}_1, ..., \tilde{n}_k\}$ doit être déterminée et associée avec le nouveau sommet $\tilde{n}'_k$ (il faut bien voir que $\tilde{n}_k$ peut déjà posséder des $'$).

Mais si $\tilde{n}_k$ est un sommet faisant partie d'un chemin q précédemment calculé, alors la meilleure alternative à $\{s \equiv \tilde{n}_0, \tilde{n}_1, ..., \tilde{n}_k\}$ a déjà été calculée et associée au sommet $\tilde{n}'_k$. Il faut donc trouver le premier sommet de p dont le chemin alternatif n'a pas encore été calculé.

Soit $\tilde{n}'_i$ ce sommet. Deux cas sont à considérer soit $\tilde{n}_i$ est le sommet $\tilde{n}_k$, soit il ne l'est pas. Dans ce dernier cas, on note $\tilde{n}'_{i-1}$ le sommet associé à la meilleure alternative à $\{\tilde{n}_0, \tilde{n}_1, ..., \tilde{n}_{i-1}\}$; on a alors

$$w^-(\tilde{n}'_i) = w^-(\tilde{n}_i) - \{\tilde{n}_{i-1}\} \cup \{\tilde{n}'_{i-1}\}.$$

Quand $\tilde{n}_i$ coïncide avec $\tilde{n}_k$,

$$w^-(\tilde{n}'_i) = w^-(\tilde{n}_i) - \{\tilde{n}_{i-1}\}.$$

2.2.4 Conclusion

Avec cet algorithme, une séquence $\{t, t', (t')', ..., t^{(K-1)'}\}$ de K sommets est déterminée, chacun des éléments de la séquence représente un sommet "terminal" et une alternative au plus court chemin. De plus, le $\kappa^{\text{ème}}$ plus court chemin peut aisément être déterminé depuis $t^{(\kappa-1)'}$ en supprimant tous les $'$ des sommets du chemin calculé par une procédure de *backtrack* à partir de $t^{(\kappa-1)'}$ jusqu'à s.

2.3 Algorithme

L'algorithme présenté figure 2.2 est celui proposé par Martins et dos Santos dans [Martins and dos Santos, 1999]. On recherche les K plus courts chemins du graphe $\mathcal{G}$, entre les sommets s et t. On utilise les notations suivantes :

- $\overline{x}_k$ désigne le sommet x_k sans aucun $'$ et $x^{(0)'} \equiv \overline{x}$, pour tout $x \in \mathcal{G}$
- $\hat{x}_{k-1}$ (resp. $\hat{x}_{k+1}$) désigne la queue (resp. la tête) de l'arc du chemin p dont la tête (resp. la queue) est x_k,
- $w^-_{\mathcal{G}}(n)$ (resp. $w^-_{\mathcal{H}}(n)$) est l'ensemble des prédécesseurs de n dans $\mathcal{G}$ (resp. $\mathcal{H}$),

---

Algo *MS* : détermine les K plus courts chemins de $\mathcal{G}$
Début
$\mathcal{H} = \mathcal{G}$.
Déterminer $\mathcal{T}$, l'arbre des plus courts chemins dans $\mathcal{G}$.
Soit p_1 le plus court chemin de s à t dans $\mathcal{T}$.
$\kappa = 1$, $p = p_1$.
Tant Que (il existe une alternative à p) **et** $(\kappa < K)$ **Faire**
Déterminer n_k, le premier sommet de p tel que $w^-_{\mathcal{G}}(\overline{n}_k) > 1$.
Si $n'_k \notin \mathcal{H}$
Alors
Ajouter n'_k à $\mathcal{H}$.
Mettre à jour $w^-_{\mathcal{H}}(\overline{n}_k)$ tel que $w^-_{\mathcal{H}}(\overline{n}_k) = w^-_{\mathcal{H}}(\overline{n}_k) - \{\hat{n}_{k-1}\}$.
$\pi_{n'_k} = min\{\pi_x + d_{x,\overline{n}_k} | x \in w^-_{\mathcal{H}}(\overline{n}_k)\}$.
$\xi_{n'_k} = argmin\{\pi_x + d_{x,\overline{n}_k} | x \in w^-_{\mathcal{H}}(\overline{n}_k)\}$.
Soit $n_i = n_{k+1}$.
Sinon
Soit n_i le premier sommet de p suivant n_k tel que $n'_i \notin \mathcal{H}$.
Fin Si
Pour chaque $n_j \in \{n_i, ..., t^{(\kappa-1)'}\}$ **Faire**
Ajouter n'_j à $\mathcal{H}$.
Mettre à jour $w^-_{\mathcal{H}}(\overline{n}_j)$ tel que $w^-_{\mathcal{H}}(\overline{n}_j) = w^-_{\mathcal{H}}(\overline{n}_j) - \{\hat{n}_{j-1}\} \cup \{\hat{n}'_{j-1}\}$.
$\pi_{n'_j} = min\{\pi_x + d_{x,\overline{n}_j} | x \in w^-_{\mathcal{H}}(\overline{n}_j)\}$.
$\xi_{n'_j} = argmin\{\pi_x + d_{x,\overline{n}_j} | x \in w^-_{\mathcal{H}}(\overline{n}_j)\}$.
Fin Pour
Soit p le plus court chemin de s à $t^{(\kappa)'}$ dans $\mathcal{H}$.
$\kappa = \kappa + 1$.
Fin Tant Que
Fin

FIG. 2.2: Algorithme *MS* développé par Martins et dos Santos

---

2.4 Complexité

On considère un graphe à m arcs et n sommets. La première étape de l'algorithme, qui consiste à calculer l'arbre des plus courts chemins, peut être effectuée avec l'algorithme de Bellman, puisque le graphe est acyclique. La complexité de cette partie est donc en $O(m)$.

Ensuite, à chaque étape il est nécessaire de créer des sommets alternatifs pour chaque sommet de p_κ. La création de ces alternatives demande l'analyse de tous les arcs incident au sommet correspondant dans le graphe de départ. Dans le pire cas, tous les arcs du graphe doivent être analysés, et donc la complexité est $O(m)$ pour chacune des K étapes.

Finalement, dans le pire des cas, la complexité de l'algorithme est $O(K.m)$.

Dans le cas du graphe *Marques-Positions*, qui est complet par classe topologique, chaque sommet du chemin appartient à une classe topologique différente, et ses prédécesseurs recouvrent l'ensemble des sommets de la classe topologique précédente. On a donc au plus pour une étape de l'algorithme n arcs à étudier.

Dans le cadre du graphe *Marques-Positions*, la complexité est $O(K.n + m)$.

2.5 Exemple

Considérons la recherche des 4 plus courts chemin de s à t, dans le graphe $\mathcal{G}$ de la figure 2.3. Le graphe $\mathcal{H}$ est identique au graphe $\mathcal{G}$.

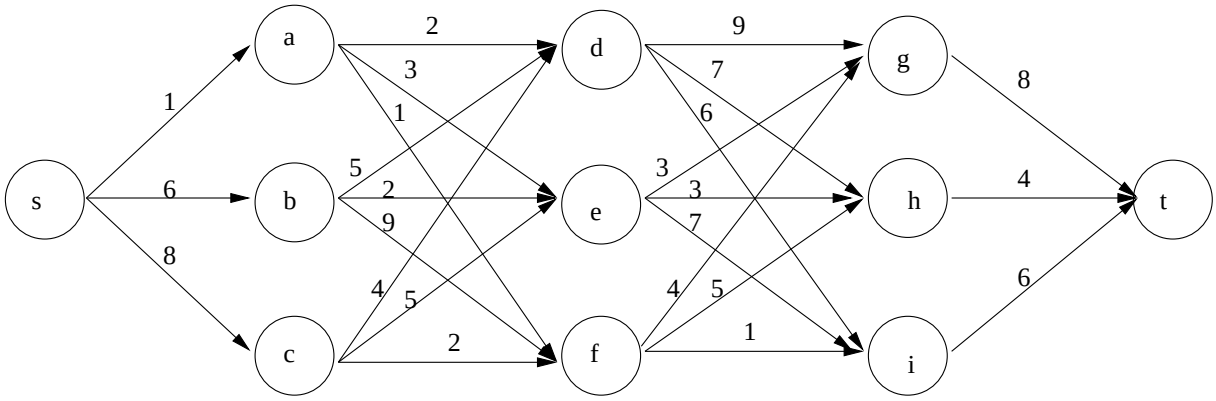

FIG. 2.3: Graphe $\mathcal{G}$ (et $\mathcal{H}$) pour la recherche des 4 plus courts chemins

Premier plus court chemin

La première chose à faire est de calculer l'arbre des plus courts chemins, représenté sur la figure 2.4. Sur les sommets est indiquée la distance du plus court chemin de s au sommet considéré.

Le premier plus court chemin est identifié par *backtracking* à partir de t. On trouve $p_1 = \{s, a, f, i, t\}$, de distance $d_{p_1} = 9$.

Second plus court chemin

Le premier sommet de p_1 à avoir plus d'un antécédent dans $\mathcal{G}$ est f. On crée donc, dans $\mathcal{H}$, un sommet f', et on met le graphe $\mathcal{H}$ à jour :

- $w^-_{\mathcal{H}}(f) = w^-_{\mathcal{H}}(f) - \{a\} = \{b, c\}$

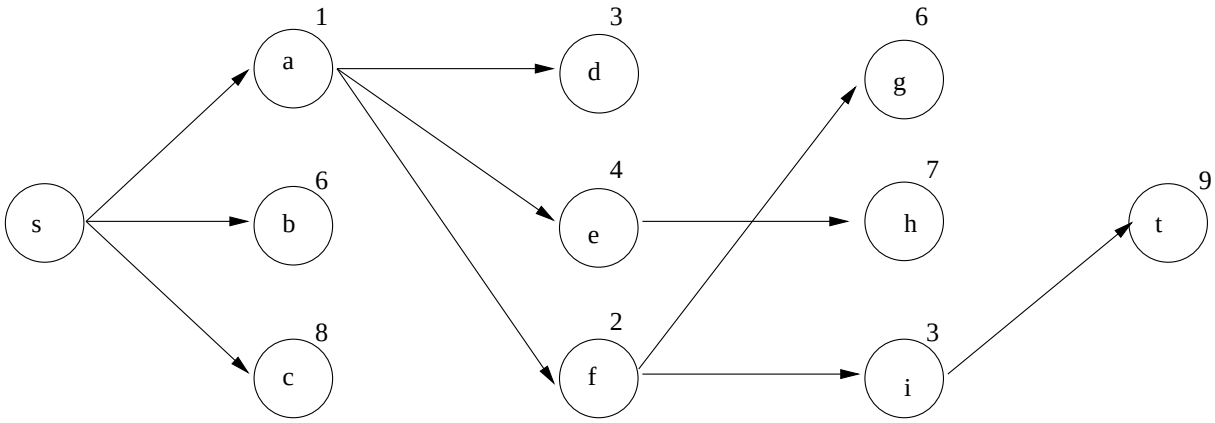

FIG. 2.4: Arbre des plus courts chemins correspondant au graphe $\mathcal{G}$ de la figure 2.3

- $\pi_{f'} = min\{\pi_x + d_{x,f} | x \in w^-_{\mathcal{H}}(f)\} = 10$
- $\xi_{f'} = argmin\{\pi_x + d_{x,f} | x \in w^-_{\mathcal{H}}(f)\} = c$

On fait la même chose pour le successeur i de f dans p_1 en créant i' et :
- $w^-_{\mathcal{H}}(i) = w^-_{\mathcal{H}}(i) - \{f\} \cup \{f'\} = \{d, e, f'\}$
- $\pi_{i'} = min\{\pi_x + d_{x,i} | x \in w^-_{\mathcal{H}}(i)\} = 9$
- $\xi_{i'} = argmin\{\pi_x + d_{x,i} | x \in w^-_{\mathcal{H}}(i)\} = d$

On fait la même chose pour le successeur t de i dans p_1 en créant t' et :
- $w^-_{\mathcal{H}}(t) = w^-_{\mathcal{H}}(t) - \{i\} \cup \{i'\} = \{g, h, i'\}$
- $\pi_{t'} = min\{\pi_x + d_{x,t} | x \in w^-_{\mathcal{H}}(t)\} = 11$
- $\xi_{t'} = argmin\{\pi_x + d_{x,t} | x \in w^-_{\mathcal{H}}(t)\} = h$

Le graphe $\mathcal{H}$ obtenu après cette première étape est représenté sur la figure 2.5.

Le second plus court chemin est obtenu par *backtracking* dans $\mathcal{H}$ à partir du sommet t' et en remontant jusqu'au premier sommet sans $'$. Le prédécesseur de t' est h, l'arbre des plus courts chemins de la figure 2.4 permet de trouver le plus court chemin de s à h. On trouve le chemin $p_2 = \{s, a, e, h, t'\}$.

Troisième plus court chemin

Le premier sommet de p_2 à avoir plus d'un antécédent dans $\mathcal{G}$est e. On crée donc un sommet e' dans $\mathcal{H}$, et on met le graphe $\mathcal{H}$ à jour :
- $w^-_{\mathcal{H}}(e) = w^-(e) - \{a\} = \{b, c\}$
- $\pi_{e'} = min\{\pi_x + d_{x,e} | x \in w^-_{\mathcal{H}}(e)\} = 8$
- $\xi_{e'} = argmin\{\pi_x + d_{x,e} | x \in w^-_{\mathcal{H}}(e)\} = b$

On fait la même chose pour le successeur h de e dans p_2 en créant h' et :
- $w^-_{\mathcal{H}}(h) = w^-(h) - \{e\} \cup \{e'\} = \{d, e', f\}$
- $\pi_{h'} = min\{\pi_x + d_{x,h} | x \in w^-_{\mathcal{H}}(h)\} = 7$
- $\xi_{h'} = argmin\{\pi_x + d_{x,h} | x \in w^-_{\mathcal{H}}(h)\} = f$

On fait la même chose pour le successeur t' de h dans p_2 en créant t'' et :
- $w^-_{\mathcal{H}}(t) = w^-(t) - \{h\} \cup \{h'\} = \{g, h', i'\}$

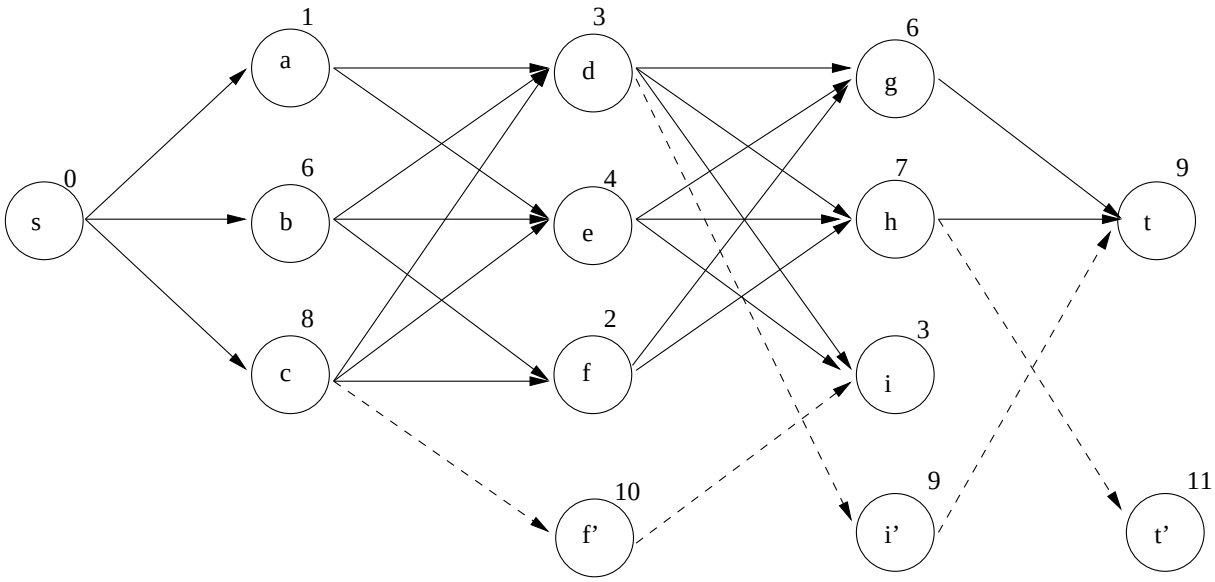

FIG. 2.5: Graphe $\mathcal{H}$ après le calcul du second plus court chemin

- $\pi_{t''} = min\{\pi_x + d_{x,t} | x \in w^-_{\mathcal{H}}(t)\} = 11$
- $\xi_{t''} = argmin\{\pi_x + d_{x,t} | x \in w^-_{\mathcal{H}}(t)\} = h'$

Le graphe $\mathcal{H}$ obtenu après cette étape est représenté sur la figure 2.6.

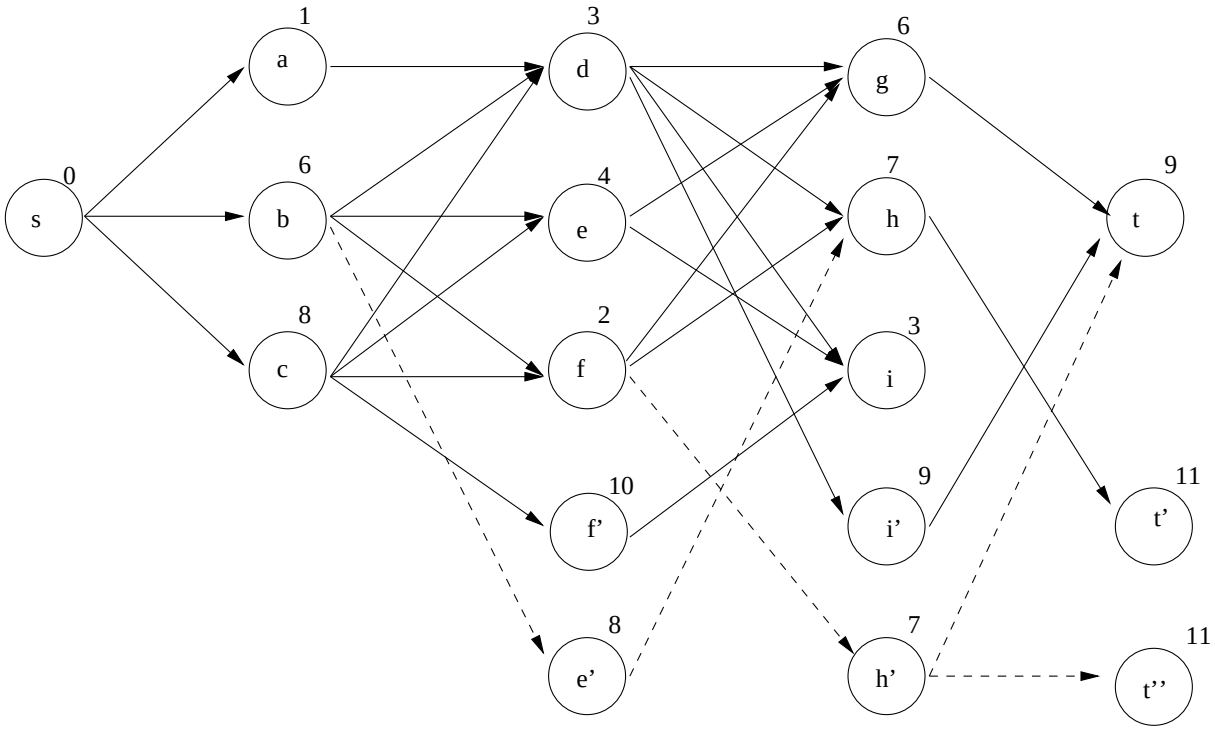

FIG. 2.6: Graphe $\mathcal{H}$ après le calcul du troisième plus court chemin

Le troisième plus court chemin est obtenu par *backtracking* dans $\mathcal{H}$ à partir du sommet t'' et en remontant jusqu'au premier sommet sans $'$. Le prédécesseur de t'' est h', le prédécesseur de h' est f, l'arbre des plus courts chemins de la figure 2.4 permet de trouver le plus court chemin de s à f. On trouve le chemin $p_3 = \{s, a, f, h', t''\}$.

Quatrième plus court chemin

Le premier sommet de p_3 à avoir plus d'un antécédent dans $\mathcal{G}$ est f, mais f' existe déjà dans $\mathcal{H}$. Le premier successeur x de f dans p_3 tel que x' n'existe pas dans $\mathcal{H}$ est h'. On crée donc un sommet h'', et on met le graphe à jour :

- $w^-_{\mathcal{H}}(h) = w^-(h) - \{f\} \cup \{f'\} = \{d, e', f'\}$
- $\pi_{h''} = min\{\pi_x + d_{x,h} | x \in w^-_{\mathcal{H}}(h)\} = 10$
- $\xi_{h''} = argmin\{\pi_x + d_{x,h} | x \in w^-_{\mathcal{H}}(h)\} = d$

On fait la même chose pour le successeur t'' de h' dans p_3 en créant t''' et :

- $w^-_{\mathcal{H}}(t) = w^-(t) - \{h'\} \cup \{h''\} = \{g, h'', i'\}$
- $\pi_{t''} = min\{\pi_x + d_{x,t} | x \in w^-_{\mathcal{H}}(t)\} = 14$
- $\xi_{t''} = argmin\{\pi_x + d_{x,t} | x \in w^-_{\mathcal{H}}(t)\} = h''$

Le graphe $\mathcal{H}$ obtenu après cette étape est représenté sur la figure 2.7.

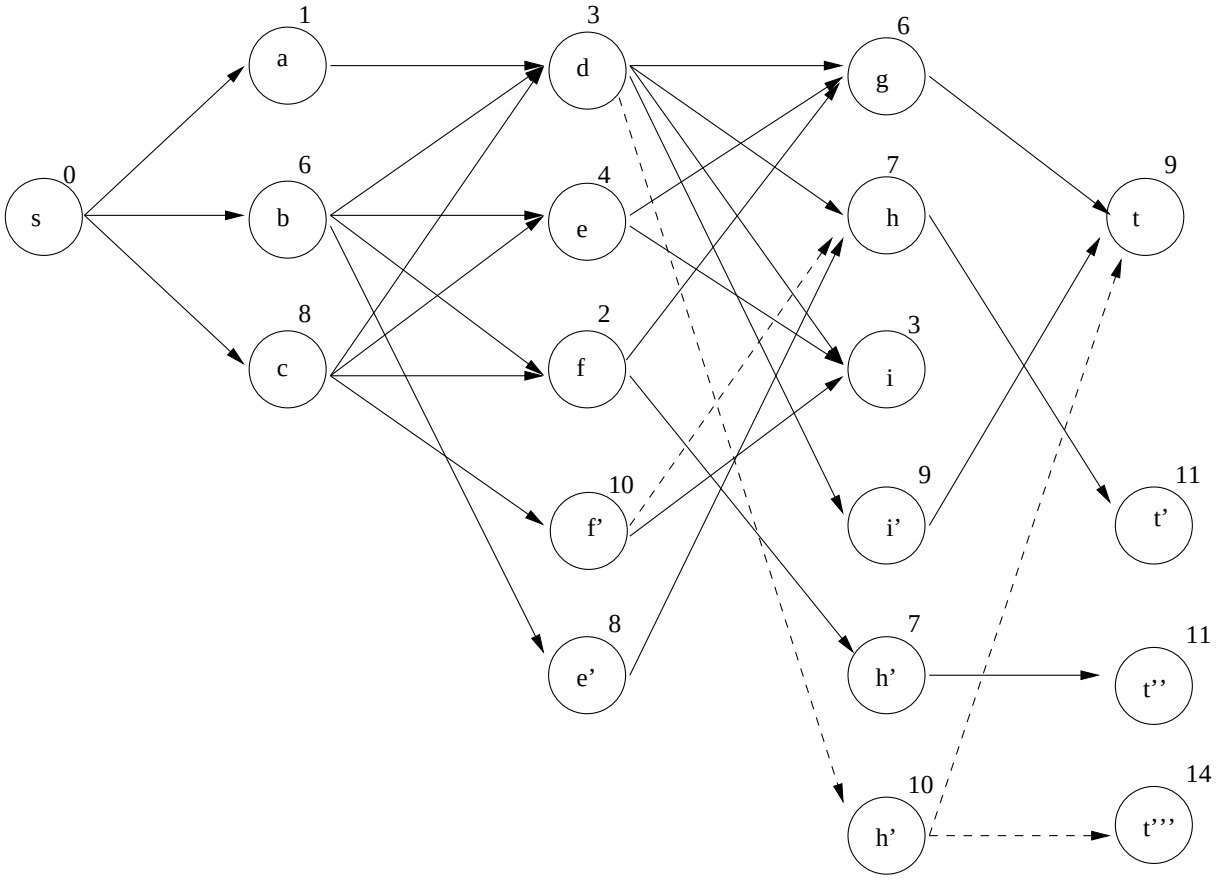

FIG. 2.7: Graphe $\mathcal{H}$ après le calcul du quatrième plus court chemin.

Le quatrième plus court chemin est obtenu par *backtracking* dans $\mathcal{H}$ à partir du sommet t''' et en remontant jusqu'au premier sommet sans $'$. Le prédécesseur de t''' est h'', le prédécesseur de h'' est d, l'arbre des plus courts chemins de la figure 2.4 permet de trouver le plus court chemin de s à d. On trouve le chemin $p_3 = \{s, a, d, h'', t'''\}$.

Conclusion

Finalement les quatre plus courts chemins sont
- $p_1 = \{s, a, f, i, t\}$, de distance 9,
- $p_2 = \{s, a, e, h, t\}$, de distance 11,
- $p_3 = \{s, a, f, h, t\}$, de distance 11,
- $p_4 = \{s, a, d, h, t\}$, de distance 14.

Quatrième partie

Tests et Validation

Chapitre 1

Graphes générés automatiquement

Afin de tester l'algorithme de recherche des K plus courts chemins, des graphes complets par classe topologique ont été générés.

Le programme de génération de graphe, présenté dans l'annexe C, prend en argument le nombre de sommets souhaités, et le nombre maximum de sommets par classe topologique.

Les résultats obtenus sont présentés dans le tableau de la figure 1.1.

n	m	K	t (s)
100	1000	10	0.02
		100	0.06
		1000	0.45
	2000	10	0.03
		100	0.06
		1000	0.47
1000	50000	10	0.70
		100	0.92
		1000	3.50
	200000	10	2.91
		100	3.08
		1000	6.80
10000	250000	10	4.02
		100	7.47
	500000	10	7.43
		100	10.03
	2500000	10	35.59
		100	38.21
	5000000	10	73.74
		100	77.11

FIG. 1.1: Résultats sur la recherche de K plus courts chemins dans un graphe complet par classe topologique à n sommets et m arcs

On peut remarquer que, lorsque le nombre de sommets est peu important, et donc le nombre d'arcs faible, le temps d'exécution est fortement lié au nombre de chemins cherchés. En revanche, lorsque le nombre d'arcs devient grand, le nombre de chemins cherchés n'a que peu d'influence sur le temps d'exécution.

Cela est lié à la complexité de l'algorithme MS qui est en $O(K.n + m)$. Le temps d'exécution

est donc directement proportionnel à $K.n + m$ comme le montre la figure 1.2.

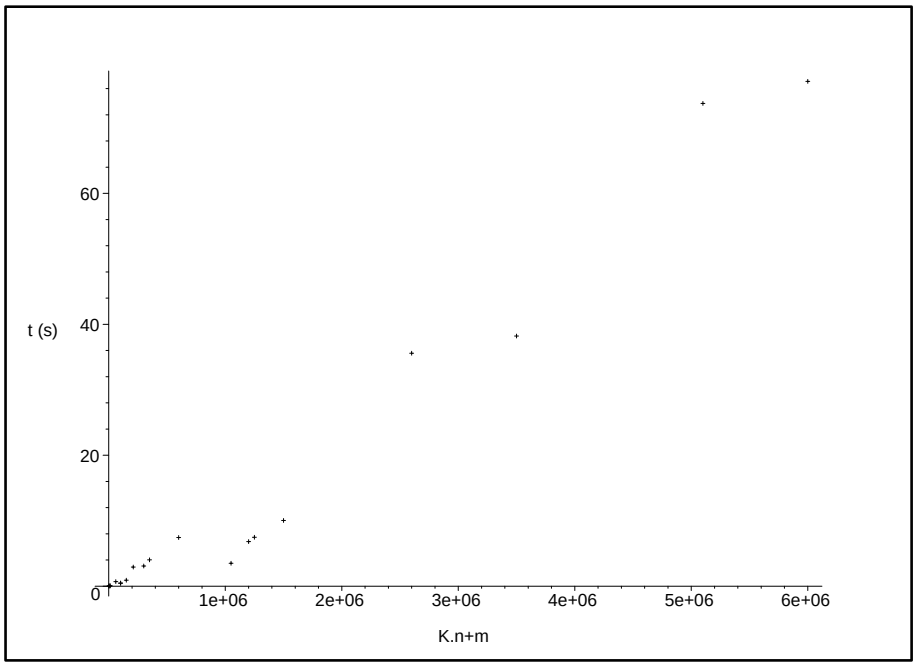

FIG. 1.2: Temps d'exécution de la recherche de K plus courts chemins dans un graphe complet par classe topologique de n sommets et m arcs en fonction de la complexité de l'algorithme MS ($O(K.n + m)$)

Plus pratiquement, on considère que l'annotation des séquences demande la pose d'une dizaine de marques, sur une centaine de positions, soit de l'ordre de 1000 sommets, 10000 si on considère les répétitions possibles d'une marque (dans la recherche d'exons par exemple). Le nombre de sommets par classe topologique est d'une centaine, soit environ 1000000 arcs. Le nombre de chemins recherchés est lui de l'ordre de 100, ce qui donne un temps d'exécution de quelques dizaines de secondes.

Chapitre 2

Recherche de CDS

La recherche de CDS a été présentée dans le chapitre I.2 afin d'illustrer le langage TAGCC. Pour mémoire le programme correspondant à cette recherche est rappelé dans la figure 2.1

```
nat len ;
N =[A T G C] ;
Start = A T G ;
Stop  = (T A A) | (T A G) | (T G A) ;
cds = N*/{BEGIN_CDS} Start (N:{len=1})+ Stop/{(len % 3)==0 -> END_CDS[len]} N* ;
cds ;
```

FIG. 2.1: Recherche de CDS en TAGCC

Ce programme permet de rechercher les CDS les plus courts.

De tests ont été réalisés sur une section du chromosome 13 d'*Homo Sapiens* composé de 3740 nucléotides.

La section possède 76 position possible pour le codon `start` (ATG), et 208 positions possibles pour le codon `STOP` (86 pour TAA, 77 pour TGA et 45 pour TAG).

Le tableau de la figure 2.2 résume les résultats obtenus.

K	t (secondes)	Solutions trouvées
10	21.71	0
100	22.15	34
1000	28.9	344

FIG. 2.2: Résultats de la recherche de CDS

La première chose à remarquer est que le temps d'exécution est supérieur à ce qui avait été prévu dans le chapitre précédent. Ceci provient en fait du temps nécessaire à la création du graphe *Marques-Position*, et en particulier à la phase d'épuration, qui consiste à supprimer les solutions non valides.

D'autre part, on peut voir que le problème de recherche de CDS, sans prendre en compte la contrainte d'une longueur multiple de 3, admet de nombreuses solutions. En effet, sur les 1000 meilleures solutions recherchées, seulement 344 respectent la contrainte. Il est donc nécessaire de rechercher plus de chemins que le nombre désiré de solutions.

Conclusion et perspectives

L'annotation *ab-initio* des génomes, et la recherche de gènes le long d'une molécule d'ADN sont des problèmes complexes, qui peuvent se ramener à la recherche de K plus courts chemins dans un graphe acyclique orienté.

TAGCC est un langage permettant l'annotation et la recherche de gènes par des méthodes *ab-initio*. Ces méthodes découvrent des gènes par des critères statistiques et l'étude syntaxique de l'ADN afin de distinguer les régions codantes des régions non-codantes.

L'objectif de ce stage était de proposer des heuristiques de recherche des K plus courts chemins dans un graphe acyclique orienté, tout en prenant en compte les particularités observées pour les programmes d'annotation en TAGCC.

Le problème d'annotation, NP-Complet, a été relaché afin d'arriver à un problème résolvable en temps polynomial. La forme particulière du graphe de recherche, le graphe *Marques-Positions*, à savoir acyclique, orienté et complet par classe topologique, ainsi que l'utilisation de l'algorithme *MS*, développé par Martins et dos Santos, ont permis de résoudre la recherche de K plus courts chemins dans le graphe *Marques-Positions* en un temps $O(K.n + m)$ où n est le nombre de sommets du graphe, et m le nombre d'arcs.

Une librairie de graphe en *OCaml* a été réalisée afin de traiter les graphes en général. D'un millier de lignes, elle permet en particulier le tri topologique des sommets d'un graphe, ainsi que la recherche du plus court chemin par l'algorithme de Bellman, ou encore le calcul du nombre de chemins passant par un arc, et bien sûr la recherche des K plus courts chemins (voir les annexes A et B).

Le logiciel TAGCC a également été modifié afin de prendre en compte une nouvelle option de compilation permettant de créer le graphe *Marques-Positions* et d'autoriser la recherche de solutions respectant les contraintes relachées dans l'espace des K meilleures solutions générales.

Une optimisation du code a également permis de ramener le temps de compilation des fichiers `.tag` les plus importants de plusieurs dizaines de minutes à quelques minutes seulement.

De fait, ces modifications ont finalement permis d'utiliser le logiciel TAGCC sur quelques données biologiques, pour la recherche de séquences codantes (CDS) en particulier.

Une amélioration possible du langage TAGCC serait liée à la création du graphe *Marques-Positions*. En effet, actuellement toutes les marques/positions possibles sont créées, et ensuite celles qui conduisent à des chemins non valides sont supprimées. Certaines de ces marques/positions pourraient dès leur lecture être considérées comme non valides, et donc ne pas être créées.

Enfin, dans le but d'améliorer l'algorithme de recherche de solutions, il pourrait être intéressant de trouver un critère permettant de déterminer un K optimal, à savoir combien de chemins chercher pour que le dernier trouvé ait encore un sens, du point de vue biologique.

Bibliographie

[Byers and Waterman, 1984] Byers, T. H. and Waterman, M. S. (1984). Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Operation Research*, 32 :1381–1384.

[Chuang and Roth, 2001] Chuang, J. and Roth, D. (2001). Gene recognition based on DAG shortest paths. *Bioinformatics*.

[Crescenzi and Kann, 1998] Crescenzi, P. and Kann, V. (1998). A Compendium of NP Optimization Problems.

[Day et al., 1993] Day, Y., Imai, H., Iwano, K., and Katoh, N. (1993). How to treat delete requests in semi-online problem. *Proc. 4th Int. Symp. Algorithms and Computation*, pages 44–57.

[de Azevado et al., 1990] de Azevado, J. A., Madeira, J. J. a. E. S., Martins, E. Q. V., and Pires, F. M. A. (1990). A shortest paths ranking algorithm. In *AIRO'90 - Models and Methods for Decision Support*, pages 1001–1011. Associazione Italiana di Ricerca Operativa.

[Delaplace, 2002a] Delaplace, F. (2002a). tagcc : Mode d'emploi. *http ://www.lami.univevry.fr/∼delapla/*.

[Delaplace, 2002b] Delaplace, F. (2002b). Tagcc a language based on transducers and constraints dedicated to biosequences analysis.

[Dubas, 2002] Dubas, A.-L. (2002). Étude d'un mécanisme de résolution par ordonnancenment des solutions pour TAGCC.

[El-Amin and Al-Ghamdi, 1993] El-Amin and Al-Ghamdi (1993). An expert system for transmission line route selection. *Int. Power Engineering Conf*, 2 :697–702.

[Eppstein, 1994] Eppstein, D. (1994). Finding the k shortest paths. In *Proc. 35th Symp. Foundations of Computer Science*, pages 154–165. IEEE.

[Eppstein, 1997] Eppstein, D. (1997). Finding the k shortest paths.

[Frederickson, 1993] Frederickson, G. (1993). An optimal algorithm for selection in a min-heap. *Information and Computation*, 104 :197–214.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability : A Guide to the Theory of Np-Completeness*. W.H. Freeman and Company.

[Hatzivassiloglou and Knight, 1985] Hatzivassiloglou, V. and Knight, K. (1985). Unification-based glossing. In *Proc. 14th Int. Joint Conf. Artificial Intelligence*, pages 1382–1389. http ://www.isi.edu/natural-language/mt/ijcai95-glosser.ps.

[Infobiogen, 2003] Infobiogen (2003). http ://www.infobiogen.fr/services/deambulum/fr/.

[Krishna, 2001] Krishna, G. P. (2001). An efficient Algorithm for finding K disjoint paths of minimum total cost between a pair of nodes in a network.

[Martins and dos Santos, 1999] Martins, E. D. Q. V. and dos Santos, J. L. E. (1999). A new shortest paths ranking algorithm. *Investigação Operational*.

[Martins and Pascoal, 2000a] Martins, E. d. Q. V. and Pascoal, M. M. B. (2000a). A new Implementation of Yen's ranking loopless paths algorithm.

[Martins and Pascoal, 2000b] Martins, E. d. Q. V. and Pascoal, M. M. B. (2000b). An Algorithm for ranking optimal paths.

[Martins et al., 1997] Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1997). A new Algorithm for rankink loopless paths.

[Martins et al., 1998] Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1998). The K shortest loopless paths problem.

[Martins et al., 1999a] Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1999a). An Algorithm for ranking loopless paths.

[Martins et al., 1999b] Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1999b). Deviation algorithm for ranking shortest paths. *International Journal of Foundations of Computer Science*, 10(3) :247–261.

[Martins et al., 1999c] Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (1999c). Labeling algorithm for ranking shortest paths.

[Martins et al., 2000] Martins, E. d. Q. V., Pascoal, M. M. B., and dos Santos, J. L. E. (2000). A new improvement for a k shortest paths algorithm.

[Naor and Brutlag, 1994] Naor, D. and Brutlag, D. (1994). On near-optimal alignements of biological sequences. *J. Computational Biology*, 1(4) :349–366. http ://cmgm.stanford.edu/∼brutlag/Papers/naor94.pdf.

Annexes

Annexe A

Algorithmes de la librairie *graph.ml*

Sont présentés ici les algorithmes de la librairie *graph.ml*.
Ils s'appliquent sur un graphe $\mathcal{G} = (\mathcal{N}, \mathcal{A})$.
Les références à s et t désignent deux sommets de G, tels que s est son sommet initial et t son sommet terminal.

A.1 Tri topologique

Le tri topologique s'applique sur un graphe sans circuit. Le principe de cet algorithme repose sur l'élimination progressive des arcs sortants de chaque sommet. A chaque fois que l'on examine un sommet, on coupe les arcs qui en sortent et on place dans une liste les sommets qui n'ont plus d'arc entrant. Ainsi, on est sûr d'examiner tous les prédécesseurs d'un sommet avant de le numéroter.

La complexité du tri topologique est en $O(m)$ où m est le nombre d'arcs du graphe.

La figure A.1 présente l'algorithme de tri topologique. Les variables $\delta(i)$ indiquent à chaque instant, combien d'arcs non encore éliminés du graphe entrent dans le sommet i. Initialement, $\delta(i) = |w^-(i)|$, où $w^-(i)$ représente l'ensemble des prédécesseur de i, pour tout $i \in \mathcal{N}$. La liste Liste contient l'ensemble des sommets qui n'ont pas de prédécesseurs et qui sont donc directement numérotables.

A.2 Algorithme de Bellman

L'algorithme de Bellman est un algorithme classique qui s'applique sur les graphes sans circuit et qui permet de trouver le plus court chemin entre un sommet initial et tous les autres sommets du graphe ; il permet donc entre autre d'obtenir le plus court chemin entre le sommet initial s et le sommet terminal t.

La complexité de l'algorithme de Bellman est $O(m)$, où m est le nombre d'arcs du graphe.

À chaque sommet j est associé une marque, un *label*, constitué d'un coût π_j, et d'un prédécesseur ξ_j. Le coût π_j correspond à la distance du plus court chemin de s à j, et le prédécesseur ξ_j est le sommet précédent j dans le plus court chemin de s à j. L'ensemble des sommets est marqué petit à petit, en les parcourant dans l'ordre topologique du graphe.
Supposons que l'on connaisse les coûts de tous les sommets de rang topologique inférieur à celui de k. De par la définition de l'ordre topologique, tous les arcs arrivant en k proviennent de sommets de rang inférieur, c'est-à-dire de sommets pour lesquels on connaît déjà les coûts (de

Algo toposort
Entrée : graphe $\mathcal{G} = (\mathcal{N}, \mathcal{A})$
Sortie : tableau de correspondance entre l'index du sommet et son ordre topologique
Début
$\forall i \in \mathcal{N}, \delta(i) \leftarrow 0$
$\forall (i,j) \in \mathcal{A}, \delta(j) \leftarrow \delta(j) + 1$
$Liste \leftarrow \{i \in \mathcal{N} | \delta(i) = 0\}$
$suivant \leftarrow 0$
Tant Que $(Liste \neq \emptyset)$ **Faire**
Retirer un sommet i de $Liste$
$suivant \leftarrow suivant + 1$
$ordre[i] \leftarrow suivant$
Pour Tout $j \in w^+(i)$ **Faire**
$\delta(j) \leftarrow \delta(j) - 1$
Si $(\delta(j) = 0)$
Alors
$Liste \leftarrow Liste \cup \{i\}$
Fin Si
Fin Pour Tout
Fin Tant Que
Retourner *ordre*
Fin

FIG. A.1: Algorithme de tri topologique implémenté dans la librairie *graph.ml*

par l'hypothèse). Alors, pour connaître π_k et ξ_k (et, par la même occasion, le plus court chemin joignant s à k), il suffit de considérer :

$$\pi_k = min\{\pi_j + d_{jk} | j \in w^-(k)\}$$

$$\xi_k = argmin\{\pi_j + d_{jk} | j \in w^-(k)\}$$

On obtient l'algorithme de la figure A.2 initialisé en s, seul sommet sans prédécesseur par définition.

Pour retrouver le chemin, il suffit de prendre ξ_t, puis ξ_{ξ_t}, ... jusqu'à revenir à s.

A.3 Nombre de chemins passant par un arc

Cet algorithme calcule le nombre de chemins d'un graphe et le nombre de chemins différents passant au travers d'un arc, autrement dit le nombre de chemins qui contiennent cet arc.

La méthode de calcul du nombre de chemins passant par un arc se décompose en trois étapes :

1. le calcul du nombre de chemins passant par l'arc et issus du sommet initial s,
2. le calcul du nombre de chemins passant par l'arc et se finissant au sommet terminal t,
3. le calcul du nombre de chemins passant par un arc, qui est égal au produit des deux termes calculés précédemment.

Pour tout $a \in \mathcal{A}$, on note
– $Cs(a)$ (resp. $Ct(a)$) le nombre de chemins depuis s passant par a (réciproquement, depuis t passant par a),

Algo bellman
Entrée : graphe $\mathcal{G} = (\mathcal{N}, \mathcal{A})$, un sommet initial s
Sortie : graphe $\mathcal{G}$ dont les sommets sont marqués
Début
$\pi_s \leftarrow 0$
$\forall i \in \mathcal{N} - \{s\}, \pi_i \leftarrow +\infty$
Numéroter les sommets de $\mathcal{G}$ dans l'ordre topologique
Pour k **variant de** 2 **à** n **Faire**
$\xi_k \leftarrow argmin\{\pi_j + d_{jk} | j \in w^-(k)\}$
$\pi_k \leftarrow \pi_{\xi_k} + d_{\xi_k k}$
Fin Pour
Retourner $\mathcal{G}$
Fin

FIG. A.2: Algorithme de Bellman implémenté dans la librairie *graph.ml*

- $C(a)$ le nombre de chemins passant par a,
- $Pred(a)$ l'ensemble des arcs prédécesseurs de a dans $\mathcal{G}$, c'est-à-dire les arcs entrant dans la queue de a ; si $a \equiv (j,k)$, $Pred(a) = \{(i,j) | i \in w^-(j)\}$,
- $Succ(a)$ l'ensemble des arcs successeurs de a dans $\mathcal{G}$, c'est-à-dire les arcs sortant de la tête de a ; si $a \equiv (j,k)$, $Pred(a) = \{(k,l) | l \in w^+(k)\}$.

Enfin, on note $C_\mathcal{G}$ le nombre de chemins du graphe $\mathcal{G}$.

On a alors

$$\forall a \in \mathcal{A}, Cs(a) = \sum_{u \in Pred(a)} Cs(u)$$

$$\forall a \in \mathcal{A}, Ct(a) = \sum_{u \in Succ(a)} Ct(u)$$

Et pour $Pred(a) = \emptyset$ (resp. $Succ(a) = \emptyset$), on a $Cs(a) = 1$ (resp. $Ct(a) = 1$).
De plus

$$\forall a \in \mathcal{A}, C(a) = Cs(a).Ct(a)$$

$$C_\mathcal{G} = \Sigma_{a \in Pred(t)} C(a) = \Sigma_{a \in Succ(s)} C(a)$$

La figure A.3 illustre ces calculs.

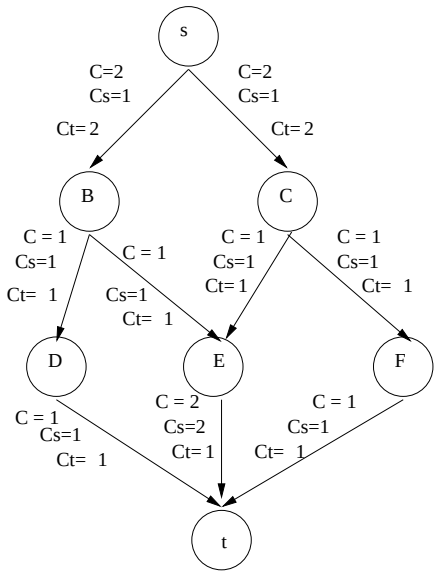

FIG. A.3: Exemple de calculs du nombre de chemins passant par les arcs d'un graphe à $C_{\mathcal{G}} = 4$ chemins.

Annexe B

Librairie *graph.ml*

B.1 Signature de la librairie

La librairie *graph.ml* est basée sur l'utilisation d'un foncteur. Un foncteur est à un module ce qu'une fonction est à une valeur. Autrement dit, le foncteur prend en paramètre un module qui implémente une certaine signature, et permet d'effectuer des opérations identiques quelles que soit l'implémentation de ce module.

Dans notre cas, deux implémentations pour la forme du graphe ont été utilisées.

- Le module `Mat` implémente le graphe sous forme de matrice d'adjacence. La case (i,j) de la matrice contient la distance $d_{i,j}$ de l'arc (i,j).
- le module `HashtblArray` implémente le graphe sous forme d'un vecteur de tables de hachage. La table de hachage de la composante j du vecteur fait correspondre à chacun des prédécesseurs i du sommet j, la distance $d_{i,j}$ de l'arc (i, j).

Ces deux modules ont la même signature `EDGETANK`, ce qui est nécessaire pour être utilisé par le foncteur.

Ci-après, se trouve la signature `EDGETANK`, ainsi que la signature du foncteur `Make`. Le code de la librairie est détaillé dans la section suivante.

B.2 Implémentation de la librairie

Ci-après se trouve l'implémentation de la librairie *graph.ml*.
Cette implémentation est constituée de :
- le module `Mat` pour la matrice d'adjacence ;
- le module `HashtblArray` pour le vecteur de tables de hachage ,
- le foncteur `Make` qui contient quatre sous-modules :
  - le module `Vertex` concerne les fonctions s'appliquant sur les sommets du graphe,
  - le module `Edge` concerne les fonctions s'appliquant sur les arcs du graphe,
  - le module `Succ` concerne les fonctions s'appliquant sur les successeurs d'un sommet du graphe donné en paramètre,
  - le module `Pred` concerne les fonctions s'appliquant sur les prédécesseurs d'un sommet du graphe donné en paramètre,
- les autres fonctions sont celles correspondant aux algorithmes détaillés dans l'annexe A, et à la recherche des K plus courts chemins.

Annexe C

Génération de graphes complets par classe topologique

L'algorithme de la figure C.1 décrit la création d'un graphe complet par classe topologique. Cet algorithme prend en paramètres le nombre de sommets du graphe, et le nombre maximum de sommets par classe topologique.

Algo gengraph
Entrée : n le nombre de sommets, npc le nombre maximum de sommets par classe
Sortie : un graphe complet par classe topologique, et le nombre d'arcs du graphe
Début
    $sommets\_restants = n$
    $arcs = 0$
    Créer un sommet initial
    Tant Que ($sommets\_restants > npc$) **Faire**
        Tirer aléatoirement $sommets$ entre 1 et npc, le nombre de sommets pour la classe
        Créer $sommets$ sommets
        Relier les sommets aux sommets de la classe précédente, la distance des arcs est aléatoire
        Mettre à jour le nombre d'arcs, $arcs$
        $sommets\_restants = sommets\_restants - sommets$
    Fin Tant Que
    Créer un sommet terminal
    Créer $sommets\_restants$ sommets
    Relier les sommets créés aux sommets de la classe topologique précédente
    Mettre à jour le nombre d'arcs, $arcs$
    Relier les sommets créés au sommet terminal, la distance des arcs est nul
    Mettre à jour le nombre d'arcs, $arcs$
    Retourner le graphe et le nombre d'arcs, $arcs$
Fin

FIG. C.1: Algorithme de création d'un graphe complet par classe topologique

---

Ci-après est donnée l'implémentation en *OCaml* de l'algorithme.