A la recherche de l'échafaudage parfait : efficace, de qualité et garanti

Tom Davot

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche : Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier

À la recherche de l'échafaudage parfait : efficace, de qualité et garanti

Présentée par Tom Davot

Le 5 octobre 2020

Sous la direction de Annie Chateau et Rodolphe Giroudeau

Devant le jury composé de

Jean-Claude König	Professeur des universités	LIRMM	Président
Guillaume Blin	Professeur des universités	LaBRI	Rapporteur
Bruno Escoffier	Professeur des universités	LIP6	Rapporteur
Nadia El-Mabrouk	Professeure des universités	Université de Montréal	Examinatrice
Stéphan Thomassé	Professeur des universités	LIP	Examinateur
Mathias Weller	Chargé de recherche CNRS	LIGM	Invité
Annie Chateau	Maîtresse de conférence	LIRMM	Directrice de thèse
Rodolphe Giroudeau	Maître de conférence	LIRMM	Directeur de thèse

UNIVERSITÉ DE MONTPELLIER

Table des matières

Remerciements

Je tiens en premier lieu à remercier mes deux directeurs Annie Chateau et Rodolphe Giroudeau. J'ai beaucoup appris à vos côtés. Durant ces trois ans, vous m'avez accompagné scientifiquement et, plus important, humainement. Ce fut un grand plaisir de travailler et discuter avec vous, de science ou d'autres choses. Je me suis rendu compte que l'on pouvait apprécier un supporter d'un certain club de foot (Annie, j'aurais bien fait une blague sur les jaunards, mais pas sûr que tu sois assez impliquée pour ça fasse mouche).

Merci également à Mathias Weller qui a également pris une part active de mon encadrement. Grâce à toi et tes correction, j'ai à présent une bien meilleure idée de la façon dont il faut rédiger un papier scientifique. Avec évidemment toutes les petites astuces pour rester dans la limite impartie du nombre de pages.

Je remercie les membres de mon jury qui ont pris le temps de lire mon tapuscrit et d'assister à ma soutenance : Jean-Claude König, Guillaume Blin, Bruno Escoffier, Nadia El-Mabrouk, Stéphan Thomassé. Les problèmes techniques ont rendu la soutenance compliquée et ça n'a pas du être agréable non plus d'être de l'autre côté de la visioconférence.

J'ai fait de superbes rencontres au LIRMM et je tiens à remercier toutes les personnes que j'ai côtoyées. En particulier Jocelyn et Lucas sans qui ma vie de thésard n'aurait pas été la même, surtout la première année où je ne connaissais personne. Merci aussi à Louise, membre du comité scientifique de MICRO. Avec un peu de travail ça devient LA conférence majeure internationale en géographie-informatique. Votre amitié et votre présence (distancielle) m'ont beaucoup aidé lors de cette période difficile qu'a été le confinement. Je pense que quand le Hanabi deviendra sport olympique, nous serons prêts.

Merci aux membres de l'équipe MAORE, à Michael, Éric et Benoit avec qui les discussions sur divers sujets sont toujours passionnantes. Merci à mon cobureau, Gabrielsen qui m'a supporté ces deux dernières années, bon tu m'as un peu contaminé au jeu d'échecs par contre. Sam avec qui le retour en vélo à Metz a été épique, juste parce qu'on ne savait pas lire les horaires de bus. Iago avec qui j'ai notamment découvert que la meilleure pizzéria de Montpellier n'était vraisemblablement pas la meilleure pizzéria de Montpellier. Je remercie les doctorants MAREL, Vincent, Nicolas et Pascal, de s'être imposés dans notre salle de pause, finalement vous êtes plutôt sympathiques en fin de compte. Armelle pour ces pauses thé très agréables.

Je remercie également mes collègues micro-électroniciens : Thibault, Loïc (fois deux), Guillaume, Yohan, Théo et Quentin. Je vous laisse à vos FPGA, vos radios, vos raspberry... J'espère que grâce à moi, vous aurez compris que quand vous ne voulez pas vous attaquer à un problème, il suffit de dire qu'il est NP-complet.

Un grand merci à ma famille pour s'être déplacée et m'avoir soutenu pour la soutenance. Enfin, un merci tout particulier à Patrick pour la relecture de ma thèse qui m'a énormément aidé.

Introduction

En biologie, la génomique est une branche qui étudie les génomes. En particulier, elle s'intéresse à déchiffrer l'information qui est contenue au sein de la molécule d'acide désoxyribonucléique (ADN) qui est située dans les cellules des organismes vivants. Cette information est encodée à l'aide d'une séquence de nucléotides et prédétermine notamment certains caractères du vivant. La connaissance de cette séquence génomique est essentielle pour comprendre certains mécanismes biologiques. Elle est également utile pour beaucoup d'applications, on peut citer par exemple la médecine, la police scientifique ou la paléontologie. Pour l'obtenir, il est nécessaire de passer par un processus appelé séquençage. Il s'agit d'une technologie qui a émergé dans les années 1970 et qui n'a cessé d'évoluer jusqu'à aujourd'hui. Seulement, le séquençage nécessite de découper la molécule d'ADN en fragments, appelés lectures, avant d'obtenir l'ordre des nucléotides dans ceux-ci. On obtient donc après ce processus un ensemble de sous-séquences génomiques. Ces sous-séquences nécessitent d'être assemblées afin de pouvoir obtenir la séquence originelle complète. En général, on obtient cette séquence complète en passant par deux étapes : l'assemblage, où les lectures sont assemblées en séquences plus longues appelées contigs et l'échafaudage où les contigs sont échafaudés en séquences génomiques. La quantité de sous-séquences produites par le séquençage est telle que la reconstruction de la séquence complète nécessite l'utilisation de traitements automatisés, appelés algorithmes.

Les algorithmes sont des objets étudiés dans un champ disciplinaire appelé informatique fondamentale. Ils consistent en une suite d'instructions permettant la résolution de problèmes mathématiques. L'utilisation d'un algorithme pour la résolution d'un problème biologique nécessite au préalable de formuler ce problème sous la forme d'un modèle abstrait. La plupart du temps, on utilisera un outil appelé graphe, qui est constitué d'un ensemble de sommets et d'un ensemble d'arêtes qui relient ces sommets. Construire une telle modélisation permet de mettre à profit l'ensemble des connaissances accumulées par les mathématiciens en théorie des graphes et en informatique fondamentale pour résoudre le problème.

Une fois qu'un problème a été modélisé, il faut concevoir un algorithme pour pouvoir le résoudre. Il est facile de construire un algorithme pour la plupart des problèmes, mais il peut être en revanche très difficile d'en trouver un qui soit performant. La performance d'un algorithme s'évalue la plupart du temps par son temps d'exécution par rapport à la taille de l'objet sur lequel il travaille. Étant donné le volume très important de données à traiter pour les problèmes biologiques, l'idéal est un temps d'exécution polynomial par rapport à la taille de l'entrée. En effet, si le temps d'exécution d'un algorithme est exponentiel par rapport à la taille de l'entrée, alors il est illusoire d'es-

pérer le dérouler sur des génomes de taille importante car il mettra trop de temps à donner une solution. Malheureusement, et c'est le cas pour de nombreux problèmes classiques, l'échafaudage ne peut probablement pas être résolu en temps polynomial. Il faut alors trouver des moyens pour produire malgré tout une solution, quitte à dégrader légèrement la qualité du génome produit.

Dans ce tapuscrit, nous étudierons deux problèmes. Le premier est celui de l'échafaudage qui intervient après l'assemblage des lectures produites par le séquençage. Il est modélisé sous la forme de la recherche de chemins et de cycles dans un graphe d'échafaudage, qui vont représenter les chromosomes linéaires et circulaires, respectivement, de l'organisme étudié. Le second problème est la linéarisation. Il intervient directement après l'échafaudage : lorsque l'on prend en compte la possibilité que des contigs puissent apparaître plusieurs fois dans la séquence génomique, des chemins ambigus apparaissent. Ces chemins ambigus empêchent la résolution complète du problème de l'échafaudage, sous peine de possiblement créer des séquences chimériques. Pour supprimer ces ambiguïtés, le problème de la linéarisation est proposé. Ce problème prend en entrée un graphe de solution, qui est constitué de la fusion de tous les cycles et chemins alternés calculés lors de l'échafaudage. L'idée de la linéarisation est de dégrader de façon parcimonieuse la solution construite lors de l'échafaudage en enlevant des arêtes dans le graphe de solution. Le but étant d'enlever tous les chemins ambigus du graphe de solution.

Le travail théorique d'un informaticien peut comporte en deux facettes. La première consiste à développer des algorithmes pour un problème donné, il peut s'agir d'algorithmes exacts, approchés, paramétrés, de recherche locale… La deuxième facette consiste à montrer la complexité des problèmes étudiés, c'est-à-dire la difficulté à obtenir une solution respectant une certaine propriété. Ainsi il est par exemple possible, en admettant certaines hypothèses, de montrer qu'un problème n'admet pas d'algorithme calculant une solution exacte en temps polynomial. Enfin, on peut ajouter une dernière facette au travail d'un informaticien, plus appliquée cette fois-ci. Elle consiste à implémenter ces algorithmes afin d'observer leurs comportements sur des données réelles. Le travail effectué au cours de cette thèse, que nous présentons dans ce document, comprend ces trois facettes. Nous montrerons ainsi des résultats théoriques et pratiques sur le problème de l'échafaudage et sur la linéarisation.

Le premier chapitre sera consacré à définir des termes et propriétés mathématiques nécessaires pour comprendre les résultats que nous présenterons. Dans les premières sections, nous rappellerons quelques bases mathématiques sur la logique propositionnelle et la théorie des ensembles. Une longue section introduira ensuite la définition d'un graphe, qui est un objet central pour les problèmes étudiés ici. Nous présenterons également quelques propriétés simples de la théorie des graphes, entre autres, nous présenterons les notions de sous-graphe et de couplage. Nous montrerons également quelques exemples de classes de graphes. Ensuite, nous décrirons du point de vue mathématique la notion de problème, nous donnerons des exemples classiques de problèmes de graphes et de satisfaisabilité. La notion d'algorithme sera ensuite abordée avant la présentation de classes de complexité. En particulier, nous caractériserons les différences entre plusieurs types d'algorithmes : exact, paramétré, de recherche locale. Certaines hiérarchies de classes de complexité formées autour de ces types algorithmes seront également définies. Enfin, nous finirons ce chapitre par une brève explication

sur ce que sont les solveurs.

Dans le chapitre deux, nous décrirons le contexte biologique dans lequel s'inscrit cette thèse, avant de présenter les problèmes étudiés. Dans les premières sections, nous indiquerons comment fonctionne le séquençage, notamment comment il produit des « lectures » qui sont des sous-séquences connues de la séquence génomique. Ensuite, on définira le problème de l'assemblage qui est un problème important en bio-informatique. Ce problème consiste à assembler les lectures produites par le séquençage en sous-séquences plus importantes, les contigs. S'ensuivra une section sur le problème de l'échafaudage, problème éponyme de cette thèse. Nous décrirons comment un graphe d'échafaudage est construit à partir de l'ensemble des contigs produits avant de décrire de façon formelle ce problème. Un état de l'art de l'échafaudage mettra en perspective les résultats de cette thèse par rapport à la littérature existante. En particulier, nous verrons qu'il existe d'autres façons de modéliser ce problème et également un ensemble de travaux dont les résultats du troisième chapitre sont la continuité. Enfin, nous terminerons ce chapitre en présentant l'outil `scaftool` qui est le programme dans lequel ont été intégrés les algorithmes développés dans cette thèse. Nous présenterons également le jeu de données utilisé pour les tests.

Le chapitre trois présentera des résultats autour du problème de l'échafaudage. Dans un premier temps, nous montrerons des résultats de complexité sur ce problème, en particulier l'appartenance aux classes de complexité NP-complet, APX-complet et poly-APX-complet pour des classes particulières de graphes d'échafaudage. La seconde partie de ce chapitre sera consacrée à la présentation d'un algorithme d'approximation glouton. Cet algorithme a été initialement décrit dans des travaux précédents pour pouvoir s'exécuter sur des graphes complets. Nous montrerons comment cet algorithme peut s'adapter à une classe de graphes particulière : les graphes de clusters connectés. Nous montrerons également comment on peut faire fonctionner cet algorithme glouton dans le cas général à l'aide d'une formulation SAT. Enfin, des détails des tests effectués sur différentes versions de l'algorithme glouton clôtureront ce chapitre. Nous montrerons que la version de l'algorithme glouton développée ici améliore les résultats de celle sur les graphes complets.

Pour finir, le quatrième chapitre portera sur des résultats afférents au problème de la linéarisation. Dans un premier temps, nous étendrons des résultats précédents sur la complexité du problème. Nous explorerons différentes classes de graphes, allant des graphes peu denses à des graphes très denses, afin d'établir un tableau assez complet de la complexité de ce problème. Nous montrerons quelles sont les classes pour lesquelles le problème est difficile à résoudre. Une grande partie de ce chapitre sera également consacrée à la difficulté d'approximation de ce problème, là aussi nous explorerons des classes de graphes peu denses et des classes de graphes denses. Ensuite, nous montrerons des algorithmes exacts et approchés pour résoudre le problème de la linéarisation. Nous finirons ce dernier chapitre en montrant les bonnes performances de ces algorithmes sur des données réelles.

Certains résultats présentés dans cette thèse ont donné lieu aux publications suivantes (dans l'ordre chronologique).

[28] *On the Hardness of Approximating Linearization of Scaffolds Sharing Repeated Contigs*, avec Annie Chateau, Rodolphe Giroudeau et Mathias Weller pour la conférence RECOMB-CG 2018 à Sherbrook (Canada).

[83] *New results about the linearization of scaffolds sharing repeated contigs* avec Annie Chateau, Rodolphe Giroudeau, Dorine Tabary et Mathias Weller pour la conférence COCOA 2018 à Atlanta (États-Unis).

[29] *New Polynomial-Time Algorithm Around the Scaffolding Problem* avec Annie Chateau, Rodolphe Giroudeau et Mathias Weller pour la conférence ALCOB 2019 à Berkley (États-Unis).

[30] *Linearizing Genomes: Exact Methods and Local Search* avec Annie Chateau, Rodolphe Giroudeau et Mathias Weller pour la conférence SOFSEM 2020 à Limassol (Chypre).

L'article [29] décrit l'algorithme approché glouton du chapitre 3 pour le problème de l'échafaudage (théorème 10). Les articles [28, 83] présentent des résultats de complexité, de difficulté d'approximation et d'approximation pour le problème de linérisation du chapitre 3. Enfin l'article [30] porte sur la difficulté de recherche locale et présente des méthodes exactes de résolutions pour le problème de linéarisation.

I

Préliminaires

Dans ce chapitre, nous poserons les bases de la plupart des notions mathématiques abordées au sein de cette thèse. Après avoir introduit la notion de graphe et les descriptions gravitant autour de celle-ci, nous aborderons des notions plus algorithmiques.

1.1 Théorie des graphes et préliminaires mathématiques

Dans cette section, nous allons définir des notions et propriétés issues de la théorie des graphes. Tout d'abord, nous avons besoin d'aborder quelques notions préliminaires sur la logique propositionnelle et sur les ensembles.

1.1.1 Préliminaires sur la logique propositionnelle

En logique propositionnelle, les objets de bases manipulés sont les *propositions*. Une proposition est un énoncé qui peut soit être vrai soit être faux. Dans le premier cas, une proposition prendra la valeur VRAI et dans le second cas, elle prendra la valeur FAUX. Par exemple, la proposition $3 < 5$ est une proposition qui prend la valeur VRAI, à l'inverse la proposition « 3 est un nombre pair » qui prend la valeur FAUX. Une proposition qui dépend de paramètres est appelée *prédicat*. Ainsi, la proposition $x = y$ est un prédicat qui dépend des valeurs de x et y. On peut construire une proposition à partir d'autres propositions en utilisant les opérations suivantes.

— **Négation.** Soit p une proposition, la *négation* de p, notée $\neg p$ est la proposition dont la valeur est égale à VRAI si et seulement si la valeur p est égale à FAUX.

— **Conjonction.** Soient p_1 et p_2 deux propositions. La *conjonction* de p_1 et p_2, notée $p_1 \wedge p_2$ est la proposition dont la valeur est égale à VRAI si et seulement si les valeurs de p_1 et p_2 sont égales à VRAI. Dans le langage courant, on peut remplacer cet opérateur par « ET ».

— **Disjonction.** Soient p_1 et p_2 deux propositions. La *disjonction* de p_1 et p_2, notée $p_1 \vee p_2$ est la proposition dont la valeur est égale à FAUX si et seulement si les valeurs de p_1 et p_2 sont égales à FAUX. Dans le langage courant, on peut remplacer cet opérateur par « OU ».

L'ordre de priorité des opérateurs est $\neg$, $\wedge$ et $\vee$. Nous utiliserons des parenthèses quand cet ordre doit être modifié. Une proposition prenant la valeur VRAI est dite *satisfaite* et dans le cas contraire, elle est dite *insatisfaite*. Pour simplifier certaines opérations, nous introduisons également les opérateurs suivants.

— **Implication.** Soient p_1 et p_2 deux propositions. L'*implication* de p_2 par p_1, notée $p_1 \Rightarrow p_2$ correspond à la proposition $\neg p_1 \vee p_2$.

— **Double Implication.** Soient p_1 et p_2 deux propositions. La *double implication* de p_2 par p_1, notée $p_1 \Leftrightarrow p_2$ correspond à la proposition $(p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$.

1.1.2 Préliminaires sur les ensembles

Un *ensemble* est une collection d'objets uniques, il peut être fini ou infini. La notation la plus communément admise pour décrire le contenu d'un ensemble utilise des accolades. Ces accolades peuvent entourer directement la liste des objets ou un prédicat caractérisant tous les objets de la collection. Par exemple, pour un ensemble E contenant les entiers $1, 2, 3, 4$ et 5, on pourra écrire au choix $\{1, 2, 3, 4, 5\}$, $\{1, \ldots, 5\}$ ou $\{i \mid i \in \mathbb{N}^+ \wedge i \leq 5\}$ (à comprendre : E contient tous les entiers naturels inférieurs ou égaux à 5). L'ensemble des entiers naturels est dénoté $\mathbb{N}$ et l'ensemble des naturels strictement positifs, utilisé dans l'exemple précédent, est dénoté $\mathbb{N}^+$. L'ensemble des nombres réels est noté $\mathbb{R}$ et l'ensemble des nombres réels strictement positifs est noté $\mathbb{R}^+$. Enfin, l'ensemble vide, c'est-à-dire l'ensemble ne contenant aucun élément, est dénoté $\varnothing$. La *cardinalité* d'un ensemble fini E, notée $|E|$ est le nombre d'éléments que E contient.

Pour construire des propositions, on utilisera les notations suivantes. Pour indiquer qu'un objet x appartient à un ensemble E, on utilise le symbole $\in$. Ainsi $1 \in E$ signifie que l'entier 1 appartient à l'ensemble E. Le symbole $\forall$ permet d'exprimer que tous les éléments x d'un ensemble E vérifient une propriété $P(x)$, ce qui se notera $\forall x \in E, P(x)$. Par exemple, la proposition $\forall x \in \{1, \ldots, 5\}, x \leq 5$ est satisfaite. Le symbole $\exists$, permet d'indiquer qu'il existe au moins un élément x de l'ensemble E qui vérifie une propriété $P(x)$, ce qui se notera $\exists x \in E, P(x)$. Ainsi, la proposition $\exists x \in \{1, \ldots, 5\}, x = 6$ est insatisfaite. Les symboles $\forall$ et $\exists$ sont appelés des *quantificateurs* et peuvent être remplacés dans le langage courant par « pour tout » et « il existe », respectivement. Un ensemble E' est *inclus* dans un ensemble E si tous les éléments de E' appartiennent à E et on note cela $E' \subseteq E$. Autrement dit les deux propositions $E' \subseteq E$ et $\forall x \in E', x \in E$ sont équivalentes. S'il existe au moins un élément de E qui n'appartient pas à E', on parle alors d'*inclusion stricte* et on dénote cela par $E' \subset E$. Si E' est inclus dans E, strictement ou non, on dit que E' est un *sous-ensemble* de E.

On peut créer des ensembles en utilisant des opérations sur d'autres ensembles. Une *union* de deux ensembles E_1 et E_2, désignée par l'opérateur $\cup$, est l'ensemble contenant tous les éléments de E_1 et E_2 : $E_1 \cup E_2 = \{x \mid x \in E_1 \vee x \in E_2\}$. Par exemple,

on a $\{1,2,3\} \cup \{3,4,5\} = \{1,2,3,4,5\}$. Une *intersection* de deux ensembles E_1 et E_2, désignée par l'opérateur $\cap$, est l'ensemble contenant les éléments contenus à la fois dans E_1 et E_2 : $E_1 \cap E_2 = \{x \mid x \in E_1 \wedge x \in E_2\}$. Par exemple, on a $\{1,2,3\} \cup \{3,4,5\} = \{3\}$. Deux ensembles pour lesquels l'intersection est nulle sont dits *disjoints*. La *différence* entre un ensemble E_1 et un ensemble E_2, désignée par l'opérateur $\setminus$ est l'ensemble contenant les éléments de E_1 qui ne sont pas dans E_2 : $E_1 \setminus E_2 = \{x \mid x \in E_1 \wedge x \notin E_2\}$. Une *partition* d'un ensemble E est un ensemble d'ensembles disjoints tels que leur union est égale à E. Dans la définition la plus communément admise de la partition, chacun des ensembles de la partition ne doit pas être vide ; nous ne respecterons pas nécessairement cette contrainte ici. Par exemple, une partition de l'ensemble $\{1,2,3,4,5\}$ est donnée par $\{\{1,2\}\{3\},\{4,5\}\}$. L'ensemble des parties d'un ensemble E, noté $\mathcal{P}(E)$ est l'ensemble contenant tous les sous-ensembles de E, par exemple $\mathcal{P}(\{1,2,3\}) = \{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{2,3\},\{1,3\},\{1,2,3\}\}$.

Un *multiensemble* est une collection d'objets qui ne sont pas nécessairement uniques. Un *n-uplet* est un multiensemble de cardinalité n possédant un ordre sur ses éléments. Pour un tel ensemble, nous remplaçons les accolades par des parenthèses lorsque nous décrivons le contenu d'un n-uplet, et la position des objets dans la notation correspond alors à leur ordre. Ainsi les deux 3-uplets $(1,2,3)$ et $(3,2,1)$ ne sont pas égaux car leurs ordres ne sont pas les mêmes.

Une *application* f est une relation entre les éléments d'un ensemble de départ et un ensemble d'arrivée. La notation $f : D \mapsto A$ permet de préciser l'ensemble de départ D et l'ensemble d'arrivée A. Chaque élément de l'ensemble de départ est en relation avec un unique élément de D. Il n'y a pas de contrainte sur le nombre d'éléments auxquels doit être associé chaque élément de l'ensemble d'arrivée. Pour un élément $x \in D$, on note $f(x)$ l'élément de A associé à x par l'application f. Pour un élément $y \in A$, on note $f^{-1}(y)$ le sous-ensemble de D contenant les éléments associés à y par f. L'application $f : D \mapsto A$ est une *bijection* si pour tout élément $y \in A$ on a $|f^{-1}(y)| = 1$.

Pour deux ensembles E_1 et E_2, on définit le *produit cartésien* de E_1 et E_2, noté $E_1 \times E_2$ comme l'ensemble des 2-uplets contenant un élément de chacun des ensembles. Formellement, $E_1 \times E_2 = \{(e_1,e_2) \mid e_1 \in E_1, e_2 \in E_2\}$. Pour un produit cartésien impliquant n ensembles $E_1,\ldots,E_n$, on devrait en principe considérer que l'ensemble résultant contient des 2-uplets de la forme $(e_1,(e_2,(\ldots,e_n)))$ mais on préférera par simplicité de notation considérer que l'ensemble résultant est constitué de n-uplets de la forme $(e_1,\ldots,e_n)$. Le produit cartésien d'un ensemble E sur lui-même est noté E^2 que l'on peut généraliser avec la notation E^n si le produit cartésien est effectué n fois. Comme exemple, si l'on prend les ensembles $E_1 = \{1,2,3\}$ et $E_2 = \{a,b\}$, on a les produits cartésiens $E_1 \times E_2 = \{(1,a),(1,b),(2,a),(2,b),(3,a),(3,b)\}$ et $E_2^2 = \{(a,a),(a,b),(b,a),(b,b)\}$.

La *somme de deux ensembles* d'entiers E_1 et E_2, notée $E_1 + E_2$ est l'ensemble des entiers qui sont la somme d'un entier de e_1 et d'un entier de e_2 : autrement dit $E_1 + E_2 = \{e_1 + e_2 \mid e_1 \in E_1, e_2 \in E_2\}$. Par exemple, pour deux ensembles $E_1 = \{1,4,7\}$ et $E_2 = \{1,2\}$, on a $E_1 + E_2 = \{2,\ldots,9\}$.

1.1.3 Origines et définition des graphes

Les deux problèmes biologiques étudiés dans ce tapuscrit sont modélisés en utilisant des objets mathématiques, appelés graphes. Les graphes ont été introduits par le mathématicien Leonhard Euler pour résoudre le problème des ponts de Könisberg [59]. La figure 1.1 présente brièvement ce problème. Notons toutefois que le graphe présenté dans cette figure prend un peu de liberté avec la vérité historique afin de mieux s'intégrer au thème général de cette thèse. Les graphes sont des objets mathématiques relativement simples permettant de modéliser de nombreux problèmes de la vie réelle. Ils permettent, entre autres, de modéliser des relations binaires entre plusieurs entités : cela peut être des relations entre des personnes (amis sur les réseaux sociaux, relations de parenté,...), des machines dans un réseau ou des classes dans un programme. Dans cette thèse, nous utiliserons les graphes pour modéliser des liaisons entre des segments d'ADN.

Formellement, un *graphe* G est un objet constitué de deux ensembles : un ensemble de *sommets*, noté $V(G)$ (pour *vertices* en anglais) et un ensemble d'*arêtes* , noté $E(G)$ (pour *edges* en anglais), composé d'ensembles non ordonnés de deux sommets. On utilise la notation uv pour désigner une arête entre les deux sommets u et v, si les deux sommets sont connus. On dit que les sommets u et v sont les *extrémités* de uv et que uv est *incidente* à u et à v. Deux sommets u et v sont *adjacents* ou *voisins* s'il existe une arête incidente à u et à v. De même, deux arêtes e_1 et e_2 sont *adjacentes* si elles partagent une extrémité, c'est-à-dire si $e_1 \cap e_2 \neq \varnothing$. Pour un sommet u, on note $N(u)$ l'ensemble des voisins de u (pour *neighbours* en anglais). Le *degré* d'un sommet u, noté $deg(u)$ correspond au nombre de voisins de u, autrement dit $deg(u) = |N(u)|$. On note $\Delta(G)$ la valeur maximum des degrés des sommets du graphe G (ou Δ s'il n'y a pas d'ambiguïté sur le graphe). L'*ordre* d'un graphe est le nombre de sommets du graphe.

1.1.4 Définitions et propriétés des graphes

Nous présentons quelques définitions et propriétés de la théorie des graphes qui seront utilisées dans ce tapuscrit. Cette section se contentera de définir uniquement les propriétés qui serviront dans cette thèse. Pour une étude plus approfondie, nous référons le lecteur à des ouvrages plus complets comme celui écrit par Bondy et Murty [16] ou celui écrit par Diestel [33].

1.1.4.1 Sous-graphes

Soit G un graphe. On a parfois besoin de se restreindre à certaines parties du graphe pour définir un algorithme ou une propriété. Pour cela, on introduit la notion de *sous-graphe*. Un graphe H est un sous-graphe de G si $V(H) \subseteq V(G)$ et $E(H) \subseteq E(G)$. On dit dans ce cas que G est un *supergraphe* de H. On distingue deux types particuliers de sous-graphes.

— H est un *sous-graphe induit* s'il se restreint à un sous-ensemble de sommets. Formellement, soit $V' \subseteq V(G)$ un ensemble de sommets de G, on dit que H est induit

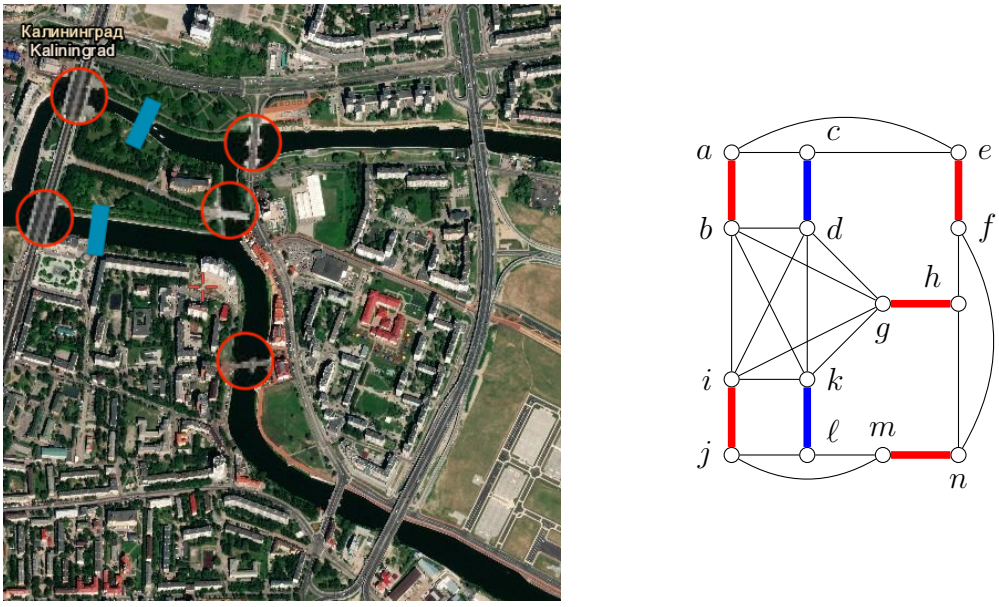


FIGURE 1.1 – *Problème des sept ponts de Königsberg. Le problème que Euler cherchait à résoudre consistait à trouver un parcours pour une promenade traversant une seule fois chacun des ponts et revenant au point de départ.* ***Gauche :*** *vue satellite de la ville de Königsberg, prise sur le moteur Esri. La ville est maintenant appelée Kaliningrad et seuls cinq des sept ponts subsistent aujourd'hui (entourés en rouge) car deux ponts ont été détruits pendant la seconde guerre mondiale, leurs emplacements sont dessinés en bleu sur la vue.* ***Droite :*** *une façon de modéliser le modèle sans utiliser de multigraphe. Chaque sommet représente une extrémité d'un pont, la présence d'une arête entre deux sommets indique qu'il est possible de se déplacer d'une de ces extrémités à l'autre. Les arêtes dessinées en gras représentent les ponts.*

par V' si $V(H) = V'$ et $E(H) = \{uv \mid uv \in E(G) \wedge uv \subseteq V(G)\}$. On peut noter $G[V']$ le sous-graphe de G induit par les sommets de V'.

— H est un *sous-graphe partiel* de G s'il contient tous les sommets G (seules des arêtes ont été enlevées du graphe original).

La figure 1.2 donne des exemples de sous-graphes.

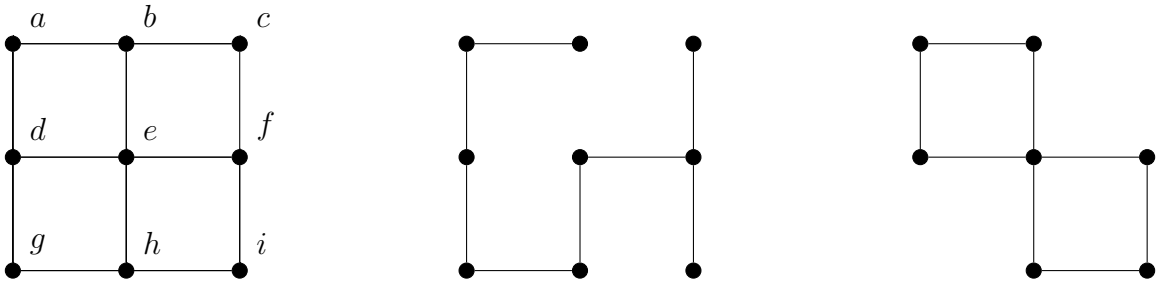


FIGURE 1.2 – *Illustration de la définition des sous-graphes. À gauche, le graphe original G, au centre un sous-graphe partiel de G et à droite, le sous-graphe de G induit par l'ensemble $\{a, b, d, e, f, h, i\}$.*

1.1.4.2 Graphes particuliers

Une *classe de graphes* est un ensemble de graphes partageant une même propriété. On présente ici quelques classes classiques qui seront utilisées par la suite.

Un *chemin* est un graphe constitué d'une suite d'arêtes consécutives telle que $\Delta \leq 2$. Autrement dit, un graphe est un chemin s'il existe un n-uplet $(u_1, \ldots, u_n)$ de sommets uniques tels que pour toute paire d'entiers (i, j) avec $0 < i < j \leq n$, il existe une arête entre u_i et u_j si et seulement si $j = i+1$. La *longueur* d'un chemin correspond au nombre d'arêtes qu'il contient. Les deux sommets u_1 et u_n sont les *extrémités* du chemin. On utilisera d'ailleurs ce n-uplet pour désigner un tel chemin. Un graphe G est *connexe* si pour chaque paire de sommets (u, v), il existe un chemin ayant pour extrémités u et v. Intuitivement, cela veut dire qu'à partir d'un sommet u, on peut atteindre tous les autres sommets du graphe en se déplaçant sur les arêtes. Une *composante connexe* d'un graphe est un sous-graphe induit maximal connexe. Dit autrement un sous-graphe pour lequel on ne peut pas ajouter un autre sommet sans briser la propriété de connexité. Une arête e d'un graphe G est appelée un *isthme* si le sous-graphe partiel possédant les arêtes $E(G) \setminus \{e\}$ possède une composante connexe de plus que G, autrement dit si la suppression de e sépare une composante connexe en deux. Une notion similaire pour un sommet est le *point d'articulation*. Un sommet v d'un graphe est un point d'articulation si le graphe induit par les sommets $V(G) \setminus \{v\}$ contient une composante connexe de plus que G.

Un graphe k-régulier est un graphe tel que tous ses sommets sont de degré k. En particulier, un graphe 0-régulier est un *stable*, il s'agit d'un ensemble de sommets isolés. Un graphe 1-régulier est un *couplage*, il s'agit d'un ensemble d'arêtes disjointes. Un graphe 2-régulier connexe est un *cycle*. De la même manière qu'un chemin, on peut utiliser un n-uplet de sommets $(u_1, \ldots, u_n)$ pour désigner un cycle, il faut alors rajouter la condition qu'il existe une arête entre les sommets u_1 et u_n ; on qualifiera également sa longueur comme étant son nombre d'arêtes. Un graphe 3-régulier est dit *cubique* (un sous-graphe strict d'un graphe cubique est dit *sous-cubique*). Un graphe n-régulier d'ordre $n + 1$ est appelé graphe complet, il s'agit du graphe contenant toutes les arêtes possibles. Notons que pour un graphe quelconque G, une *clique* de G est un sous-ensemble maximal de sommets de G tel que le graphe induit par cet ensemble est un graphe complet.

Un *graphe scindé* (*split graph* en anglais) est un graphe G pour lequel il existe une partition de ses sommets en deux ensembles S et C tels que le graphe induit par S est un stable et le graphe induit par C est une clique. Un *cograph*e est un graphe pour lequel il existe une partition de ses sommets en deux cliques. Un *graphe biparti* est un graphe pour lequel il existe une partition de ses sommets en deux stables. Un *graphe biparti complet* est un graphe biparti maximal au niveau des arêtes. Un exemple est donné pour les trois classes précédentes dans la figure 1.3. Une *grille* $n \times m$ est un graphe ayant pour ensemble de sommets $\{u_{x,y} \mid 0 < x \leq n \wedge 0 < y \leq m\}$ et pour ensemble d'arêtes $\{u_{x,y}u_{x+1,y} \mid 0 < x < n \wedge 0 < y \leq m\} \cup \{u_{x,y}u_{x,y+1} \mid 0 < x \leq n \wedge 0 < y < m\}$, le graphe de gauche dans la figure 1.2 est un exemple de grille. Un *arbre* est un graphe connexe pour lequel il n'existe pas de cycle induit, un exemple d'arbre est donné par le graphe du milieu dans la figure 1.2. Il est possible d'*enraciner* un arbre en un de ses

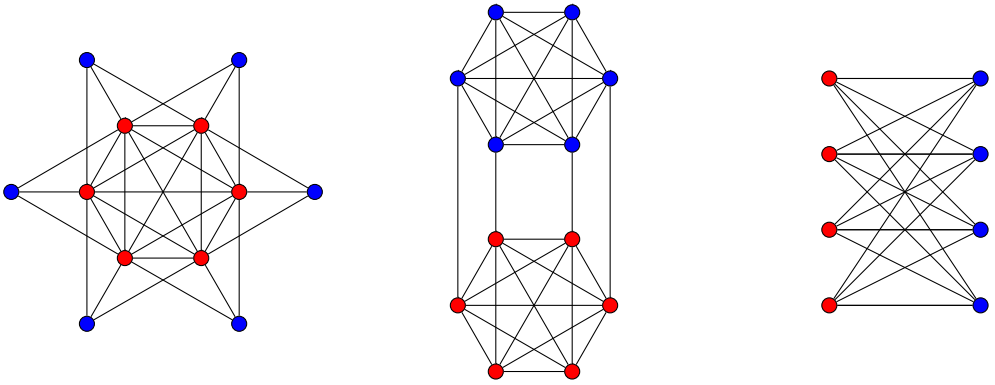


FIGURE 1.3 – *Exemples de graphes particuliers, pour ces trois graphes, la partition des sommets est donnée par l'ensemble des sommets colorés en rouges et l'ensemble des sommets colorés en bleus.* ***Gauche :*** *graphe scindé.* ***Centre :*** *Cographe.* ***Droite :*** *Biparti complet.*

sommets, noté r. Dans ce cas, pour tout sommet u, le parent de u est son voisin dans le chemin ayant pour extrémités u et r. Les enfants d'un sommet u sont les sommets ayant pour parent u. Les sommets n'ayant pas d'enfants sont appelés *feuilles*. La *hauteur* d'un arbre est la longueur du plus long chemin entre la racine et une des feuilles.

Un graphe est *planaire* s'il est possible de le dessiner sur un plan sans que ses arêtes ne se croisent. Dans les classes de graphes que nous avons présentées précédemment, les chemins, les cycles, les arbres et les grilles sont des graphes planaires. À l'inverse, les graphes complets, les graphes bipartis complets, les cographes, les graphes scindés ne sont la plupart du temps pas planaires. Un graphe G est dit *dense* si son nombre d'arêtes est proche de celui du graphe complet à $|V(G)|$ sommets. Les graphes planaires sont en général considérés comme des graphes peu denses.

1.1.4.3 Définitions autour du couplage

Le couplage est une propriété importante des graphes pour ce tapuscrit. Un *couplage parfait* d'un graphe G est un ensemble d'arêtes M^* tel que le sous-graphe partiel G' avec $E(G') = M^*$ est un couplage. Autrement dit, pour chacun des sommets u de G il existe exactement une arête incidente à u dans M^*. Ainsi, dans G', chaque sommet u possède exactement un voisin, que l'on note $M^*(u)$. On dit que u est *couplé* avec $M^*(u)$. On parlera d'arêtes *couplantes* (ou éventuellement de *couplage*) ou *non couplantes* suivant qu'elles appartiennent au couplage ou non. On aura parfois besoin de désigner le voisinage d'un sommet sans prendre en compte le sommet couplé. On notera N^* un tel voisinage. Formellement, on a $N^* = N \setminus \{M^*(u)\}$. Étant donné un couplage, un *élément alterné* est un chemin ou cycle $(u_1, \ldots, u_n)$ où n est pair et tel que pour tout $i \leq n$ impair, $M^*(u_i) = u_{i+1}$. Plus spécifiquement, on pourra utiliser les termes *chemin alterné* ou *cycle alterné*. À titre d'exemple, on pourra regarder la figure 1.1 : les arêtes représentant les ponts forment un couplage et pour résoudre le problème soulevé par Euler, il faut trouver un cycle alterné contenant toutes les arêtes couplantes. Dans cet exemple, il n'existe pas de tel cycle. La *maille alternée*, notée g^*, d'un graphe doté d'un couplage est le nombre d'arêtes couplantes dans le plus petit cycle alterné du graphe. Une *marche* est une suite d'arêtes consécutives qui, contrairement à un chemin ou à un cycle, peut « traverser » plusieurs fois un même sommet ou une même arête. De

même, on peut définir une marche en exhibant un n-uplet $(u_1, \ldots, u_n)$ de sommets, non nécessairement uniques, tel que pour tout $i \leq n$, les sommets u_i et u_{i+1} sont voisins. Une marche est dite *ouverte* si $u_1 \neq u_n$ et *fermée* dans le cas contraire. De même, une marche est *alternée* si n est pair et pour tout $i \leq n$ impair, l'arête u_iu_{i+1} appartient au couplage. En revenant à notre exemple des ponts de Könisberg, on peut trouver une promenade parcourant tous les ponts si l'on s'autorise à traverser plusieurs fois un même pont. La solution consistera alors en une marche alternée fermée. Le nombre de fois qu'une marche traverse une arête correspond à sa *longueur* ; ainsi si une marche traverse deux fois une arête, sa longueur augmente également de deux. On supposera qu'une longueur d'une marche est toujours finie.

1.1.4.4 Décomposition arborescente

Un outil très utilisé pour résoudre des problèmes sur les graphes est la *décomposition arborescente*. Cet outil a été découvert indépendamment par plusieurs auteurs. Il a surtout été popularisé par Neil Robertson et Paul Seymour dans leurs travaux sur les mineurs de graphes [75]. Sa définition est la suivante.

Définition 1

Soit G un graphe. Une décomposition arborescente de G est un arbre T possédant les propriétés suivantes.

- Chaque sommet $B \in V(T)$ de T est un sous-ensemble de $V(G)$. On appelle les sommets de T des *sacs*.
- Pour chaque arête uv de G, il existe un sac B tel que $uv \subseteq B$.
- Pour chaque sommet u de G, le sous-graphe induit par les sacs contenant u est connexe.

La *largeur* de T, notée $w(T)$ est égale à la cardinalité du plus grand sac de $V(T)$. Soit T_{min} une décomposition arborescente de G telle que sa largeur soit minimum. La *largeur arborescente* G, notée $tw(G)$ est égale à $w(T) - 1$.

La largeur arborescente permet de voir à quel point un graphe connexe est proche d'un arbre : en effet un graphe connexe est un arbre si et seulement si sa largeur arborescente est égale à un. Une propriété intéressante d'une décomposition arborescente T d'un graphe connexe G est que chaque sac B est un *séparateur* : si on enlève les sommets de B dans G, alors le graphe n'est plus connexe, autrement dit $G[V(G) \setminus B]$ possède deux composantes connexes. À titre, d'exemple, nous pouvons regarder la figure 1.9, qui est donnée plus loin dans ce chapitre.

1.2 Problèmes algorithmiques

1.2.1 Définition

Un *problème* est une notion centrale dans le domaine de l'informatique fondamentale. Il s'agit d'une question dépendant d'un certain nombre de paramètres à laquelle il faudra répondre. On distingue deux principaux types de problèmes : les *problèmes de décision* pour lesquels il faudra répondre OUI ou NON et les *problèmes d'optimisation* pour lesquels il faudra trouver une « meilleure » solution, en fonction d'un ou plusieurs paramètres. Les paramètres d'un problème peuvent prendre plusieurs valeurs potentielles ce qui détermine un certain nombre d'entrées possibles pour le problème, qui sont alors appelées *instances*. Résoudre une instance consistera à répondre à la question du problème pour cette instance. À titre d'exemple, on peut revenir sur le problème des ponts de Könisberg. Euler voulait savoir s'il existait une promenade passant par tous les ponts, il s'agit donc d'un problème de décision. On peut généraliser ce problème et le formuler de la façon suivante.

PROMENADE DES PONTS

Entrée : Un graphe G muni d'un couplage parfait M^*.
Question : Existe-t-il un cycle alterné contenant toutes les arêtes du couplage ?

Le graphe de la figure 1.1 est une instance de ce problème et la réponse pour cette instance est NON. On parle alors d'*instance négative*. Pour les instances pour lesquelles la réponse est OUI, on parle d'*instance positive*. Si l'on souhaite effectuer une promenade qui passe par le plus grand nombre de ponts sans passer deux fois par le même et sans revenir à son point de départ, on peut formuler un problème d'optimisation.

PROMENADE DES PONTS MAXIMUM

Entrée : Un graphe G muni d'un couplage parfait M^*.
Sortie : Un chemin alterné de longueur maximum.

Le critère à optimiser ici est la longueur du chemin alterné, que l'on cherche à maximiser. On parle ainsi de problème de maximisation. On pourrait formuler de la même façon un problème de minimisation, dans lequel il faudrait trouver une marche alternée contenant toutes les arêtes de couplage minimisant sa longueur.

Le terme *solution* prend différents sens en fonction du type de problème. Pour un problème de décision, on parlera de l'objet permettant de répondre favorablement à la question. Pour le problème PROMENADE DES PONTS, il s'agira d'exhiber un cycle alterné de longueur $2|M^*|$. Il s'agit en vérité d'un abus de langage, dans les manuels de complexité, on parle plutôt de *certificat*. Pour un problème d'optimisation, il s'agira de l'objet répondant au problème mais n'optimisant pas nécessairement le critère souhaité. Pour le problème PROMENADE DES PONTS MAXIMUM, n'importe quel cycle al-

terné est une solution. Pour la solution optimisant le critère souhaité, on parlera de *solution optimale*. Notons qu'il est toujours possible de « transformer » un problème d'optimisation en problème de décision en autorisant une valeur maximale ou minimale sur le critère à optimiser. Par exemple, on pourra transformer le problème PROMENADE DES PONTS MAXIMUM comme suit.

k-PROMENADE DES PONTS MAXIMUM

Entrée : Un graphe G muni d'un couplage parfait M^* et un entier k.
Question : Existe-t-il un chemin alterné de longueur au moins k ?

On parle alors de *problème de décision associé* au problème d'optimisation. Il est facile de voir que l'on sait trouver une solution optimale à un problème d'optimisation si et seulement si on sait trouver une solution au problème de décision associé pour n'importe quelle valeur de k.

Dans les deux sous-sections qui suivent nous allons présenter des problèmes classiques en informatique théorique qui seront utilisés dans ce document.

1.2.2 Problème de graphes

Nous présentons ici des problèmes classiques d'optimisation se rapportant à des graphes. Soit G un graphe. On appelle une *couverture* $V' \subseteq V(G)$ de G, un ensemble de sommets tel que $\forall uv \in E(G), uv \cap V' \neq \varnothing$, autrement dit, tel que chaque arête de G est incidente à au moins un sommet de V'. Pour un ensemble de sommets V' quelconque, on dit que l'arête uv est *couverte* par V' si u ou v appartiennent à V'. Trouver la plus petite couverture dans un graphe est un problème classique qui peut être formulé comme suit.

COUVERTURE PAR SOMMETS

Entrée : Un graphe G.
Sortie : Une couverture V' de taille minimum.

Remarquons que, étant donné une couverture V' d'un graphe G, les sommets de G n'appartenant pas à cette couverture induisent un stable. De même, pour n'importe quel ensemble de sommets I de G, induisant un stable, les sommets de G n'appartenant pas à I forment une couverture. Un ensemble de sommets induisant un stable est appelé *ensemble indépendant*. On peut formuler le problème suivant.

ENSEMBLE INDÉPENDANT

Entrée : Un graphe G.
Sortie : Un ensemble indépendant V' de taille maximum.

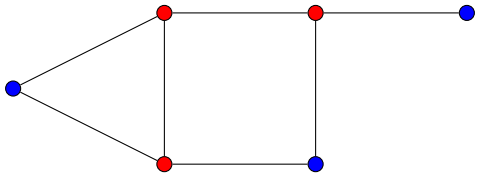


FIGURE 1.4 – *Exemple de la complémentarité d'une couverture et d'un ensemble indépendant. Les sommets colorés en rouge forment une couverture, tandis que les sommets colorés en bleu forment un ensemble indépendant.*

Ces deux problèmes sont illustrés par la figure 1.4.

Pour les deux derniers problèmes de cette section, nous avons besoin d'introduire un nouveau type de graphes : les *graphes orientés*. Un graphe orienté est un graphe dans lequel on a donné un ordre sur les sommets pour chacune des arêtes, alors appelées *arcs*. Pour un sommet u d'un graphe orienté, un arc de la forme (v, u) est un *arc entrant* et un arc de la forme (u, v) est un *arc sortant*. On dénote par $A(G)$ l'ensemble des arcs d'un graphe orienté. Notons que pour deux sommets u et v, il peut exister à la fois l'arc (u, v) et l'arc (v, u). Quand on indique un chemin ou un cycle $(u_1, \ldots, u_k)$ dans un graphe orienté, il doit respecter l'orientation des arcs : c'est-à-dire que pour deux sommets consécutifs u_i et u_{i+1}, l'arc (u_i, u_{i+1}) doit exister dans le graphe.

Étant donné un graphe orienté, un des problèmes en théorie des graphes consiste à trouver un chemin ou un cycle de longueur maximum.

PLUS LONG CHEMIN (RESP. CYCLE)

Entrée : Un graphe orienté G.
Sortie : Un chemin (resp. cycle) de longueur maximum.

Enfin, le dernier problème sur les graphes que nous introduisons est le VOYAGEUR DE COMMERCE qui n'est à l'origine pas un problème de graphes. Dans ce problème, un voyageur doit visiter un ensemble de villes et revenir à son point de départ tout en minimisant le temps total nécessaire pour effectuer le parcours. Nous pouvons modéliser ce problème à l'aide d'un graphe orienté complet G où chaque sommet représente une ville. Pour représenter la distance, nous utilisons une fonction de coût $c : V(G)^2 \mapsto \mathbb{N}$ renvoyant le temps utilisé lorsqu'on emprunte l'arc entre deux sommets. Le parcours est modélisé à l'aide d'un cycle $C = (u_1, \ldots, u_n)$ (avec $u_1 = u_n$) et le coût de cycle correspond à la valeur $\sum_{i<n} c(u_i, u_{i+1})$. Dans ce document, nous utiliserons une version modifiée du problème de base qui est dite *asymétrique* car les valeurs $c(u_i, u_j)$ et $c(u_j, u_i)$ peuvent être différentes (cela peut se justifier par exemple si le voyageur utilise des skis et que les villes sont situées à des altitudes différentes). Contrairement au problème de base, nous chercherons à maximiser la durée du voyage. On formule ce problème de la façon suivante.

VOYAGEUR DE COMMERCE ASYMÉTRIQUE MAXIMUM

Entrée : Un graphe complet G et une fonction de coût $c : V^2 \mapsto \mathbb{N}$ asymétrique.
Sortie : Un cycle contenant tous les sommets de G, tel que le coût de C est maximisé.

1.2.3 Problème de satisfaisabilité

Le deuxième type de problème que nous utiliserons est celui des problèmes de satisfaisabilité. Ce type de problème implique des formules propositionnelles. Une *variable booléenne* est une variable à laquelle on peut assigner soit la valeur VRAI, soit la valeur FAUX. Un *littéral* est une proposition qui est soit égale à une variable booléenne, soit égale à une négation d'une variable booléenne. Une *clause* est une disjonction de littéraux, par exemple, étant donné trois variables booléennes x_1, x_2 et x_3, le prédicat $x_1 \vee \neg x_2 \vee x_3$ est une clause impliquant les trois littéraux $x_1, \neg x_2$ et x_3. Une *formule booléenne* sous *forme normale conjonctive* est une formule propositionnelle composée de conjonctions de clauses. Ainsi la formule $\varphi = (x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3)$ est une formule booléenne composée des trois clauses $C_1 = x_1 \vee x_2, C_2 = \neg x_2 \vee x_3$ et $C_3 = x_1 \vee \neg x_3$. Pour une variable booléenne x_i, on note ψ_i la liste des clauses dans laquelle la variable apparaît, c'est-à-dire les clauses contenant le littéral x_i ou le littéral $\neg x_i$. Une assignation β des variables de φ est une attribution de valeurs VRAI ou FAUX à chacune des variables booléennes qui composent la formule. On note par $\beta(x_i)$ la valeur attribuée à la variable x_i dans β. La formule φ est *satisfaisable* s'il existe une assignation qui satisfait φ et *insatisfaisable* dans le cas contraire. Savoir si oui ou non une formule est satisfaisable est un problème très important en informatique fondamentale. Il peut être formulé comme suit.

SATISFAISABILITÉ

Entrée : Une formule booléenne φ sous forme normale conjonctive.
Question : φ est elle satisfaisable ?

Pour une formule booléenne sous forme normale conjonctive φ, on note G_φ le graphe tel que l'ensemble des sommets de G_φ est composé par l'ensemble des variables et des clauses de φ et l'ensemble des arêtes est donné par $E(G_\varphi) = \{x_i C_\ell \mid C_\ell \in \psi_i\}$ (on ajoute une arête entre une variable et les clauses dans lesquelles elle apparaît). La figure 1.5 donne un exemple d'un tel graphe.

Le problème SATISFAISABILITÉ a été très étudié et de nombreuses variantes existent pour lesquelles des contraintes sont données sur les instances.

— k-SATISFAISABILITÉ : chaque clause composant la formule contient exactement k littéraux. Par exemple la formule $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4)$ est une formule 2-SAT.

— Monotone : chaque clause contient exclusivement des littéraux positifs ou des littéraux négatifs. Par exemple la formule $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$ est une

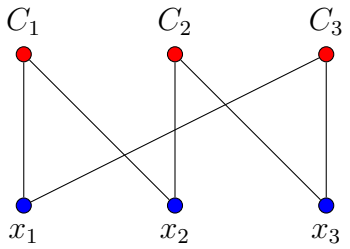


FIGURE 1.5 – *Exemple de graphe G_φ pour la formule $\varphi = (x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3)$.*

formule monotone.

— Planaire : le graphe de la formule est planaire. C'est le cas de la formule illustrée par la figure 1.5 car l'arête entre C_3 et x_1 peut être courbée pour ne pas croiser d'autres arêtes.

— SATISFAISABILITÉ (b) : le nombre d'occurrences de chaque variable x_i est borné par b, autrement dit $|\psi_i| \leq b$. Ainsi la formule $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee x_4)$ est une instance de SAT(3).

Notons qu'il est tout à fait possible de combiner ces contraintes : on peut ainsi formuler le problème 3-SATISFAISABILITÉ(4) Planaire Monotone. Pour les formules insatisfaisables, il peut être intéressant de trouver une assignation qui maximise le nombre de clauses satisfaites. On peut donc formuler le problème d'optimisation suivant.

SATISFAISABILITÉ MAXIMUM

Entrée : Une formule booléenne sous forme normale conjonctive φ.
Sortie : Une assignation β des variables de φ telle que le nombre de clauses satisfaites par β soit maximum.

On peut là aussi utiliser les mêmes variantes que pour le problème de décision. Une autre variante intéressante concernant ce problème d'optimisation est celle où les clauses sont pondérées. Formellement, une formule booléenne pondérée est composée du 2-uplet (φ, ω) tel que φ est une formule booléenne sous forme normale conjonctive et ω est une fonction attribuant un poids positif à chacune des clauses. La version impliquant une formule booléenne pondérée est décrite comme suit.

SATISFAISABILITÉ MAXIMUM PONDÉRÉE

Entrée : Une formule booléenne pondérée (φ, ω).
Sortie : Trouver une assignation β des variables de φ telle que la somme des poids des clauses satisfaites par β soit maximum.

On dénote par $\omega(\varphi)$ la somme totale des poids des clauses et par $\omega(\beta)$ la somme des poids des clauses satisfaites par l'assignation β.

Pour les formules booléennes dans le cas général, c'est-à-dire non nécessairement sous forme normale conjonctive, on peut utiliser une représentation sous la forme d'un

graphe particulier appelé *circuit booléen*. Étant donné une formule booléenne φ, un circuit booléen C de φ est un graphe orienté tel que l'ensemble des sommets de C peuvent être partitionnés en trois ensembles I,O et P tels que :

— chaque sommet de I correspond à une variable de φ et il n'existe pas d'arc entrant incident à un sommet de I ;

— O est composé d'un unique sommet, appelé *sortie* et il existe un unique arc sortant incident à ce sommet ;

— chaque sommet p de P, appelé *porte*, correspond à une fonction logique $p : \{0,1\}^n \mapsto \{0,1\}$.

Pour une assignation β des variables booléennes de φ, on peut évaluer si φ est satisfaite par β en évaluant successivement les valeurs de sortie des portes en partant des sommets de I et en suivant l'orientation des arcs. Les variables assignées à VRAI envoient la valeur 1 sur leurs arcs sortants et les variables assignées à FAUX envoient la valeur 0 sur leurs arcs sortants. Les portes envoient sur leurs arcs sortants le retour de leur fonction logique, en prenant en entrée les valeurs données sur leurs arcs entrants. L'assignation β satisfait φ si et seulement si la valeur 1 est envoyée à la sortie de C, dans ce cas on dit également que β satisfait C. Un problème que l'on peut formuler sur les circuits booléens est le suivant.

CIRCUIT BOOLÉEN

Entrée : Un circuit booléen C.
Sortie : Une assignation β qui satisfait C.

Notons qu'il existe plusieurs circuits booléens possibles pour représenter une même formule φ, en fonction des portes utilisées dans le circuit. On appelle *hauteur* du circuit booléen la longueur du plus long chemin partant d'un sommet de I et allant vers la sortie. La *trame* (*weft* en anglais) d'un circuit booléen correspond au nombre de portes ayant au moins trois arcs entrants incidents. On peut la plupart du temps diminuer la valeur de la trame en augmentant la hauteur du circuit booléen. Des exemples de circuits booléens sont donnés par la figure 1.6.

1.3 Résolution des problèmes

1.3.1 Algorithme

Pour trouver une solution à un problème, on utilise un *algorithme*. Un algorithme est tout simplement une suite finie d'instructions qui ne sont pas ambiguës, c'est-à-dire qui ne nécessitent pas d'interprétation de la part du lecteur. Un algorithme nécessite en général une entrée et retourne une sortie répondant au problème désiré. Pour effectuer des calculs intermédiaires, un espace de stockage est nécessaire. Cet espace de stockage apparaît sous la forme de variables pour lesquelles on peut attribuer des valeurs. Plus le nombre de variables nécessaires est grand, plus l'espace de stockage nécessaire est important. Il existe plusieurs moyens d'écrire des algorithmes. Par exemple, pour un

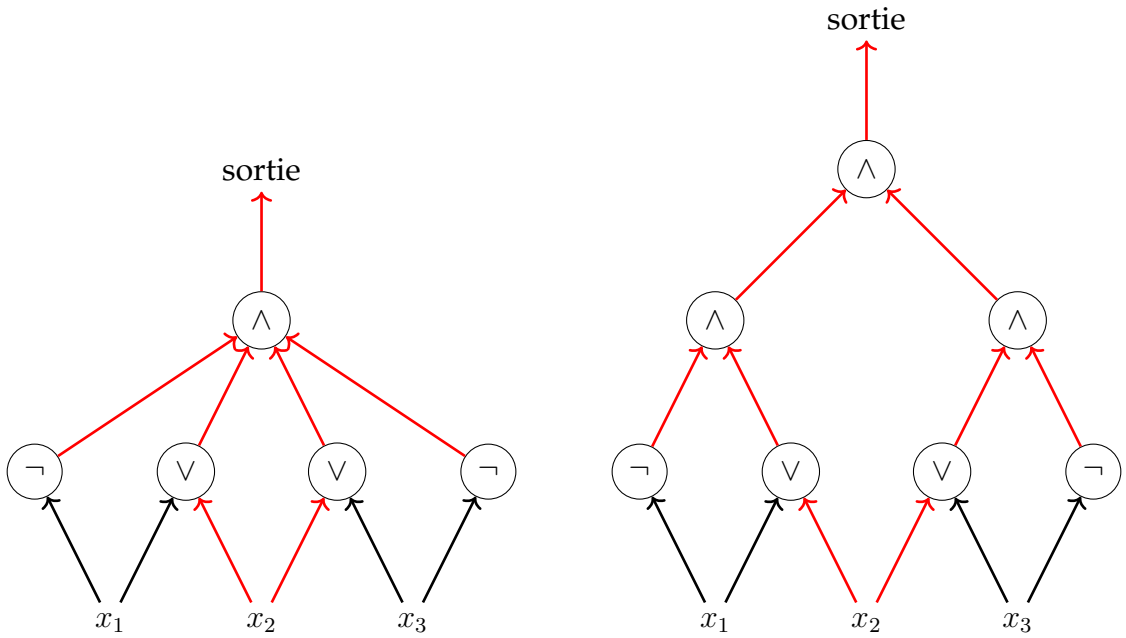


FIGURE 1.6 – *Deux exemples de circuits booléens pour la formule booléenne* $\neg x_1 \wedge (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge \neg x_3$. *Les flèches sur les arcs représentent leurs orientations. Lors de l'évaluation, les arcs rouges ont la valeur* 1 *et les arcs noirs ont la valeur* 0. *Le circuit de gauche a une hauteur égale à trois et une trame égale à un. Le circuit de droite a une hauteur égale à quatre et une trame égale à zéro.*

algorithme devant être exécuté par un système d'exploitation, on utilisera un langage de programmation (C, C++, Java,...). Ce tapuscrit ne s'adressant pas à des compilateurs, nous utiliserons un langage appelé *pseudo-code* qui est un langage proche du langage naturel.

Comme exemple, on peut construire un algorithme répondant au problème PROMENADE DES PONTS. Pour une instance issue d'un exemple réel, on peut supposer que pour chaque sommet u, les sommets dans $N^*(u) \cup \{u\}$ forment une clique (car toutes les extrémités de ponts sont situées sur la même île). On appellera un tel graphe un *graphe réaliste des ponts*. Pour un graphe réaliste des ponts, on peut répondre au problème PROMENADE DES PONTS en exécutant l'algorithme suivant. Nous ne ferons

Algorithme 1 : Algorithme répondant à PROMENADE DES PONTS pour les instances réalistes.

Entrée : Un graphe réaliste des ponts G.
Sortie : VRAI si G est une instance positive de PROMENADE DES PONTS, FAUX sinon.

```
1 Pour tout u ∈ V(G) faire
2     Si |N*(u)| est pair alors
3         retourner FAUX
4 retourner VRAI
```

pas la preuve formelle de l'exactitude de cet algorithme, mais l'idée générale est que chaque fois que nous entrons sur une île, il faut pouvoir en sortir et donc le nombre de

ponts sur chaque île doit être pair (et donc le voisinage de chaque extrémité de pont doit être impair).

Soit A un algorithme. Comme un algorithme prend en entrée une instance et donne en sortie une solution, il peut être vu comme une application. On peut utiliser la même notation : ainsi $A(I)$ désignera la solution retournée par l'algorithme A pour l'instance I. De même, on dira qu'une application $f : D \mapsto S$ est *calculable* s'il existe un algorithme A tel que $\forall I \in D, f(I) = A(I)$

1.3.2 Complexité

Il y a plusieurs critères pour évaluer la performance d'un algorithme : vitesse d'exécution moyenne, vitesse d'exécution dans le pire des cas, espace nécessaire en mémoire, possibilité de parallélisation... Dans cette thèse, nous mesurerons la performance des algorithmes selon le critère de la vitesse d'exécution dans le pire des cas. Celle-ci dépend en partie du matériel utilisé. Ainsi pour s'affranchir des considérations technologiques, il est plus pertinent de regarder le nombre d'opérations élémentaires à effectuer en fonction de la taille de l'entrée : c'est ce qu'on appelle la *complexité temporelle*.

Pour s'entendre sur le nombre d'opérations élémentaires d'un algorithme, on prend le nombre d'étapes nécessaires à l'exécution lorsque l'algorithme est simulé sur une *machine de Turing*. Une machine de Turing est un objet théorique permettant de suivre des procédures algorithmiques. Elle est composée d'un ruban de taille infinie, décomposé en cases dans lesquelles sont inscrits des symboles. Sur ce ruban une tête de lecture/écriture est posée. Elle contient également un registre d'états permettant de mémoriser l'état de la machine et une table d'actions, qui en fonction de l'état de la machine et du symbole lu par la tête, indique quel symbole écrire dans la case courante, le nouvel état de la machine et la direction dans laquelle la tête doit se déplacer. Il existe plusieurs définitions pour une machine de Turing, on peut par exemple donner la définition suivante décrite dans [60].

Définition 2

Une *machine de Turing déterministe* est définie par le 5-uplet $(Q, \Gamma, q_0, \delta, F)$ où

- Q est un ensemble fini d'états ;
- Γ est un ensemble de symboles possibles pour les cases de la bande, contenant notamment un symbole blanc \$;
- $q_i \in Q$ est l'état initial de la machine ;
- $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\leftarrow, \rightarrow\}$ est la fonction de transition ;
- $F \subseteq Q$ est l'ensemble des états terminaux.

Exécuter une machine de Turing revient à appliquer la fonction de transition successivement jusqu'à ce que la machine atteigne un état terminal. La fonction de transition s'applique comme suit. Supposons qu'une machine de Turing T soit dans un état q_i, que la tête de lecture soit située sur la case c_x contenant le symbole S_1 et que

$\delta(q_i, S_1) = (q_j, S_2, \rightarrow)$. Appliquer la fonction de transition signifie, enregistrer l'état q_j dans le registre d'état, écrire le symbole S_2 dans la case c_x et enfin, déplacer la tête de lecture sur la case c_{x+1}. Le nombre d'étapes avant d'atteindre un état terminal correspond tout simplement au nombre de fois que la fonction de transition est appliquée. Notons qu'il existe également des machines de Turing dites *non-déterministes* pour lesquelles la fonction de transition n'est pas une application : dans ce cas, pour un état q_i et un symbole S_j, plusieurs possibilités existent pour la fonction de transition et la machine devra faire un choix parmi ces transitions (que l'on supposera toujours le bon).

En exemple, donnons la machine de Turing $T = (Q, \Gamma, q_0, \delta, F)$ suivante. Celle-ci parcourt un mot binaire (*i.e.* sur l'alphabet $\{0, 1\}$) et indique si celui-ci est composé alternativement de 1 et de 0. Elle est composée d'un état initial q_i et de deux états finals q_{vrai} et q_{faux}, qui indiquent si le mot respecte ou non la propriété, respectivement. On utilisera également deux états intermédiaires q_0 et q_1 qui indiquent si le dernier symbole lu est un 0 ou un 1, respectivement. La fonction de transition δ est donnée dans le tableau suivant.

	q_i			q_0			q_1		
	Q	Γ	direction	Q	Γ	direction	Q	Γ	direction
\$	q_{vrai}	–	–	q_{vrai}	–	–	q_{vrai}	–	–
0	q_0	0	$\rightarrow$	q_{faux}	–	–	q_0	0	$\rightarrow$
1	q_1	1	$\rightarrow$	q_1	1	$\rightarrow$	q_{faux}	–	–

Dès que la machine parcourt deux symboles identiques consécutifs, elle se met dans l'état final q_{faux} car la propriété n'est pas respectée. Sinon, si elle atteint un symbole blanc, cela veut dire qu'elle n'a pas rencontré deux symboles identiques consécutifs et donc que la propriété est vraie pour ce mot. Un exemple de déroulement de cette machine est illustré par la figure 1.7.

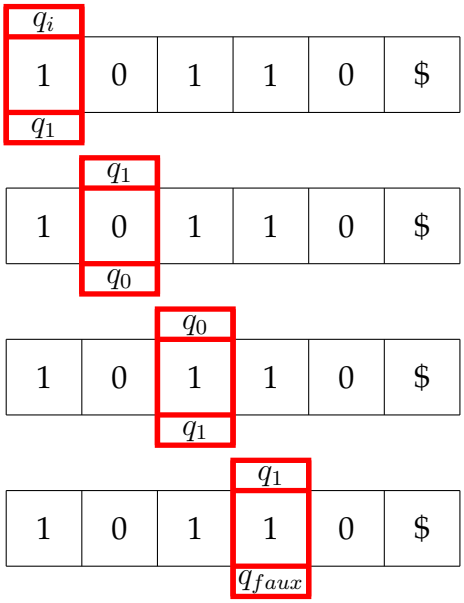


FIGURE 1.7 – *Exemple de déroulement de la machine de Turing donné en exemple sur le mot* 10110. *La tête est symbolisée par le rectangle rouge. L'état au-dessus de la tête est l'état actuel de la machine et l'état en-dessous de la tête est l'état retourné par la fonction de transition.*

Bien sûr, les algorithmes présentés ici ne seront pas décrits par une machine de Turing. Il peut donc être parfois difficile de connaître exactement le nombre d'opérations élémentaires nécessaires pour leur exécution. Pour pallier ce problème, nous utiliserons la notation de Landau $\mathcal{O}$ pour donner la complexité temporelle d'un algorithme. Cette notation est définie comme suit.

Définition 3

Soient $f : \mathbb{N} \mapsto \mathbb{N}$ et $g : \mathbb{N} \mapsto \mathbb{N}$ deux applications. On dit que f est *dominée* par g s'il existe deux constantes K et N telles que :

$$\forall n > N, f(n) < K \cdot g(n).$$

Pour indiquer que f est dominée par g, on utilisera la notation $f = \mathcal{O}(g)$.

Cette notation permet de décrire l'allure générale d'une fonction. Pour désigner la complexité temporelle, on donnera la plus petite fonction dominant le nombre d'opérations élémentaires à l'exécution. On peut donner un exemple, en calculant la complexité de l'algorithme 1. On peut supposer que l'ensemble $N^*(u)$ est fourni avec l'entrée pour chaque sommet u. Dans le pire des cas, on doit parcourir tous les sommets pour exécuter l'instruction à la ligne 2, ce qui fait un total de $|V(G)|$ instructions. On a donc une complexité temporelle de $\mathcal{O}(|V(G)|)$. Par simplicité, on pourra dire qu'un algorithme est polynomial (ou exponentiel, factoriel...) si sa complexité temporelle est polynomiale.

Une autre notation de Landau que l'on pourra utiliser est la notation o.

Définition 4

Soient deux applications $f : \mathbb{N} \mapsto \mathbb{R}^+$ et $g : \mathbb{N} \mapsto \mathbb{R}^+$, f est dite *négligeable* devant g si pour toute constante C, il existe une constante N telle que pour tout

$$n \leq N, \frac{f(n)}{g(n)} < C.$$

Autrement dit, le quotient de f par g tend vers zéro. Pour indiquer que f est négligeable devant g, on utilisera la notation $f = o(g)$.

1.3.3 Temps d'exécution

Pour un usage applicatif, il est important d'avoir la meilleure complexité temporelle possible. Prenons par exemple le problème COUVERTURE PAR SOMMETS. On peut formuler un algorithme un peu naïf consistant à regarder tous les sous-ensembles possibles de sommets du graphe afin de sélectionner la couverture minimum parmi ceux-ci. Cet algorithme est appelé *algorithme par force brute* et il peut être adapté pour résoudre n'importe quel problème. Il est décrit par l'algorithme 2.

La complexité de cet algorithme est égal à $\mathcal{O}(2^{|V(G)|})$. Plaçons-nous d'un point de

Algorithme 2 : Algorithme par force brute pour COUVERTURE PAR SOMMETS.

Entrée : Un graphe G.
Sortie : Une couverture $V' \subseteq V(G)$ des sommets de G de taille minimum.

```
1 V' ← V(G);
2 Pour tout V'' ∈ P(V(G)) faire
3     Si |V''| < |V'| et ∀e ∈ E(G), e ∩ V'' ≠ ∅ alors
4         V' ← V'';
5 retourner V'
```

vue pratique et supposons que cet algorithme soit implémenté sur un système pouvant effectuer une instruction toutes les dix nanosecondes. S'il faut moins d'une seconde pour trouver une solution minimum dans un graphe contenant moins de trente sommets, il faudra en revanche environ trente-six années pour trouver une solution minimum dans un graphe de soixante sommets et plus de 10^5 milliards d'années dans un graphe à cent sommets. À titre de comparaison, l'âge de l'univers est estimé à environ 13,8 milliards d'années [25]. Ce genre d'algorithme n'est donc pas utilisable lorsque la taille des instances devient trop importante. Le tableau 1.1 donne des ordres de grandeur de temps d'exécution pour différentes complexités temporelles. On voit que seules les complexités polynomiales sont utilisables quand la taille de l'entrée devient importante.

TABLE 1.1 – *Comparaison des temps d'exécution pour des complexités temporelles polynomiales (en vert) et exponentielles ou factorielles (en rouge). On suppose que la machine exécutant les algorithmes peut effectuer une instruction élémentaire toutes les dix nanosecondes.*

Complexité temporelle	Adjectif	Temps d'exécution en fonction de n			
		$n = 10$	$n = 30$	$n = 50$	$n = 100$
$\mathcal{O}(1)$	Constante	$10ns$	$10ns$	$10ns$	$10ns$
$\mathcal{O}(\log n)$	Logarithmique	$10ns$	$14ns$	$17ns$	$20ns$
$\mathcal{O}(n)$	Linéaire	$100ns$	$300ns$	$500ns$	$1\mu s$
$\mathcal{O}(n^2)$	Quadratique	$1\mu s$	$9\mu s$	$25\mu s$	$100\mu s$
$\mathcal{O}(n^3)$	Cubique	$10\mu s$	$270\mu s$	$1,25ms$	$10ms$
$\mathcal{O}(n^7)$		$100ms$	$218s$	2 heures	11 jours
$\mathcal{O}(2^n)$	Exponentielle	$10\mu s$	$10s$	130 jours	$10^5 Ga$
$\mathcal{O}(n!)$	Factorielle	$36ms$	$10^7 Ga$	$10^{39} Ga$	$10^{133} Ga$

unités de temps : nanosecondes (ns), millisecondes (ms), secondes (s), heures, jours ou milliard d'année (Ga)

1.4 Classes de complexité

On va regrouper les problèmes qui possèdent des propriétés communes dans leurs résolutions sous la forme d'ensembles appelés *classes de complexité*. Ces classes servent à caractériser la difficulté intrinsèque des problèmes algorithmiques. On va s'intéresser aux classes utilisant trois types d'algorithmes : les algorithmes exacts, les algorithmes approchés et les algorithmes de recherche locale. Là aussi, nous ne ferons pas une pré-

sentation exhaustive de la théorie de la complexité, le lecteur pourra approfondir ses connaissances sur cette théorie en consultant un ouvrage comme celui de Garey et Johnson [38].

Les *algorithmes exacts* sont les algorithmes permettant de répondre aux problèmes de décision ou de donner une solution optimale pour les problèmes d'optimisation. L'appartenance pour un problème à une classe de complexité prenant en considération des algorithmes exacts prendra comme critère la complexité temporelle des algorithmes exacts permettant de résoudre ce problème.

Les *algorithmes approchés* sont des algorithmes permettant de donner une solution qui n'est pas forcément optimale mais qu'on espère proche d'une solution optimale. On a tendance à réserver le terme algorithme approché aux algorithmes pour lesquels on a une garantie de performance sur le score de la solution. Pour les algorithmes qui n'ont pas de telles garanties, on parlera d'*heuristiques*. On exige de la plupart des algorithmes approchés qu'ils aient une complexité temporelle polynomiale. L'appartenance d'un problème à une classe de complexité prenant en considération de tels algorithmes approchés dépendra de la garantie de performance sur les solutions approchées renvoyées par ces algorithmes.

Enfin, les *algorithmes de recherche locale* sont des algorithmes permettant de trouver un optimum local, qui là aussi n'est pas nécessairement optimale. L'appartenance pour un problème de recherche locale à une classe de complexité dépendra du nombre de solutions explorées avant d'obtenir un optimum local.

Notons qu'il existe des classes de complexité prenant en compte d'autres critères, par exemple la complexité en espace mémoire. Dans tout ce qui suit, le terme « complexité », utilisé seul, désignera toujours la complexité temporelle. La suite de cette section est dévolue à la présentation de quelques classes de complexité qui seront utilisées dans le chapitre 3 et le chapitre 4. On utilisera pour cela des définitions générales pouvant impliquer des problèmes abstraits Π. Σ_Π désigne l'ensemble des instances possibles pour le problème Π et pour une instance $I \in \Sigma_\Pi$, on note $|I|$ la taille de I, qui correspond au nombre de variables élémentaires utilisées pour encoder cette instance.

1.4.1 Généralités

Les classes de complexité regroupent les problèmes en fonction de la difficulté de les résoudre en respectant une certaine propriété τ. Cette propriété peut correspondre au nombre minimum d'opérations nécessaires, à l'espace mémoire minimum nécessaire ou à la distance des solutions fournies avec une solution optimale… Les *réductions* sont des applications permettant de montrer qu'un problème est plus difficile qu'un autre selon cette propriété. De façon très générale, elles prennent généralement la forme suivante.

Définition 5

Soient Π_1 et Π_2 deux problèmes et une propriété τ. Π_1 est réductible à Π_2 en préservant la propriété τ s'il existe une application $f : \Sigma_{\Pi_1} \mapsto \Sigma_{\Pi_2}$ telle que pour toute instance $I \in \Sigma_{\Pi_1}$, si $f(I)$ est solvable en respectant la propriété τ, alors I est solvable en respectant τ.

Le principe d'une réduction est donc de construire une instance de Π_2 pour chaque instance de Π_1. Si l'on arrive à calculer une solution en préservant la propriété τ dans l'instance de Π_2 créée, alors on arrive également à le faire dans l'instance de Π_1. L'idée sous-jacente est que, si l'on sait (ou suspecte) que Π_1 ne peut pas être résolu en respectant τ, alors la résolution des instances de Π_2 ne pourra pas non plus se faire en respectant τ.

On peut par exemple utiliser la notation $\Pi_1 \preccurlyeq_\tau \Pi_2$ pour indiquer que Π_1 se réduit à Π_2 en préservant la propriété τ. Une réduction peut être vue comme une relation entre les problèmes. Cette relation est *réflexive*, c'est-à-dire que pour un problème Π, on a $\Pi \preccurlyeq_\tau \Pi$. Cette relation est également *transitive* : si l'on a trois problèmes Π_1, Π_2 et Π_3 tels que $\Pi_1 \preccurlyeq_\tau \Pi_2 \wedge \Pi_2 \preccurlyeq_\tau \Pi_3$ alors on a $\Pi_1 \preccurlyeq_\tau \Pi_3$. Cela permet d'établir un *préordre partiel* entre les problèmes. Il s'agit d'un préordre et non d'un ordre car pour deux problèmes Π_1 et Π_2, on peut avoir $\Pi_1 \preccurlyeq_\tau \Pi_2 \wedge \Pi_2 \preccurlyeq_\tau \Pi_1$. Autrement dit, les deux problèmes sont aussi difficiles selon la propriété τ. Ce préordre est « partiel » car deux problèmes ne sont pas toujours comparables selon cette relation.

En prenant ce préordre partiel, pour une classe de complexité C, on peut désigner l'ensemble des problèmes de C qui sont les plus difficiles suivant la propriété τ.

Définition 6

Soit C une classe de complexité. Un problème Π est C-*complet* (pour la réduction $\preccurlyeq_\tau$) si

— $\Pi \in C$;

— pour tout problème $\Pi' \in C$, $\Pi' \preccurlyeq_\tau \Pi$.

Pour montrer qu'un problème est C-complet, il n'est pas nécessaire de construire une réduction à partir de tous les problèmes de C. En effet, comme une réduction est une relation transitive, il suffit de construire une réduction à partir d'un seul problème C-complet.

Propriété 1

Soit C une classe de complexité. Un problème Π est C-*complet* (pour la réduction $\preccurlyeq_\tau$) si

— $\Pi \in C$;

— il existe un problème C-complet Π' tel que $\Pi' \preccurlyeq_\tau \Pi$.

Il faut cependant connaître au préalable un problème C-complet pour pouvoir uti-

liser cette propriété. On peut également désigner l'ensemble des problèmes de C qui sont au moins aussi difficiles que tout problème de C.

Définition 7

Soit C une classe de complexité. Un problème Π est C-*difficile* (pour la réduction $\preccurlyeq_\tau$) si pour tout problème $\Pi' \in C$, $\Pi' \preccurlyeq_\tau \Pi$.

La différence avec les problèmes C-complets est que l'appartenance à la classe C n'est pas obligatoire. De la même façon que pour les problèmes C-complets, pour montrer qu'un problème est C-difficile il suffit de faire une réduction à partir d'un seul problème C-difficile.

Propriété 2

Soit C une classe de complexité. Un problème Π est C-difficile (pour la réduction $\preccurlyeq_\tau$) s'il existe un problème C-difficile Π' tel que $\Pi' \preccurlyeq_\tau \Pi$.

Les notions de complétude et de difficulté sont intéressantes quand elles se définissent autour d'un ensemble de classes appelé *hiérarchie*. Une hiérarchie est un ensemble de classes de complexité $C_1, \ldots, C_n$ tel que

$$C_1 \subseteq C_2 \subseteq \cdots \subseteq C_n.$$

Pour une classe C_i de cette hiérarchie, la réduction utilisée pour définir les problèmes C_i-complets et C_i-difficiles préserve l'appartenance à la classe de complexité C_{i-1}. Autrement dit, une telle réduction assure que si un problème Π_1 se réduit à un problème Π_2, alors si Π_2 appartient à C_{i-1}, Π_1 appartient également à C_{i-1}. Ainsi, si les classes C_{i-1} et C_i sont différentes, aucun problème C_i-complets n'appartient à C_{i-1}.

Nous abordons à présents dans les sous-sections suivantes, des exemples concrets de hiérarchies de classes de complexité faisant intervenir les notions que nous venons de définir.

1.4.2 Algorithmes exacts

Pour les algorithmes exacts, on va regrouper les problèmes en classes selon la complexité minimum nécessaire pour les résoudre.

1.4.2.1 Les classes P et NP

Les classes de complexité P (pour *polynomial*) et NP (pour *non deterministic polynomial* en anglais) sont des classes de problèmes contenant des problèmes de décision. Pour la classe de complexité P, la définition est la suivante.

Définition 8

Le problème de décision Π appartient à la classe P si et seulement si il existe un algorithme polynomial pouvant répondre à Π.

Typiquement P contient les problèmes pouvant être résolus avec un algorithme ayant une complexité $\mathcal{O}(n^k)$ où n est la taille de l'entrée et k est une constante. Plus spécifiquement, on parlera d'une complexité constante si $k = 0$, d'une complexité linéaire si $k = 1$ et d'une complexité quadratique si $k = 2$. Notons que la complexité peut également faire intervenir une fonction logarithmique en la taille de l'entrée. D'un point de vue pratique, ce sont les problèmes qui sont solvables en temps humain. Pour cette raison, on qualifiera ces problèmes comme étant « faciles ». Pour montrer qu'un problème appartient à cette classe, il suffit d'exhiber un algorithme polynomial le résolvant. Ainsi le problème PROMENADE DES PONTS appartient à P si l'on se restreint aux graphes de ponts réalistes. On peut également mettre en avant les algorithmes de tri permettant d'ordonner une liste d'éléments qui ont une complexité de $\mathcal{O}(n \log n)$.

Pour définir la classe NP, nous devons d'abord revenir sur la notion de certificat. Rappelons qu'un certificat est une entrée permettant de prouver qu'une instance est positive. Pour un problème Π et une instance $I \in \Sigma_\Pi$, on dénote par $sol_\Pi(I)$ l'ensemble des certificats pour I ($sol_\Pi = \varnothing$ si I est une instance négative). La définition de la classe NP fait intervenir la notion de certificat.

Définition 9

Le problème de décision Π appartient à la classe NP si et seulement si il existe un algorithme polynomial permettant de déterminer toute instance $I \in \Sigma_\Pi$ et pour toute entrée E si $E \in sol_\Pi(I)$.

Par exemple, les problèmes de SATISFAISABILITÉ appartiennent à NP : un certificat est tout simplement une assignation des variables composant la formule booléenne. Pour vérifier que le certificat satisfait cette formule booléenne, il suffit de parcourir chacune des clauses et de vérifier si au moins un de ces littéraux est satisfait par β. Cette vérification se fait en temps linéaire et donc SATISFAISABILITÉ appartient bien à NP. Une autre définition de la classe NP est que les problèmes qui la composent peuvent être résolus en temps polynomial par une machine de Turing non-déterministe. Autrement dit, on peut résoudre un problème dans NP en faisant un nombre polynomial de choix au hasard.

De façon évidente, on peut voir que tous les problèmes appartenant à P appartiennent également à NP, et donc P $\subseteq$ NP. En revanche, on ne sait pas si tous les problèmes appartenant à NP appartiennent à P. Une hypothèse formulée par Stephen Cook [26] est que ces deux classes de complexité sont différentes et cela reste une question ouverte aujourd'hui.

Hypothèse

P $\neq$ NP

Il s'agit en vérité de la question fondamentale en informatique. Si l'hypothèse est fausse, cela implique qu'il est aussi facile de vérifier une solution que de produire une solution satisfaisant un problème. Le consensus scientifique autour de cette question admet que cette hypothèse est vraie et la plupart des travaux présentés dans cette thèse n'ont d'intérêt que si c'est réellement le cas.

1.4.2.2 Complétude et difficulté de la classe NP

Pour la complétude et la difficulté de la classe NP, on prendra en compte la possibilité de résolution en temps polynomial. Ainsi, un problème Π_2 est plus difficile qu'un autre problème Π_1 si l'existence d'un algorithme polynomial pour Π_2 implique l'existence d'un algorithme polynomial pour Π_1. Si l'on suspecte que l'hypothèse $\mathrm{P} \neq \mathrm{NP}$ est vraie, c'est qu'il existe un certain nombre de problèmes dans NP pour lesquels on n'arrive pas à trouver d'algorithmes polynomiaux pour les résoudre et pour lesquels on suspecte qu'il n'en existe pas (sans pour autant que cela ne soit démontré, sinon la question ne serait plus ouverte). Les problèmes NP-difficiles ne peuvent pas, avec cette hypothèse, être résolus en temps polynomial. Pour montrer qu'un problème est au moins aussi difficile qu'un autre on utilise une *réduction polynomiale*, appelée également réduction de Karp.

Définition 10

Soient Π_1 et Π_2 deux problèmes de décision. Π_1 est *réductible polynomialement* à Π_2 s'il existe une application $f : \Sigma_{\Pi_1} \mapsto \Sigma_{\Pi_2}$ telle que pour toute instance $I \in \Sigma_{\Pi_1}$:

- $f(I)$ est calculable en temps polynomial ;
- $f(I)$ est une instance positive de Π_2 si et seulement si I est une instance positive de Π_1.

Pour indiquer que Π_1 est réductible polynomialement à Π_2 on utilisera la notation $\Pi_1 \preccurlyeq \Pi_2$.

Grâce à cette réduction, on peut définir les classes de problèmes NP-complets et NP-difficiles. Historiquement, le premier problème a avoir été montré NP-complet est le problème SATISFAISABILITÉ [26]. La plupart des restrictions que l'on peut imposer sur les formules booléennes données en entrée laissent le problème NP-complet. On peut par exemple citer le problème 3-SATISFAISABILITÉ MONOTONE PLANAIRE qui est NP-complet [32]. En revanche pour le problème 2-SATISFAISABILITÉ, il existe un algorithme polynomial [9].

Pour les problèmes NP-difficiles, on peut également ajouter des problèmes d'optimisation. Il faudra au préalable adapter la définition 10 : ainsi Π_1 et Π_2 pourront désigner des problèmes d'optimisation et la réduction polynomiale se fera en prenant en compte leurs problèmes de décision associés. Tous les problèmes présentés dans la section 1.2 sont NP-difficiles. Notons que dans la littérature, on trouve une classe spécifique pour les problèmes d'optimisation : il s'agit de la classe NPO. Un problème d'optimisation appartient à la classe NPO si son problème de décision associé appartient à NP. Cette thèse ne traitant pas directement de complexité, nous ne ferons pas

de distinction entre ces deux classes.

Donnons un exemple simple de réduction polynomiale. Nous allons montrer que PROMENADE DES PONTS est NP-complet dans le cas général. Pour cela, nous allons créer une réduction à partir du problème PLUS LONG CYCLE. À partir d'un graphe orienté nous, allons créer une instance de PROMENADE DES PONTS à l'aide de la construction suivante.

Construction 1

À partir d'un graphe orienté G, nous produisons le graphe G', muni du couplage M^* de la façon suivante :

— pour chaque sommet $v_i \in V(G)$, construire une arête de couplage $v_i^{in}v_i^{out}$;

— pour chaque arc $(v_i, v_j) \in A(G)$, ajouter l'arête $v_i^{out}v_j^{in}$ avec $w(v_i^{out}v_j^{in}) = c(v_i, v_j)$.

L'idée de la construction est simplement de remplacer chaque sommet dans le graphe de départ par une arête de couplage. À l'aide de cette construction, nous pouvons montrer le résultat suivant

Théorème 1

PROMENADE DES PONTS est NP-complet.

Preuve.

Tout d'abord, pour vérifier qu'un sous-graphe correspond bien à une solution, il suffit de regarder s'il est connexe, 2-régulier que son nombre d'arêtes de couplage est égal à la taille du couplage du graphe. Ces trois opérations peuvent se faire en temps polynomial et donc PROMENADE DES PONTS est dans NP. Il reste à montrer qu'il est NP-difficile. Soit G un graphe orienté et soit G' son graphe produit par la construction 1. Montrons que G contient un cycle de longueur $|V(G)|$ si et seulement si G' est une instance positive de PROMENADE DES PONTS.

$\Rightarrow$ Soit $C = \{v_0, \ldots, v_n\}$ un cycle de longueur $|V(G)|$ dans G. Le cycle alterné $M^* \cup \{v_i^{out}v_{(i+1 \mod n)}^{in} \mid v_i \in C\}$ est une solution pour PROMENADE DES PONTS dans G'.

$\Leftarrow$ Soit C' une solution de PROMENADE DES PONTS dans G'. Le cycle contenant les arêtes $\{v_iv_j \mid v_i^{out}v_j^{in} \in E(C') \setminus M^*\}$ est un cycle de longueur $|V(G)|$ dans le graphe G.

Le graphe G' peut être produit en temps polynomial. Ainsi, on a montré que la construction 1 est une réduction polynomiale et donc PROMENADE DES PONTS est NP-complet □

Notons à titre informatif qu'il existe des problèmes qui ne sont à priori ni dans P, ni NP-complets. Cela a été montré par Ladner [57] sous l'hypothèse P $\neq$ NP. On appelle de tels problèmes des problèmes NP-*intermédiaires*. La figure 1.8 montre les inclusions des différentes classes présentées jusqu'à présent.

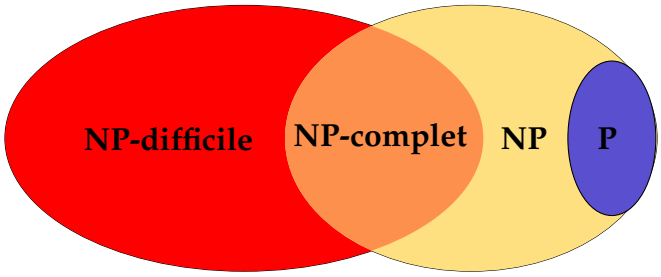


FIGURE 1.8 – *Diagramme de Venn illustrant les inclusions des classes de complexité.*

1.4.2.3 Complexité paramétrée

Quand un problème n'appartient pas à la classe P, il est nécessaire d'utiliser un algorithme de complexité au moins exponentielle pour le résoudre de façon exacte. Cependant, comme vu précédemment dans le tableau 1.1, si la complexité d'un algorithme est exponentielle ou factorielle en la taille de l'instance donnée en entrée, il ne sera pas possible d'utiliser cet algorithme sur des instances de taille importante. Il faut donc trouver d'autres méthodes pour pouvoir résoudre des problèmes NP-difficiles.

L'une d'entre elles est la *complexité paramétrée*. Dans ce type d'approche, on essaie de construire un algorithme pour lequel le facteur exponentiel ne va pas dépendre directement de la taille de l'instance mais d'un paramètre qu'on espère petit. Un *problème paramétré* (Π, κ) est un couple composé d'un problème Π et d'une fonction calculable $\kappa : \Sigma_\Pi \mapsto \mathbb{N}$ appelée *paramétrisation* de Π. La classe de complexité FPT (pour *fixed-parameter tractable*) est une classe définie sur les problèmes paramétrés.

Définition 11

Le problème paramétré (Π, κ) appartient à la classe FPT si et seulement si il existe une application $f : \mathbb{N} \mapsto \mathbb{N}$ telle que Π peut être résolu par un algorithme de complexité $\mathcal{O}(f(\kappa(I)) \cdot |I|^{o(1)})$ où $I \in \Sigma_\Pi$ est l'instance donnée en entrée du problème.

On appelle un tel algorithme un algorithme FPT. On peut remarquer que si la fonction de paramétrisation est constante, alors le problème peut être résolu en temps polynomial. Cette fonction de paramétrisation peut être soit structurelle au problème soit structurelle à l'instance donnée en entrée.

Illustrons cela avec le problème COUVERTURE PAR SOMMETS. Si l'on cherche une paramétrisation structurelle au problème, on peut regarder la taille k de la couverture dans le problème de décision associé. Au lieu de regarder tous les sous-ensembles de sommets comme dans l'algorithme brut, on regarde seulement ceux de cardinalité k. Cela permet d'avoir un algorithme de complexité de $\mathcal{O}(|V(G)|^k)$, ce qui est bien un algorithme FPT (en prenant $c = 0$). Actuellement, le meilleur algorithme FPT connu avec ce paramètre a une complexité de $\mathcal{O}(1.2738^k + k|V(G)|)$ [54]. Si on cherche une paramétrisation structurelle à l'instance donnée en entrée, on peut regarder la largeur arborescente tw du graphe. Sans rentrer dans les détails, si l'on possède une décomposition arborescente T du graphe, une idée pour résoudre COUVERTURE PAR SOMMETS peut être d'appliquer l'algorithme de brute force uniquement dans chacun des sacs qui composent T et non dans l'intégralité du graphe et d'assembler ensuite ces sous-

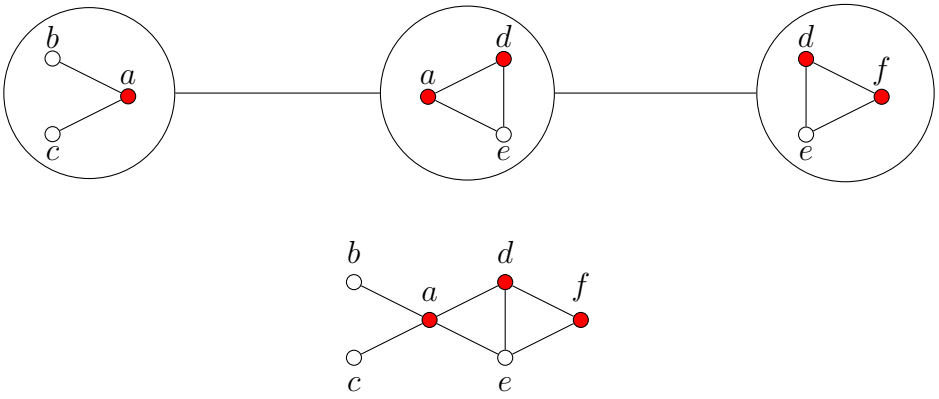


FIGURE 1.9 – *Illustration de la résolution de* COUVERTURE PAR SOMMETS *au moyen d'une décomposition arborescente. On peut calculer les couvertures (non forcément minimums) dans les graphes induits pour chacun des sacs de la décomposition puis considérer toutes les unions possibles et sélectionner la couverture minimum parmi ces unions.*

ensembles pour construire une couverture du graphe (voir la figure 1.9). Cela permet d'obtenir un algorithme de complexité de $\mathcal{O}(2^{tw} \cdot |V(G)|)$ qui est un algorithme FPT. Il existe bien sûr d'autres paramètres que l'on pourrait prendre en compte comme le degré maximum du graphe, la densité...

La décomposition arborescente est une technique assez classique pour trouver un algorithme FPT. Cependant, il est à noter que calculer une décomposition arborescente de largeur minimum est un problème NP-difficile.

1.4.2.4 Hiérarchie W

Il existe une collection de classes de complexité contenant des problèmes paramétrés. Il s'agit de la hiérarchie W (pour *weft* en anglais, par rapport à la trame du circuit booléen) qui a été définie par Downey et Fellows [36]. Dans cette hiérarchie, pour les notions de complétude et de difficulté, on va utiliser une *FPT-réduction*.

Définition 12

Soient (Π_1, κ_1) et (Π_2, κ_2) deux problèmes paramétrés où Π_1 et Π_2 sont des problèmes de décision. (Π_1, κ_1) est *FPT-réductible* à (Π_2, κ_2) s'il existe une application $f : \Sigma_{\Pi_1} \mapsto \Sigma_{\Pi_2}$ telle que pour toute instance $I \in \Sigma_\Pi$:

- il existe une application $g : \mathbb{N} \mapsto \mathbb{N}$ telle que l'instance $f(I)$ est calculable en temps $\mathcal{O}(g(\kappa_1(I)) \cdot |I|^{\mathcal{O}(1)})$;
- I est une instance positive de Π_1 si et seulement si $f(I)$ est une instance positive de Π_2 ;
- il existe une application calculable $p : \mathbb{N} \mapsto \mathbb{N}$ telle que $\kappa_2(f(I)) \leq p(\kappa_1(I))$.

Notons que les fonctions p et g ne dépendent pas de l'instance I mais juste de la valeur du paramètre. Cette réduction préserve l'appartenance à la classe FPT : cela veut dire que si (Π_1, κ_1) se FPT-réduit à (Π_2, κ_2) et que le problème (Π_2, κ_2) appartient à

la classe FPT, alors le problème (Π_1, κ_1) appartient également à FPT. De la même façon que pour la réduction polynomiale, on peut également utiliser la réduction précédente sur des problèmes paramétrés (Π, κ) où Π est un problème d'optimisation, en prenant une version de décision de Π.

La hiérarchie W contient un ensemble de classes $W[k]$, pour tout entier $k \geq 0$. Un problème paramétré (Π, κ) appartient à la classe $W[k]$ s'il existe une constante t telle qu'il est possible de FPT-réduire (Π, κ) au problème suivant.

(k,t)-CIRCUIT BOOLÉEN PONDÉRÉ

Entrée : Un circuit booléen C de trame au plus égal à k et de longueur au plus égale à t.
Sortie : Trouver une assignation β contenant au plus k variables assignées à VRAI et qui satisfait C.

Le paramètre associé au circuit booléen correspond alors à la trame du circuit. Il existe également une classe contenant l'union de toutes les classes $W[k]$ qui est appelée $W[P]$ (pour polynomial). On a donc un ensemble de classes qui dessinent une hiérarchie, à savoir :

$$W[0] \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[P].$$

La classe $W[0]$ correspond à la classe FPT. On a également un lot d'hypothèses associées à cette hiérarchie W. On suppose que pour tout k, on a $W[k] \neq W[k+1]$. Si jamais on a FPT $\neq W[1]$, cela implique que P $\neq$ NP. En revanche, si FPT $= W[1]$, cela n'implique rien sur le reste de la hiérarchie W : on pourrait très bien avoir FPT $= W[1]$ et $W[1] \neq W[2]$.

Parmi les problèmes présentés précédemment, ENSEMBLE INDÉPENDANT paramétré par la taille du stable est $W[1]$-complet. Pour toute constante $c \geq 2$, le problème de décision associé à MAXIMUM c-SAT PONDÉRÉ et paramétré par le poids de l'assignation est également $W[1]$-complet [19].

1.4.2.5 Hypothèse ETH

Pour les problèmes paramétrés, il existe une hypothèse appelé ETH (pour *Exponential time hypothesis* en anglais). Une complexité est dite *sous-exponentielle* si elle est égale à $\mathcal{O}(2^{o(n)})$ pour un paramètre n. L'hypothèse ETH affirme que 3-SATISFAISABILITÉ ne peut pas être résolu en temps sous-exponentiel. Elle a été formulée par Impagliazzo et Paturi [45] de la façon suivante.

Hypothèse – ETH

Le problème 3-SATISFAISABILITÉ ne peut pas être résolu en temps $\mathcal{O}(2^{o(n)}(n+m)^{\mathcal{O}(1)})$ où n correspond au nombre de variables et m correspond au nombre de clauses.

Le meilleur algorithme connu pour résoudre 3-SATISFAISABILITÉ a une complexité égale à $\mathcal{O}(1.30704^n)$ [41]. Cette hypothèse est plus forte que l'hypothèse P $\neq$ NP : en effet, si P $=$ NP, cela implique que ETH est fausse mais la réciproque n'est pas vraie. Dans la hiérarchie W, on a aussi si FPT $= W[1]$ est vraie, alors l'hypothèse ETH est fausse.

Une FPT-réduction pour laquelle la fonction p serait linéaire préserve l'existence d'un algorithme de complexité sous-exponentielle : ainsi si l'on fait une telle FPT-réduction en partant d'un problème paramétré (Π_1, κ_1) ne pouvant pas être résolu de façon sous-exponentielle vers un problème paramétré (Π_2, κ_2), cela signifie que (Π_2, κ_2) ne peut pas être résolu de façon sous-exponentielle. À l'aide de cette réduction, on peut donc montrer que si l'hypothèse ETH est vraie, alors un certain nombre de problèmes NP-complets ne peuvent pas être résolus de façon sous-exponentielle. On a notamment les problèmes ENSEMBLE INDÉPENDANT et COUVERTURE PAR SOMMETS qui ne peuvent pas être résolus en temps $\mathcal{O}(2^{o(n)})$ où n correspond au nombre de sommets dans le graphe [46].

Notons qu'il existe une notion de difficulté et de complétude relative à l'hypothèse ETH mais que nous n'aborderons pas ici. Elle concerne la classe SUBEPT (pour *sub-exponential time solvable* en anglais) qui contient les problèmes pouvant se résoudre de façon sous-exponentielle.

1.4.3 Algorithmes approchés

1.4.3.1 Les classes PTAS et APX

Une autre approche pour résoudre les problèmes NP-difficiles est de construire des algorithmes approchés (aussi appelés algorithmes d'approximation). Comme dit précédemment, un algorithme approché est un algorithme qui retourne une bonne solution mais qui n'est pas forcément optimale. Pour une instance I d'un problème Π et une solution x pour l'instance I, on note $val(x)$ la valeur du critère à optimiser dans la solution x et $OPT(I)$ la valeur de ce critère dans une solution optimale pour l'instance I. On suppose que pour toute solution x, $val(x) \geq 0$. Pour mesurer la performance d'un algorithme d'approximation, on regarde la distance entre la valeur de la solution retournée par celui-ci et une solution optimale dans le pire des cas.

Définition 13

Soit A un algorithme permettant de calculer une solution pour un problème d'optimisation Π. L'application $\rho : \Sigma_\Pi \mapsto [1; +\infty[$ est un facteur d'approximation de A si

$$\forall I \in \Sigma_\Pi, max\left\{\frac{OPT(I)}{val(A(I))}; \frac{val(A(I))}{OPT(I)}\right\} \leq \rho(I) \cdot OPT(I).$$

Dans ce cas, on dit que A est un algorithme ρ-*approché*.

La définition précédente n'est valable que si les valeurs de toutes les solutions pour un problème sont exclusivement négatives ou exclusivement positives. Si ce n'est pas le cas, il existe dans la littérature un autre facteur d'approximation appelé *facteur d'approximation différentiel* où le ratio prend en compte la différence de la solution renvoyée par l'algorithme avec la solution optimale. Nous n'utiliserons cependant pas ce facteur dans ce document.

L'idée sous-jacente derrière ce facteur d'approximation est que, plus ce facteur est proche de un, meilleure sera la qualité de la solution produite par l'algorithme approché. Les algorithmes d'approximation permettent de construire en temps humain des solutions pour des problèmes NP-difficiles. Les classes d'approximation PTAS (pour *polynomial-time approximation scheme*) et APX (pour *approximation*) regroupent les problèmes selon le critère des facteurs d'approximation possibles pour ces problèmes. Formellement, on utilise les définitions suivantes.

Définition 14

Soit Π un problème d'optimisation :

- Π appartient à la classe PTAS si et seulement si pour tout $\epsilon > 0$, Π admet un algorithme approché polynomial ayant un facteur d'approximation égal à $1 + \epsilon$;
- Π appartient à la classe APX si et seulement si Π admet un algorithme approché polynomial ayant un facteur d'approximation constant ;
- Π appartient à la classe $\log$-APX si et seulement si Π admet un algorithme approché polynomial ayant un facteur d'approximation logarithmique ;
- Π appartient à la classe $poly$-APX si et seulement si Π admet un algorithme approché polynomial ayant un facteur d'approximation polynomial ;

Si un problème peut être approché avec un certain facteur d'approximation, alors il peut nécessairement être approché par un algorithme d'approximation « moins bon ». Ainsi, la définition précédente permet de dessiner une hiérarchie parmi ces classes à savoir PTAS $\subseteq$ APX $\subseteq$ $\log$-APX $\subseteq$ $poly$-APX.

En exemple, considérons l'algorithme suivant pour le problème COUVERTURE PAR SOMMETS. On peut montrer que cet algorithme possède un facteur d'approximation constant.

Algorithme 3 : Algorithme d'approximation pour COUVERTURE PAR SOMMETS.

Entrée : Un graphe G
Sortie : Une couverture $V' \subseteq V(G)$ des sommets

```
1 V' ← ∅ ;
2 Tant que il existe une arête uv telle que uv ∩ V' = ∅ faire
3 |  V' ← V' ∪ {u, v};
4 retourner V'
```

Théorème 2

L'algorithme 3 est un algorithme 2-approché pour COUVERTURE PAR SOMMETS.

Preuve.

Soit V_{opt} une couverture optimale du graphe G. Soit $u_i v_i$ l'arête considérée à l'étape i de la boucle **Tant que** de l'algorithme. On a nécessairement soit $u_i \in V_{opt}$, soit $v_i \in V_{opt}$ sinon l'arête $u_i v_i$ ne serait pas couverte par V_{opt}. Ainsi, on peut faire correspondre chaque paire de sommets ajoutée dans V' à la ligne V' avec un unique sommet de V_{opt} et donc $|V'| \leq 2|V_{opt}|$. □

Ce théorème suffit à montrer que COUVERTURE PAR SOMMETS appartient à la classe d'approximation APX.

1.4.3.2 Complétude et difficulté des classes d'approximation

Pour les notions de complétude et de difficulté des classes d'approximation, on va utiliser des réductions qui préservent le facteur d'approximation. La plus naturelle est appelée la *réduction stricte*, autrement appelée S-*réduction*, qui est définie comme suit.

Définition 15

Soient Π_1 et Π_2 deux problèmes d'optimisation. Π_1 est S-réductible à Π_2 s'il existe une application $f : \Sigma_{\Pi_1} \mapsto \Sigma_{\Pi_2}$ et une constante $\alpha \geq 1$ telles que pour toute instance $I \in \Sigma_{\Pi_1}$:

- $f(I)$ est calculable en temps polynomial ;
- il existe une application polynomiale $g : sol(f(I)) \mapsto sol(I)$ telle que pour toute solution $x \in sol(f(I)), val(x) \leq \alpha \cdot val(g(x))$.

Pour indiquer que Π_1 est S-réductible à Π_2 on utilisera la notation $\Pi_1 \preccurlyeq_S \Pi_2$.

Il est important de noter qu'une S-réduction ne préserve le facteur d'approximation que si Π_1 et Π_2 sont tous les deux des problèmes de maximisation ou tous les deux des problèmes de minimisation. Soit A un algorithme ρ_A-approché pour Π_2 et tel que $g \circ A$ est ρ'_A approché pour Π_1. On a $\rho'_A \leq \alpha \cdot \rho_A$. Soient ρ_1 et ρ_2 les bornes inférieures d'approximation de Π_1 et Π_2, respectivement. On a clairement $\rho_2 = \mathcal{O}(\rho_1)$ et donc si Π_2 appartient à une classe d'approximation C_{app}, Π_1 appartient aussi à C_{app}.

Une autre réduction préservant les facteurs d'approximation pour la classe PTAS est la *réduction linéaire*, autrement appelée L-réduction.

Définition 16

Soient Π_1 et Π_2 deux problèmes d'optimisation. Π_1 est L-réductible à Π_2 s'il existe une application $f : \Sigma_{\Pi_1} \mapsto \Sigma_{\Pi_2}$ et deux constantes $\alpha_1 \geq 1$ et $\alpha_2 \geq 1$ telles que pour toute instance $I \in \Sigma_{\Pi_1}$:

— $f(I)$ est calculable en temps polynomial ;
— $OPT(f(I)) \leq \alpha_1 \cdot OPT(I)$;
— il existe une application polynomiale $g : sol(f(I)) \mapsto sol(I)$ telle que pour toute solution $x \in sol(f(I)), |OPT(I) - val(g(x))| \leq \alpha_2 \cdot |OPT(f(I)) - val(x)|$.

Pour indiquer que Π_1 est L-réductible à Π_2 on utilisera la notation $\Pi_1 \preccurlyeq_L \Pi_2$.

Cette réduction a été introduite par Papadimitriou et Yannakakis [70] et permet notamment de préserver l'appartenance à la classe PTAS. Contrairement à la S-réduction, la L-réduction peut préserver le facteur d'approximation même si les deux problèmes ne sont pas tous les deux des problèmes de maximisation ou tous les deux des problèmes de minimisation.

Sous l'hypothèse $P \neq$ NP, dans la hiérarchie d'approximation présentée précédemment, les inclusions sont strictes. Ainsi cela signifie que pour deux classes d'approximation C_1 et C_2 telles que $C_1 \subset C_2$, un problème Π qui est C_2-complet ou C_2-difficile n'appartient pas à C_1. Une information importante à en retirer est que, sous cette hypothèse, il existe une valeur minimum pour laquelle on ne peut pas approximer Π en temps polynomial. Par exemple pour un problème Π qui est APX-complet, il existe une constante c telle que pour tout $c' < c$, il n'existe pas d'algorithme polynomial c'-approché pour Π. Cette valeur n'est la plupart du temps pas connue précisément.

Pour certains problèmes, on sait en revanche qu'elle est située entre deux valeurs, appelées borne inférieure et supérieure. La borne inférieure est généralement obtenue en faisant une réduction à partir d'un autre problème. La meilleure borne inférieure connue peut être appelée *valeur d'inapproximation*. En général, on ne cherche à déterminer précisément la valeur d'inapproximation que pour les problèmes APX-complets. La borne supérieure correspond quant à elle au meilleur facteur d'approximation connu. Lorsque ces deux bornes sont égales, cela veut dire que la meilleure valeur pour laquelle il est possible d'approximer le problème en temps polynomial est connue.

Pour donner un exemple de réduction, on peut reprendre la construction 1 pour faire un S-réduction de PLUS LONG CHEMIN vers PROMENADE DES PONTS MAXIMUM. En effet, en utilisant les même arguments que ceux donnés pour montrer que PROMENADE DES PONTS est NP-complet, on peut montrer qu'il existe un chemin de longueur k dans un graphe orienté G si et seulement si il existe un chemin alterné de longueur $2k+1$ dans son graphe produit par la construction 1. Ainsi, on aura montré qu'il existe une S-réduction entre ces deux problèmes en prenant $\alpha \geq 3$. On peut donc transférer la borne inférieure d'approximation de PLUS LONG CHEMIN vers PROMENADE DES PONTS MAXIMUM. En l'occurrence, il a été montré que PLUS LONG CHEMIN ne pouvait être approximé en temps polynomial avec un facteur d'approximation meilleur que $n^{1-\epsilon}$ où n est le nombre de sommets, pour tout $\epsilon > 0$ [14].

Notons que la plupart des bornes inférieures d'approximation sont données sous l'hypothèse P $\neq$ NP. Il existe une autre hypothèse formulée par Khot [49] qui est appelée UGC (pour *unique game conjecture*). Sans rentrer dans les détails, cette conjecture permet d'obtenir de meilleures bornes inférieures pour certains problèmes d'approxi-

mation. Par exemple, il a été montré que sous l'hypothèse UGC, la borne inférieure pour COUVERTURE PAR SOMMETS est égale à $2 - \epsilon$. Ainsi, si cette hypothèse est vérifiée, cela veut dire que l'on connaît le meilleur algorithme d'approximation possible pour ce problème.

1.4.4 Recherche locale

1.4.4.1 Optimum local

Le dernier type de problèmes que nous aborderons est celui des problèmes de recherche locale. Un problème de recherche locale est défini comme suit.

Définition 17

Un problème de recherche locale est un 2-uplet $(\Pi, \mathcal{N})$ où

— Π est un problème d'optimisation ;

— $\mathcal{N} : \Sigma_\Pi \times sol \mapsto \mathcal{P}(sol)$ est une application, qui étant donné une instance $I \in \Sigma_\Pi$ et une solution $x \in sol(I)$, construit un ensemble de solutions inclus dans $sol(I)$.

On préférera noter $\Pi/\mathcal{N}$ le problème de recherche locale $(\Pi, \mathcal{N})$. L'application $\mathcal{N}$ est appelée *fonction de voisinage*. Pour une instance I et une solution x, on appelle *voisinage* de x l'ensemble $\mathcal{N}(I, x)$ et une solution $x' \in \mathcal{N}(I, x)$ est une *solution voisine* de x. Notons qu'une solution x' peut être voisine d'une solution x sans que x ne soit une solution voisine de x'. Une solution est un *optimum local* si elle est meilleure que toute solution de son voisinage. On pourra préciser en qualifiant cette solution de *minimum local* si Π est un problème de minimisation ou de *maximum local* si Π est un problème de maximisation. À titre d'exemple, on peut exprimer une fonction de voisinage pour les problèmes de SATISFAISABILITÉ, appelée *Flip*.

Définition 18

Soit φ une instance SATISFAISABILITÉ et une assignation β des variables booléennes de φ. Une assignation β' voisine de β selon la fonction de voisinage Flip est construite en changeant la valeur d'exactement une variable booléenne.

Considérons l'algorithme standard de recherche locale défini par l'algorithme 4.

Cet algorithme permet de trouver un optimum local en partant d'une solution initiale. Si la solution x donnée en entrée est un optimum local, alors elle est retournée. Sinon, on prend la meilleure solution parmi son voisinage et on fait un appel récursif sur celle-ci. Pour un algorithme de recherche local, on va mesurer le nombre d'appels récursifs à effectuer dans le pire des cas. D'un point de vue applicatif, la recherche locale peut être utile pour améliorer une solution construite à l'aide d'un algorithme d'approximation. De ce point de vue, il n'est pas forcément nécessaire de minimiser le nombre d'appels récursifs avant de trouver un optimum local. En effet, si l'on trouve

Algorithme 4 : Algorithme de recherche local standard.

Entrée : Un problème de recherche locale $\Pi/\mathcal{N}$, une instance $I \in \Sigma_\Pi$et une solution $x \in sol(I)$.
Sortie : Un optimum local $x' \in sol(I)$.

```
x' ← la meilleure solution dans N(I, x) ∪ {x} ;
Si x ≠ x' alors
    retourner recherche_locale(Π/N, I, x');
retourner x
```

un optimum local trop rapidement, cela veut dire que la solution initiale n'a pas été grandement améliorée.

À titre d'exemple, faisons une recherche locale sur l'instance de MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ suivante :

$$\underbrace{(\neg x_1 \vee \neg x_1)}_{\omega(C_1)=2} \wedge \underbrace{(\neg x_2 \vee \neg x_2)}_{\omega(C_2)=2} \wedge \underbrace{(\neg x_3 \vee \neg x_3)}_{\omega(C_3)=3} \wedge \underbrace{(x_1 \vee x_3)}_{\omega(C_4)=3} \wedge \underbrace{(x_2 \vee x_3)}_{\omega(C_5)=3} \wedge \underbrace{(\neg x_4 \vee \neg x_4)}_{\omega(C_6)=1}$$

Pour décrire une solution, on peut utiliser un 4-uplet dans $\{1,0\}^4$ où pour chaque variable booléenne x_i, la position i du 4-uplet contient un 1 si l'assignation de la variable booléenne est égale à VRAI et un 0 sinon. On considère une solution initiale $S_{init} = (1,1,1,1)$ où toutes les variables booléennes sont assignées à VRAI. Cette solution S_{init} a un poids égal à six. Déroulons les étapes de l'algorithme de recherche locale.

1. Les solutions voisines de S_{init} sont $(0,1,1,1), (1,0,1,1), (1,1,0,1)$ et $(1,1,1,0)$. La meilleure solution parmi ces solutions voisines est $S_1 = (1,1,0,1)$ qui a un poids de neuf.
2. Les solutions voisines de S_1 sont $(0,1,0,1)$, $(1,0,0,1)$, $(1,1,1,1)$ et $(1,1,0,0)$. La meilleure solution parmi ces solutions voisines est $S_2 = (1,1,0,0)$ qui a un poids de dix.
3. Les solutions voisines de S_2 sont $(0,1,0,0)$, $(1,0,0,0)$, $(1,1,1,0)$ et $(1,1,0,1)$. Parmi ces solutions voisines, aucune n'a un poids plus grand que celui de S_2. On a donc trouvé un minimum local.

Ainsi, en trois appels récursifs de la fonction de recherche locale, on a trouvé un optimum local. Notons que ce n'est cependant pas une solution optimale pour cette instance : en effet la meilleure solution est $(0,0,1,0)$ qui a un poids de onze.

1.4.4.2 La classe PLS

La classe de complexité PLS (pour *polynomial local search* en anglais) contient des problèmes de recherche locale et est définie de la façon suivante.

Définition 19

Un problème de recherche locale $\Pi/\mathcal{N}$ appartient à la classe PLS, s'il existe trois applications polynomiales A_L, B_L et C_L telles que :

— pour une instance $I \in \Sigma_\Pi$, l'application $A_L : \Sigma_\Pi \mapsto sol(I)$ donne une solution initiale pour I ;

— pour toute solution $x \in sol(I)$, l'application $B_L : \Sigma_\Pi \times sol(I) \mapsto \mathbb{R}$ calcule le score de x ;

— pour toute solution x, l'application $C_L : \Sigma_\Pi \times sol(I) \mapsto sol(I)$ détermine si S un optimum local et, si ce n'est pas le cas, retourne la solution avec le meilleur score dans son voisinage.

Dit de façon un peu informelle, les problèmes de recherche locale dans PLS sont ceux où l'on peut construire une première solution initiale en temps polynomial et tels que leurs fonctions de voisinage construisent des voisinages de taille polynomiale.

Par exemple, considérons le problème de recherche local MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ/Flip. On peut construire une fonction donnant une solution initiale en donnant à toutes les variables la valeur VRAI (ou même n'importe quelle valeur au hasard). Pour calculer le score d'une assignation, il suffit de parcourir chacune des clauses et des variables booléennes et de faire les sommes des poids des clauses satisfaites, ce qui se fait en temps polynomial. Enfin, une assignation a exactement autant de solutions voisines que de variables booléennes présentes dans la formule, on peut donc clairement parcourir tout le voisinage en temps polynomial pour déterminer la meilleure solution parmi les voisins. On peut donc construire trois applications polynomiales A_L, B_L et C_L comme décrites dans la définition 19. Et donc MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ/Flip appartient à la classe PLS.

1.4.4.3 Complétude et difficulté de la classe PLS

Pour cette classe PLS, l'ensemble des problèmes complets va contenir les problèmes de recherche locale où calculer un optimum local est le plus difficile. On peut voir ces problèmes comme ceux qui nécessitent le plus d'appels récursifs de la fonction de recherche locale avant d'obtenir un optimum local en utilisant l'algorithme de recherche locale standard. La réduction utilisée est la PLS-réduction, définie comme suit.

Définition 20

Soient $\Pi_1/\mathcal{N}_1$ et $\Pi_2/\mathcal{N}_2$ deux problèmes de recherche locale. $\Pi_1/\mathcal{N}_1$ est PLS-réductible à $\Pi_2/\mathcal{N}_2$ s'il existe une application $f : \Sigma_{\Pi_1} \mapsto \Sigma_{\Pi_2}$ telle que pour toute instance $I \in \Sigma_{\Pi_1}$:

— $f(I)$ est calculable en temps polynomial ;

— il existe une application $g : sol(f(I)) \mapsto sol(I)$ telle que pour toute solution $x \in sol(f(I))$ telle que x est un optimum local, $g(x)$ est un optimum local.

Historiquement, le premier problème PLS-complet apparaît dans un article de Johnson, Papadimitriou et Yannakakis. Il s'agit du problème de trouver une assignation des

variables dans un circuit booléen [47]. Le problème MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ/Flip a également été montré PLS-complet [55].

1.5 Solveurs

Nous abordons dans cette dernière section du chapitre l'utilisation des solveurs. Il s'agit d'une méthode très utilisée dans la communauté de recherche opérationnelle et qui a pour vocation de résoudre de façon exacte des problèmes NP-difficiles. Les *solveurs* sont des programmes permettant de résoudre des problèmes abstraits. Une grande partie des solveurs servent à résoudre un problème mathématique appelé *programmation linéaire en nombres entiers* (ILP pour *integer linear programming* en anglais). Dans ce type de problèmes, on a un ensemble de variables entières, un ensemble de contraintes et une fonction (ou plusieurs fonctions) à optimiser. Ces solveurs peuvent avoir été optimisés pendant plusieurs années afin de réduire au maximum leurs temps de résolution. Pour cela, ils peuvent utiliser un certain nombre de stratégies ou même d'heuristiques diverses afin d'accélérer la résolution des instances. Ainsi ces solveurs peuvent être utilisés pour tenter de résoudre des instances de taille assez importante. Il est relativement facile de construire une réduction à partir d'un grand nombre de problèmes vers une formulation en ILP. Ainsi, une stratégie pour résoudre un nouveau problème peut être de créer une formulation ILP pour ce problème puis d'utiliser un solveur pour résoudre cette formulation et ainsi profiter de tout le travail d'optimisation effectué sur le solveur en question. Les problèmes de SATISFAISABILITÉ peuvent être vus comme des cas particuliers d'ILP et il existe d'ailleurs de puissants solveurs SAT utilisés pour résoudre des problèmes de décision.

Par exemple, pour le problème ENSEMBLE INDÉPENDANT, on peut établir la formulation en nombres entiers suivante. Soit G le graphe donné en entrée. Pour chaque sommet $v_i \in V(G)$, on introduit une variable binaire x_i qui prend la valeur 1 si le sommet v_i appartient à l'ensemble indépendant et la valeur 0 dans le cas contraire. Pour chaque arête v_iv_j, on ne peut pas par définition avoir les deux sommets dans l'ensemble indépendant et pour empêcher cela on introduit la contrainte suivante :

$$x_i + x_j \leq 1 \tag{1.1}$$

Comme on souhaite maximiser le nombre de sommets dans l'ensemble indépendant, on souhaite maximiser la somme des valeurs des variables x_i et donc la fonction objective à optimiser est la suivante :

$$max \sum_{i \leq |V(G)|} x_i \tag{1.2}$$

Toutes les modélisations ne se valent pas en termes de temps d'exécution et cela peut aussi dépendre du solveur utilisé. La modélisation des problèmes est un champ d'étude complexe et nous ne ferons qu'effleurer le sujet dans ce tapuscrit.

II

Définitions des problèmes

Dans ce chapitre, nous introduirons les deux problèmes principaux de cette thèse : l'échafaudage et la linéarisation. Avant cela, nous présenterons le contexte biologique dans lequel ils s'inscrivent puis nous aborderons brièvement le problème de l'assemblage des lectures.

2.1 Séquençage génomique

Le *génome* est un ensemble d'informations relatives à un organisme qui permettent le bon fonctionnement de celui-ci en lui indiquant une liste de protéines à produire. Chez la plupart des organismes, cette liste est encodée sur une molécule appelée *acide désoxyribonucléique* (ADN) qui est présente dans toutes les cellules sous la forme de *chromosomes*. Une molécule d'ADN est constituée d'une suite d'éléments de base qui sont les *nucléotides*. Il existe quatre types de nucléotides différents dans l'ADN : l'adénine (A), la guanine (G), la thymine (T) et la cytosine (C). C'est l'ordre de ces nucléotides qui va permettre d'encoder l'information sur les protéines à produire. Ainsi, on peut voir l'ADN comme une longue séquence de lettres. Chaque protéine à produire va être déterminée par la présence d'une sous-séquence dans l'ADN, appelée *gène*. Les protéines produites prédéterminent des caractéristiques, appelées traits, pour l'organisme dans lequel elles sont produites. Par exemple, la présence d'un ensemble de gènes chez l'humain peut prédéterminer la couleur des cheveux, la forme des oreilles...

FIGURE 2.1 – *Première séquence génomique publiée en 1973 qui comprend vingt-quatre nucléotides.*

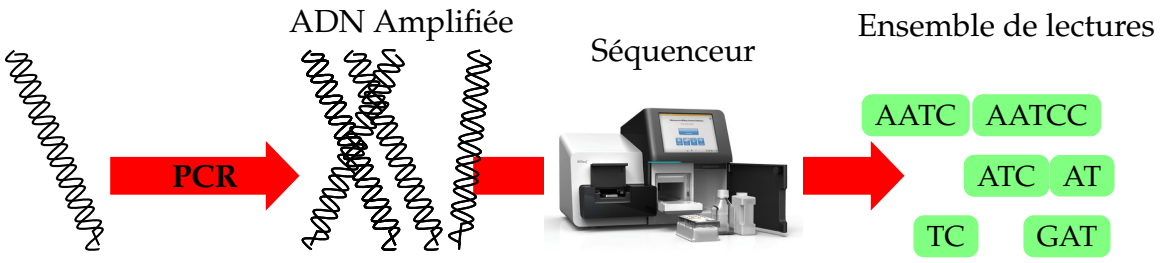


FIGURE 2.2 – *Les différentes étapes du séquençage.*

La connaissance de cette information génétique est cruciale dans beaucoup de domaines. Tout d'abord, il faut savoir que le génome est unique pour chaque individu (à l'exception des individus issus du clonage). Ainsi, il est possible de déterminer de quel organisme est issue une cellule pour laquelle on aurait réussi à lire le matériel génétique. En médecine légale, on peut par exemple utiliser ce principe pour produire des indices pour une enquête policière. Une autre propriété de l'ADN est que l'information génétique d'un individu se transmet en partie à sa descendance, on parle d'*hérédité*. Plus deux individus ont des similitudes dans leurs génomes, plus ils sont proches du point de vue de la parenté. Cela permet des applications immédiates en généalogie ou en phylogénétique. Enfin, on peut également citer des utilisations en médecine comme la détection de maladies héréditaires.

Malheureusement, il n'existe pas, à l'heure actuelle, d'outil permettant d'observer directement l'information renfermée dans l'ADN. Pour connaître la séquence de nucléotides constituant un fragment d'ADN, on doit au préalable passer par un processus appelé *séquençage*. La méthode de séquençage la plus ancienne à s'être démocratisée a été inventée par Frederick Sanger [79] vers la fin des années 1970 avant de prendre son essor dans les années 1980 avec l'avènement des séquenceurs automatiques grâce à la découverte de la *réaction en chaîne par polymérase* (*PCR* pour *Polymerase Chain Reaction* en anglais). Dans la méthode Sanger, le séquençage est composé des étapes suivantes.

1. Extraction du fragment génomique dont on veut connaître l'ordre des nucléotides.
2. Amplification de ce fragment génomique à l'aide de la PCR : il s'agit d'une réaction chimique qui permet de faire de multiples copies du fragment originel.
3. Passage de cet ADN amplifiée dans une machine appelée *séquenceur*. Cette machine va couper aléatoirement les brins d'ADN en plusieurs petits fragments. Dans chacun de ces fragments, on connaît l'ordre des nucléotides aux extrémités. On obtient ainsi deux sous-séquences génomiques qu'on appelle *lectures* (ou *reads* en anglais). On dit que ces deux lectures sont *appariées*. La *taille d'insertion* du fragment correspond à la taille de la sous-séquence présente entre ces deux lectures, qui n'est la plupart du temps pas lue, voir la figure 2.3 . La taille d'insertion peut être nulle, dans ce cas la séquence génomique du fragment est entièrement connue.

Depuis les années 2005, une nouvelle technologie de séquenceurs est apparue, il s'agit des séquenceurs haut-débits. Ces séquenceurs, grâce à de nouvelles techniques, permettent de produire des quantités plus grandes de données à un coût inférieur. Certains séquenceurs de cette nouvelle génération permettent également de produire des lectures plus longues par rapport à ceux de la génération précédente. Sortir des lec-

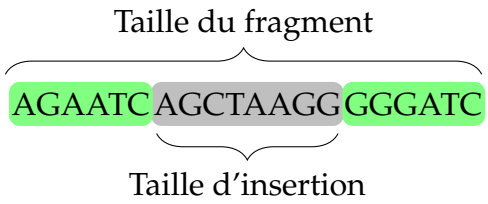


FIGURE 2.3 – *Dans un séquençage en lectures courtes, pour chaque fragment découpé, une paire de lecture est obtenue. La taille d'insertion correspond à la distance entre les deux lectures. Dans cet exemple, elle est égale à huit.*

tures longues permet notamment de gérer plus facilement les sous-séquences répétées du génome. Cependant, des lectures plus longues contiennent en général davantage d'erreurs (ce n'est pas forcément le cas pour les toutes dernières générations de séquenceur, mais leur prix est encore assez élevé et ils sont donc peu répandus). Le modèle sur le lequel nous travaillerons dans ce document s'intéresse à l'échafaudage des génomes séquencés par des lectures courtes. En effet, bien que la dernière génération de séquenceurs produit des lectures longues, une grande partie des génomes présents dans les bases de données sont obtenus à l'aide de lectures courtes. Utiliser un modèle utilisant ces lectures courtes permet donc de retirer des informations de ces génomes sans avoir besoin de les séquencer à nouveau.

Une fois que le génome a été séquencé, le problème est que celui-ci n'est connu que de façon parcellaire. Il faut ensuite le reconstruire à partir des lectures obtenues par le séquençage. Globalement, cela peut être vu comme un immense puzzle où l'on ne connaîtrait pas l'image à assembler et où chaque pièce correspond à une lecture. Chacune des lectures contient environ une centaine de nucléotides. Comme le génome peut contenir des millions de nucléotides, il est illusoire de le reconstituer « à la main ». On doit donc faire intervenir des traitements informatiques pour créer la séquence génomique originelle. Afin d'y parvenir, une des méthodes utilisées est la *méthode globale* (ou *shotgun* en anglais pour illustrer le caractère aléatoire de la fragmentation de la séquence génomique). Cette méthode globale est traditionnellement séparée en deux étapes.

— L'assemblage qui permet de créer des séquences en utilisant les chevauchements existants entre les lectures. L'assemblage permet de créer des séquences consensus du génome appelées *contigs*.

— L'échafaudage qui va utiliser la structure du génome afin d'échafauder les contigs en séquences.

Les travaux de cette thèse se concentrent sur l'échafaudage ainsi que sur un autre problème appelé linéarisation qui intervient quand certains contigs sont présents plusieurs fois dans la séquence originelle. Dans ce cas-là, plusieurs séquences différentes peuvent être construites à partir du même ensemble de contigs et un traitement supplémentaire est nécessaire pour ne pas créer une séquence chimérique. Un schéma représentant ces étapes est dessiné dans la figure 2.4. Nous y reviendrons plus en détail dans les sections suivantes.

Le traitement informatique des données issues du séquençage est un véritable défi en informatique théorique. Les problèmes que nous présenterons ici ne constituent qu'une infirme partie des traitements que l'on peut réaliser. Parmi les problèmes que

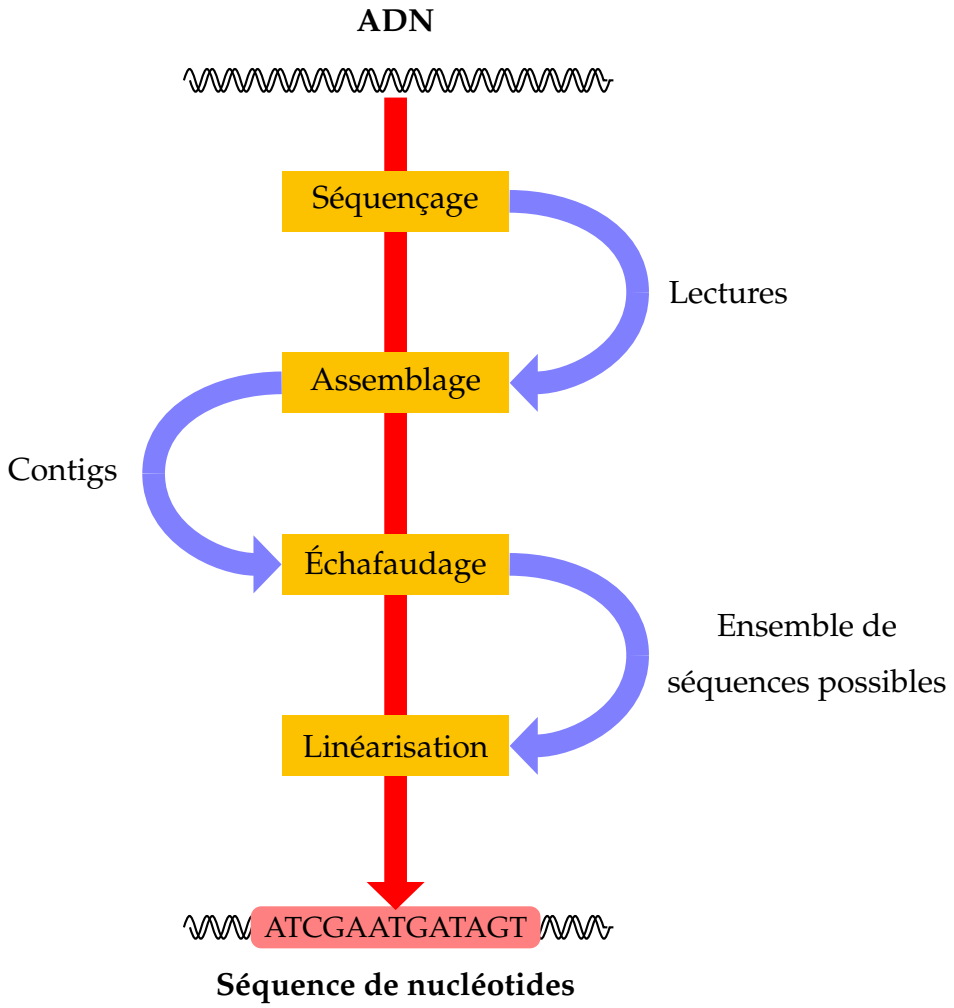


FIGURE 2.4 – *Schéma représentant les étapes de la méthode globale permettant d'obtenir la séquence génomique à partir de l'ADN.*

nous n'aborderons pas, on peut par exemple citer le traitement des erreurs de lecture lors du séquençage : en effet en fonction du type de séquenceur utilisé ou de la zone de l'ADN séquencée (les polymères des chromosomes sont plus difficiles à séquencer à cause des plis de la molécule d'ADN), les données peuvent contenir un nombre plus ou moins important d'erreurs. Repérer et corriger ces erreurs peut s'avérer compliqué. Comme outil servant à palier ce problème de mauvaises lectures, on peut citer par exemple LoRDEC [78] qui est un outil permettant la correction de lectures longues à l'aide de lectures courtes. D'autres problèmes interviennent bien plus tard dans la chaîne des traitements : construction d'arbres phylogénétiques, identification des régions codantes du génome…

Nous aimerions finir cette section en invitant le lecteur à prendre un peu de recul avec les modèles qui seront présentés dans cette thèse. En effet les séquences génomiques produites à partir de ceux-ci ne sont pas forcément complètement exactes. On peut citer trois raisons à cela. D'une part, comme dit précédemment, les données issues du séquençage peuvent comporter des erreurs de lectures et celles-ci ne peuvent pas tout le temps être corrigées avec des traitements informatiques. Une deuxième raison est que les algorithmes utilisés pour résoudre les modèles peuvent consister en des

algorithmes d'approximation ou des heuristiques. Enfin, les modèles créés permettent d'utiliser des outils mathématiques afin de résoudre des problèmes issus du réel. Malheureusement ces modèles ne permettent pas de capturer la réalité dans son entière complexité et donc la solution construite à partir des modèles peut être différente de la solution réelle. Malgré ces remarques, la résolution de ces problèmes est importante, ne serait-ce que d'un point de vue théorique.

2.2 Assemblage

La première étape de la méthode globale est l'*assemblage*. Pour ce problème, on travaille sur les lectures issues du séquençage. Ces lectures sont typiquement des séquences de plusieurs centaines de nucléotides. On peut les voir comme des mots appartenant à un alphabet $\mathcal{A}$, composés de quatre lettres : $\mathcal{A} = \{A, T, C, G\}$. Un *mot* est tout simplement une séquence de symboles $(s_1, \ldots, s_n)$ contenus dans un alphabet $\mathcal{A}$. On rappelle qu'un brin d'ADN est copié plusieurs fois avant d'être segmenté en lectures. Comme les coupes des brins d'ADN sont effectuées de façon aléatoire, il existe des chevauchements de mêmes sous-séquences nucléiques entre les lectures. On peut utiliser ces chevauchements pour essayer de trouver la séquence originelle. Par exemple si on a deux lectures $ACTGCAG$ et $CAGTTAG$, on peut remarquer que les trois derniers nucléotides de la première lecture correspondent aux trois premiers nucléotides de la seconde. On peut donc suspecter que la séquence originelle contient la sous-séquence $ACTGTTAG$.

On va modéliser ce problème avec un problème d'optimisation appelé PLUS COURTE SUPER-SÉQUENCE. Soient $m = (\ell_1, \ldots, \ell_n)$ et $m' = (\ell'_1, \ldots, \ell'_{n'})$ deux mots tels que $n \leq n'$. Le mot m' *contient* m s'il existe deux entiers i et i' tels que pour tout $j < n$, $\ell_{i+j} = \ell'_{i'+j}$. Soit un ensemble de mots $\mathcal{F} = \{m_1, \ldots, m_n\}$. Une *super-séquence* de $\mathcal{F}$ est un mot contenant tous les mots de $\mathcal{F}$. Le problème utilisé pour résoudre l'assemblage est défini comme suit.

PLUS COURTE SUPER-SÉQUENCE

Entrée : Un ensemble de mots $\mathcal{F}$.
Sortie : Une super-séquence de longueur minimum.

Ce problème est NP-difficile, même si l'alphabet utilisé ne contient que deux symboles [73]. Le meilleur algorithme approché connu pour ce problème possède un facteur d'approximation égal à $^{23}/_{22}$ [48]. La résolution de ce problème va permettre d'extraire des séquences consensus qui vont constituer des *contigs*. Ces contigs vont ensuite être utilisés pour échafauder le génome. La figure 2.5 explique comment les contigs sont créés à partir de la résolution du problème PLUS COURTE SUPER-SÉQUENCE. Dans le problème formulé ici, on ne possède pas de génome de référence pour aider à placer les lectures. On appelle cela un *assemblage de novo*. Dans certains cas, il peut cependant arriver qu'un génome de référence soit utilisé. Dans ce cas, la séquence de référence utilisée est similaire mais pas forcément identique.

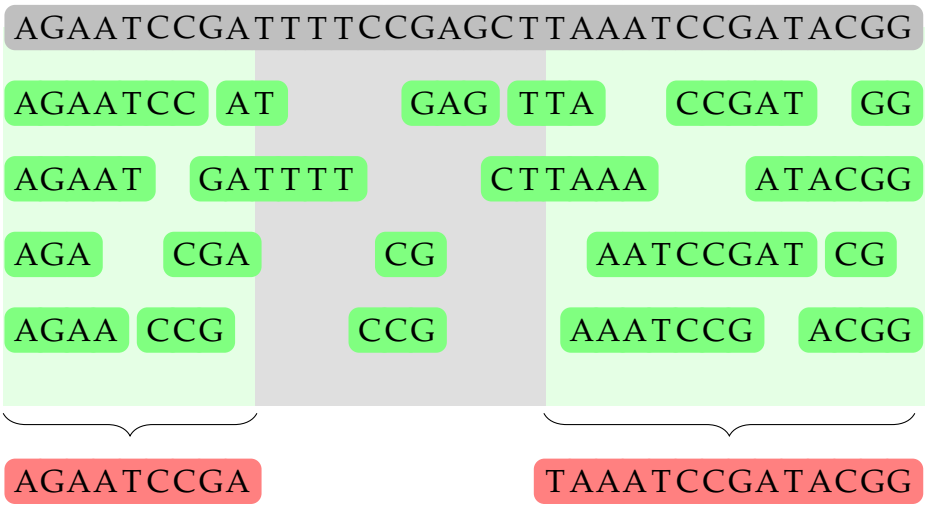


FIGURE 2.5 – *Assemblage d'un ensemble de lectures. La séquence du haut, dessinée en gris, est la séquence originelle à reconstituer. Celle-ci est inconnue. Les lectures sont placées relativement les unes par rapport aux autres en fonction des chevauchements. On remarque que des sous-séquences de la séquence originelle sont couvertes par une forte densité de lectures (sur fond vert) et d'autres par une plus faible densité (sur fond gris). Les sous-séquences possédant une forte densité vont former des contigs (en rouge) qui vont être utilisés pour l'échafaudage.*

2.3 Échafaudage

Une fois les lectures assemblées en contigs, on va utiliser la structure du génome afin d'échafauder ces contigs en séquences. Pour cela, à partir des contigs issus de l'assemblage, on va construire un graphe d'échafaudage défini comme suit.

Définition 21

Un *graphe d'échafaudage* est un 3-uplet (G^*, M^*, ω) où :

— G^* est un graphe ;

— M^* est un couplage parfait de G^* ;

— $\omega : E(G^*) \setminus M^* \mapsto \mathbb{N}$ est une fonction de poids sur les arêtes non couplantes.

Concrètement, chaque sommet du graphe d'échafaudage va correspondre à une extrémité d'un contig. Les arêtes de couplage vont donc représenter les contigs. Une arête non couplante va relier deux extrémités de deux contigs différents si l'on suspecte que ces deux contigs sont reliés dans la séquence génomique. Enfin la fonction de poids sur les arêtes non couplantes va représenter le taux de confiance sur l'existence de la liaison entre les deux contigs. Plus le poids sur une arête non couplante est élevé, plus la liaison a de chance d'exister. La fonction de poids est la plupart du temps construite en prenant en compte les lectures appariées partagées entre les deux contigs. La fonction de poids peut tout simplement correspondre au nombre de paires de lectures partagées entre deux contigs (les positions des lectures sur les contigs permettent de connaître quelles extrémités de ces contigs relier). Bien sûr, on pourrait

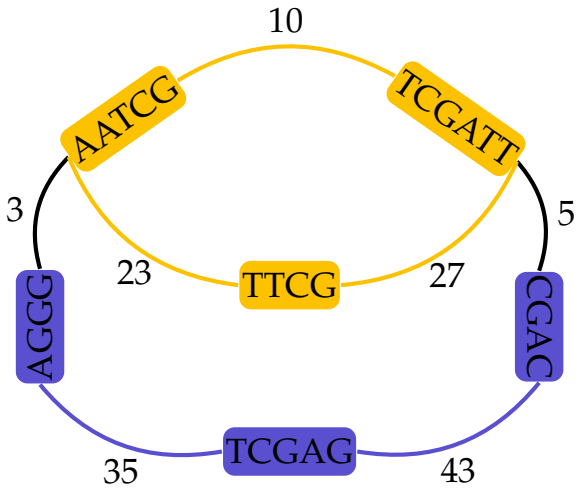


FIGURE 2.6 – *Exemple d'un graphe d'échafaudage dans lequel on recherche un cycle alterné et un chemin alterné. Les arêtes de couplage sont représentées avec les rectangles contenant les séquences. Les nombres sur les arêtes non couplantes correspondent aux poids. Une solution optimale de* ÉCHAFAUDAGE MAXIMUM *est représentée avec un cycle alterné en jaune et un chemin alterné en bleu.*

utiliser d'autres types de données pour construire la fonction de poids, par exemple la distance entre deux contigs ou des informations phylogénétiques. Mais cela ne change pas la forme du modèle.

Une fois ce graphe d'échafaudage construit, on va chercher à reconstruire les séquences génomiques en utilisant les informations sur la structure du génome. L'ADN est stockée dans une cellule sous la forme de chromosomes qui peuvent être soit circulaires, soit linéaires. Par exemple, une cellule humaine contient quarante-six chromosomes linéaires et un chromosome circulaire (ce dernier est présent dans le matériel génétique des mitochondries qui sont une structure présente dans la cellule). Dans un graphe d'échafaudage, ces chromosomes vont prendre la forme de cycles alternés pour les chromosomes circulaires ou de chemins alternés pour les chromosomes linéaires. Si l'on connaît le nombre de chromosomes de chaque type dans l'organisme, la construction des séquences va correspondre à la recherche du nombre correspondant de chemins et de cycles alternés dans le graphe d'échafaudage. Si ce nombre n'est pas connu, on va en général rechercher un unique chemin alterné dans le graphe.

2.3.1 Définition du problème

Concrètement, on peut formuler le problème de l'échafaudage par une version généralisée du problème PROMENADE DES PONTS. En entrée du problème, on aura le graphe d'échafaudage et deux entiers σ_c et σ_p qui vont représenter le nombre de chromosomes circulaires et linéaires, respectivement.

Nous étudierons deux versions du problème l'échafaudage : une version sous la forme d'un problème de décision et une version sous la forme d'un problème d'optimisation. Le problème de décision est défini comme suit.

ÉCHAFAUDAGE

Entrée : un graphe d'échafaudage (G^*, M^*, ω), deux entiers σ_c et σ_p
Question : existe-t-il une collection de σ_c cycles alternés et σ_p chemins alternés disjoints ?

Pour la version d'optimisation, on va utiliser la fonction de poids ω du graphe d'échafaudage afin de choisir la meilleure collection de cycles et de chemins alternés.

ÉCHAFAUDAGE MAXIMUM

Entrée : un graphe d'échafaudage (G^*, M^*, ω), deux entiers σ_c et σ_p
Sortie : une collection de σ_c cycles alternés et σ_p chemins alternés disjoints telle que la collection soit de poids maximum.

Notons qu'on a pu formuler un problème de décision ÉCHAFAUDAGE pour le problème de l'échafaudage qui ne correspond pas tout à fait au problème de décision associé à ÉCHAFAUDAGE MAXIMUM (il n'y a pas de contrainte sur le poids dans ÉCHAFAUDAGE). Comme nous le verrons par la suite, cela fait de ÉCHAFAUDAGE MAXIMUM un problème structurellement difficile. Un petit exemple de ÉCHAFAUDAGE MAXIMUM est donné par la figure 2.6.

Soit (G^*, M^*, ω) un graphe d'échafaudage et soit S une collection de cycles alternés et de chemins alternés dans (G^*, M^*, ω). On appelle *cardinalité* de S le nombre de cycles alternés et de chemins alternés de S. On dénotera par $\sigma_c(S)$ le nombre de cycles alternés de cette collection et par $\sigma_p(S)$ le nombre de chemins alternés de S. Le poids de S, noté $\omega(S)$, correspond à la somme des poids des arêtes non couplantes de S. Formellement, on a $\omega(S) = \sum_{e \in S \setminus M^*} \omega(e)$.

Dans la formulation de ce problème, on cherche des chemins et des cycles alternés, ce qui implique que chaque arête de couplage n'apparaît qu'une seule fois dans la collection. En pratique, comme nous le verrons plus loin, un contig peut apparaître plusieurs fois dans la séquence génomique. Nous verrons dans la section 2.4 comment modéliser les possibles multiples occurences des contigs.

2.3.2 État de l'art et contributions

Dans cette section, nous faisons un état de l'art du problème de l'échafaudage. Il est bien entendu impossible d'être exhaustif, nous ne présenterons qu'une partie de l'ensemble des travaux existants sur ce problème.

Tout d'abord, notons que certaines approches peuvent être très spécifiques par rapport aux types de données sur lesquelles elles vont travailler. C'est le cas notamment de celle présentée dans [80] qui introduit `Bambus2`, un outil permettant d'effectuer l'écha-

faudage de métagénomes, c'est-à-dire l'ensemble de l'information génétique présente dans un échantillon d'un milieu prélevé dans la nature : estomacs, air, sols… L'échafaudage des métagénomes diffère de l'échafaudage classique car on ne souhaite pas reconstituer un ensemble de chromosomes mais avoir un aperçu de l'ensemble des gènes qui composent les organismes d'un environnement. De ce fait, cet outil n'est en pratique pas utilisable pour l'échafaudage classique. On peut également citer [66, 74] dans lesquels est décrite une méthode permettant d'échafauder les génomes anciens en temps polynomial à l'aide d'autres génomes déjà assemblés.

Pour le problème de l'échafaudage « classique », il existe plusieurs façons de modéliser le génome avant de l'échafauder. Une caractéristique en général commune aux différentes modélisations est que le problème consiste en la recherche d'un ou plusieurs chemins dans un graphe. Par exemple, dans [44], il est présenté comme un problème NP-difficile consistant à fusionner des chemins dans un graphe. Ces travaux présentent également une heuristique pour résoudre ce problème. De façon générale, il existe deux principaux moyens de modéliser un graphe d'échafaudage : soit comme nous l'avons fait en utilisant un couplage, où les arêtes couplantes vont représenter les contigs, soit en modélisant chaque contig par un unique sommet, l'orientation des contigs est alors simulée en orientant les arêtes du graphe.

Par exemple, dans [77], la façon de modéliser le graphe d'échafaudage diffère de celle que nous avons présentée dans ce chapitre. En effet, dans cet article les contigs ne sont pas représentés par des arêtes de couplage mais par un unique sommet. Pour modéliser l'orientation des contigs, les arêtes du graphe sont *biorientées*, cela veut dire qu'elles possèdent deux orientations, une pour chaque contig relié à l'arête. Chaque arête du graphe possède une distance. Pour reconstituer la séquence génomique, on essaie de calculer un chemin parcourant tout le graphe qui maximise le nombre de distances d'insertion qui sont respectées. Une formulation MIP (pour *mixed integer programming*) a été formulée pour résoudre de façon approchée cette modélisation. Cette formulation a abouti à la création d'un outil open-source `MIP Scaffolder`.

Un cadre plus général est proposé dans [21] où la modélisation est présentée comme un problème de couverture dans un hypergraphe. La modélisation qui est utilisée inclut des multiplicités sur les contigs, permettant à ceux-ci d'apparaître plusieurs fois dans la séquence. Des résultats d'approximation sont proposés dans le cas où il n'y a pas de contraintes sur le nombre de chemins et de cycles. D'autres approches utilisent l'information phylogénétique [43, 53], des informations sur la structure de la chromatine des chromosomes [18] ou des informations présentes sur les lectures longues (pour les génomes qui ont été séquencés à nouveau avec la dernière génération de séquenceurs). Dans [8], l'échafaudage est effectué à l'aide d'une analyse impliquant des informations sur l'ordre des gènes et le réarrangement du génome.

Un certain nombre d'outils combinant différentes approches sont apparus depuis les années 2010, la plupart sont accessibles en open-source. Nous avons déjà cité `MIP Scaffolder`. Dans `SOPRA` [31], les auteurs ont introduit une technique permettant de retirer les contigs problématiques et se sont servis d'une méta-heuristique appelée *recuit simulé*. Le recuit simulé consiste à simuler une température tout en effectuant une recherche locale, mais celle-ci diffère de la recherche locale « classique » car on s'autorise à parcourir des solutions voisines qui ne sont pas les meilleures, du moins tant que

la température est basse. Un autre outil, `Opera` [37], contracte le graphe d'échafaudage ce qui permet de limiter la quantité nécessaire de mémoire et donc permet d'utiliser un algorithme dynamique pour une résolution exacte. L'avantage de cette méthode exacte est qu'elle produit des génomes de bonne qualité. Dans `SSPACE` [15], un algorithme quasi-glouton est utilisé pour l'échafaudage. Dans `GRASS` [39], un algorithme évolutionniste est proposé. Dans les grandes lignes, un algorithme évolutionniste consiste à générer plusieurs solutions puis à simuler « l'évolution darwiniste » en croisant des solutions sélectionnées ou en effectuant des mutations sur ces solutions. Les qualités de génomes produites par cette approche génétique sont situées entre celles des solutions fournies par `SSPACE` et celles fournies par `Opera`. Dans `SCARPA` [35], le problème de l'échafaudage est divisé en deux sous-problèmes : l'un consistant à orienter les contigs et l'autre consistant à les ordonner les uns par rapport aux autres. L'orientation est calculée à l'aide d'un algorithme FPT et l'ordonnancement est effectué à l'aide d'une heuristique. Cet outil n'est pas nécessairement le plus rapide mais a l'avantage d'être peu gourmand en termes de mémoire. Dans [17], une heuristique en programmation linéaire en nombres entiers est décrite. Cette heuristique, en plus de produire des génomes de bonne qualité, a l'avantage de pouvoir être exécutée sur des ordinateurs personnels. L'outil `BESST` [76] utilise une approche proche de celle utilisée dans ce tapuscrit pour la modélisation, car le poids des arêtes du graphe représente le nombre de paires de lectures partagées. Cet outil permet de traiter des génomes de taille importante et semble notamment dépasser la plupart des autres solveurs quand la distribution des tailles d'insertion entre les lectures appariées est large. `ScafMatch` [68] utilise une heuristique s'appuyant sur le calcul d'un couplage de poids maximum. Plus récemment, `BOSS` [67] utilise lui aussi une heuristique pour la résolution de l'échafaudage, notamment en simplifiant le graphe d'échafaudage en lui retirant certains contigs.

En 2014, une comparaison des différents outils existants à cette date est effectuée dans [42]. Elle compare les performances des outils sur des instances réelles et simulées suivant plusieurs critères, notamment la qualité de la solution produite et l'espace mémoire nécessaire.

Concernant le problème ÉCHAFAUDAGE tel que formulé dans cette thèse, il est assez proche du problème du voyageur du commerce pour lequel il existe une littérature très abondante, un aperçu est notamment disponible dans [58]. Pour le problème de trouver une collection de cycles et chemins disjoints, certains articles mettent en lumière certaines conditions suffisantes, en particulier sur les degrés des sommets par exemple [23]. La littérature a également exploré les cas où le nombre de cycles et de chemins n'est pas fixé. Ainsi, il a été montré que trouver une collection de cycles devient polynomial lorsque le nombre de cycles n'est pas fixé [84]. Au contraire, la recherche d'une couverture optimale de chemins comprenant au moins deux arêtes est NP-difficile [82]. C'est également le cas pour la recherche d'une couverture de cycles [56].

Les travaux présentés dans cette thèse sont dans la continuité d'un certain nombre d'articles étudiant la complexité et l'approximation des problèmes ÉCHAFAUDAGE et ÉCHAFAUDAGE MAXIMUM ou de versions modifiées de ceux-ci. Dans [20], ÉCHAFAUDAGE est montré NP-complet à l'aide d'une construction similaire à la construction 1. Deux algorithmes d'approximation dans les graphes complets sont également déve-

loppés. Une exploration de la complexité en fonction des différentes classes de graphes est faite dans [87]. Dans [27], une mesure est proposée pour les solutions issues d'un algorithme d'approximation sur les graphes complets afin de donner une indication sur leurs qualités. L'exploration de la complexité est complétée dans [85], où notamment la complexité paramétrée est évoquée ainsi qu'un nouveau problème consistant à trouver les arêtes vitales d'un graphe. Enfin, dans [86], une approche exacte est développée en utilisant une décomposition arborescente. Un aperçu de l'état de l'art sur la complexité est donné dans le tableau 2.1. Un *quasi-arbre* est un graphe d'échafaudage (G^*, M^*, ω) tel que le sous-graphe partiel contenant les arêtes $E(G^*) \setminus M^*$ est un arbre. Un graphe est cobiparti G s'il existe un graphe biparti G' tel $V(G) = V(G')$ et $E(G') = \{uv \mid uv \notin E(G)\}$ (*i.e.* G est le graphe complémentaire d'un graphe biparti). À ce tableau, on peut ajouter deux résultats sur l'approximation : dans les graphes complets et les graphes bipartis complets, il existe un algorithme 2-approché et, dans les quasi-arbres, il n'est pas possible d'approximer ÉCHAFAUDAGE MAXIMUM avec un facteur d'approximation meilleur que $|V|^{\frac{1}{2}-\epsilon}$, pour tout $\epsilon > 0$.

TABLE 2.1 – *Aperçu de l'état de l'art pour le complexité de l'échafaudage.*

<table>
<tr><th rowspan="2">Topologie</th><th colspan="3">ÉCHAFAUDAGE MAXIMUM</th><th colspan="2">ÉCHAFAUDAGE</th></tr>
<tr><th colspan="2">$\sigma_p > 0$</th><th>$\sigma_p = 0$</th><th>$\sigma_p > 0$</th><th>$\sigma_p = 0$</th></tr>
<tr><td>Bipartis</td><td colspan="2" rowspan="2"></td><td colspan="2" rowspan="3">NP-difficile</td><td colspan="2">NP-complet</td></tr>
<tr><td>Cobipartis</td><td colspan="2">P</td></tr>
<tr><td>Scindés</td><td colspan="2" rowspan="2">Pas sous-exponentiel</td><td colspan="2">Ouvert</td></tr>
<tr><td>Quasi-arbres</td><td>$W[1]$-difficile</td><td>NP-difficile</td><td>P</td><td>Ouvert</td><td>P</td></tr>
</table>

En ce qui concerne les travaux de cette thèse sur le problème de l'échafaudage, nous compléterons ce tableau en y ajoutant un résultat impliquant la maille alternée du graphe d'échafaudage. Nous donnerons aussi quelques résultats d'inapproximation sur des classes particulières de graphes. Enfin, nous donnerons une généralisation d'un algorithme glouton sur les graphes complets présenté dans [20] afin qu'il puisse s'adapter à d'autres classes de graphes.

2.4 Linéarisation

2.4.1 Problèmes des multiples occurrences

Une des difficultés majeures à gérer pendant toutes les étapes est que certaines séquences du génome se répètent plusieurs fois au sein de celui-ci. Notamment, pendant l'assemblage, cela va poser problèmes quand les lectures sont plus petites que les séquences répétées. Ainsi, comme le montre la figure 2.7, ces séquences répétées vont avoir tendance à être considérées comme une unique séquence. On peut cependant grâce à des analyses statistiques inférer sur le nombre d'un contig dans la séquence. Ces données peuvent être ensuite prises en compte dans l'étape de l'échafaudage, à condition que le modèle ait été pensé pour considérer les occurrences multiples des contigs.

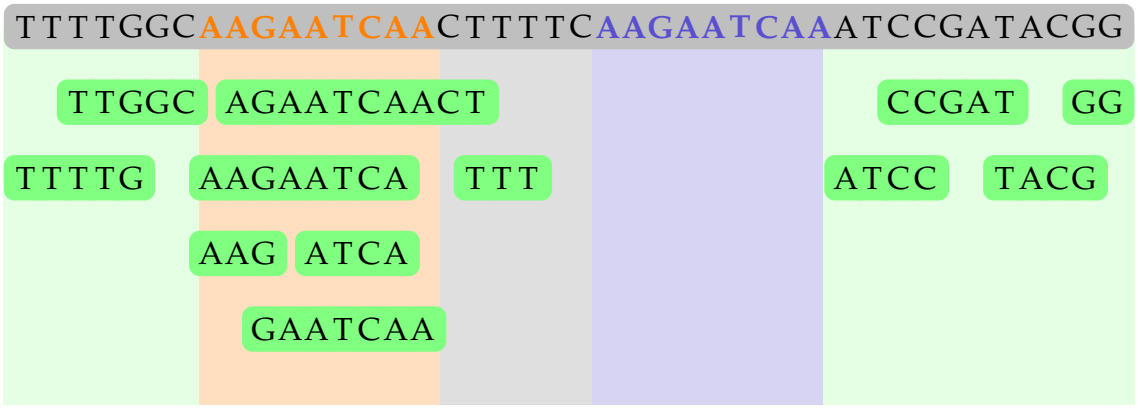


FIGURE 2.7 – *Exemple d'assemblage faussé par les multiplicités. Dans cet exemple une sous-séquence (en orange et bleu) se répète deux fois. Les lectures appartenant à la sous-séquence bleue vont se placer sous la sous-séquence orange. On peut cependant deviner que la sous-séquence orange apparaît deux fois dans le génome, car le nombre de lectures qui appartiennent à cette sous-séquence est deux fois plus élevé que le nombre de lectures appartenant aux autres sous-séquences conservées.*

2.4.2 Échafaudage avec multiplicités

Dans le modèle présenté dans ce tapuscrit pour le problème de l'échafaudage, nous avons considéré que chacun des contigs n'apparaît qu'une seule fois dans la séquence génomique. Pour pouvoir prendre en compte qu'un contig peut avoir plusieurs occurrences dans le génome, nous pouvons introduire un nouveau paramètre appelé *multiplicité*. Soit (G^*, M^*, ω) un graphe d'échafaudage, la multiplicité $m : M^* \mapsto \mathbb{N}$ est une fonction sur les arêtes de couplage indiquant le nombre de fois qu'une arête de couplage doit apparaître dans une solution. Comme une arête de couplage peut apparaître plusieurs fois au sein d'un même élément alterné, on parlera de marches alternées ouvertes ou fermées à la place de chemins alternés et de cycles alternés, respectivement. On dit qu'une collection de marches alternées ouvertes ou fermées dans (G^*, M^*, ω, m) respecte les contraintes de multiplicité si chaque arête de couplage uv apparaît $m(uv)$ dans la collection. Formellement, nous formulons le problème d'échafaudage prenant en compte les multiplicités comme suit.

ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM

Entrée : un graphe d'échafaudage avec multiplicités (G^*, M^*, ω, m).
Sortie : une collection de σ_p marches ouvertes et σ_p marches fermées dans (G^*, M^*, ω, m) respectant les contraintes de multiplicités et telle que le poids de la collection soit maximum.

Cependant, quand les multiples occurrences des contigs sont prises en comptes, des ambiguïtés peuvent apparaître, comme le montre la figure 2.8. La présence ou non d'ambiguïtés est due au fait que deux solutions différentes peuvent utiliser exactement le même ensemble d'arêtes. Pour la traduire, à partir d'une solution S, nous créons un *graphe de solution* constitué de la fusion des marches alternées de la solution. La production de séquences linéaires à partir du graphe de solution est appelée *linéari-*

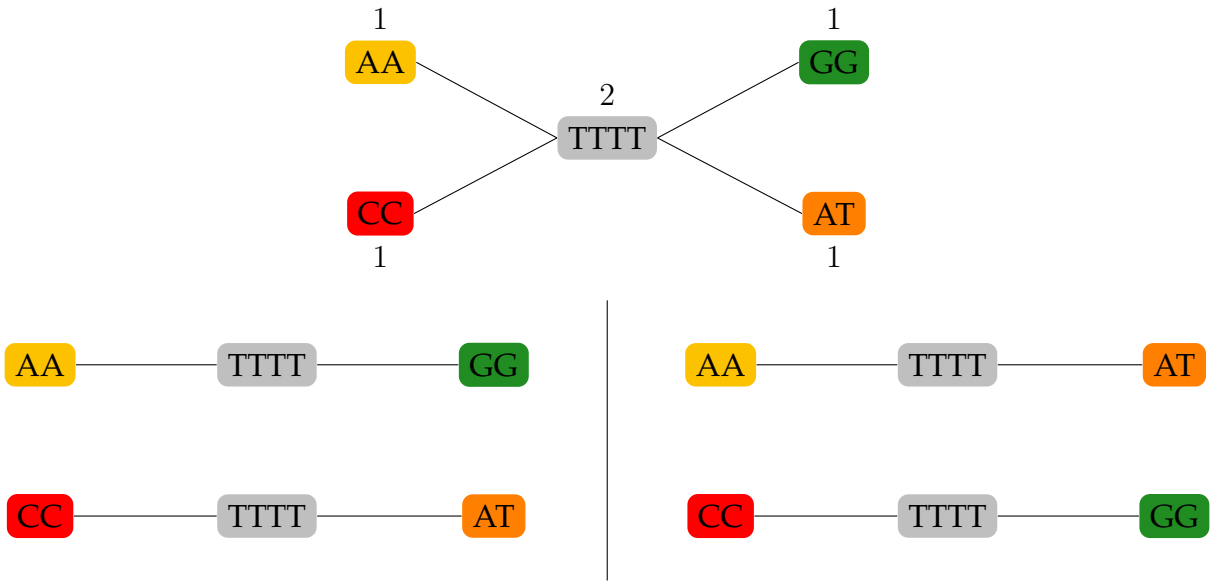


FIGURE 2.8 – *Schéma illustrant une ambiguïté.* ***Haut :*** *Graphe d'échafaudage contenant cinq arêtes de couplage et dans lequel on cherche deux marches alternées ouvertes. Le contig* `TTTT` *apparaît deux fois dans la séquence génomique, les autres contigs apparaissent une fois.* ***Bas :*** *deux solutions sont possibles pour ce graphe d'échafaudage, l'une produisant les séquences* `AATTTTGG` *et* `CCTTTTAT` *(à gauche) et l'autre produisant les séquences* `AATTTAT` *et* `CCTTTTGG` *(à droite). Comme ces deux solutions utilisent exactement le même ensemble d'arêtes, elles sont de même poids et choisir arbitrairement l'une ou l'autre peut mener à produire une séquence chimérique.*

sation. Si le graphe de solution présente des ambiguïtés, alors il n'est pas possible de le linéariser sans faire de choix arbitraires. Ces choix arbitraires sont à éviter, car ils peuvent produire des séquences chimériques (*i.e.* qui n'apparaissent pas dans le génome cible). Dans la suite de cette section, après avoir défini le graphe de solution et caractérisé les ambiguïtés, nous présenterons une stratégie pour éliminer les ambiguïtés, sous forme de problème d'optimisation. Ensuite, nous expliquerons comment « linéariser » un graphe de solution en marches alternées en cas d'absence d'ambiguïtés. Enfin, les dernières parties de cette section seront consacrées à montrer comment simplifier l'intitulé du problème, à montrer quelques propriétés utiles ainsi qu'à faire un résumé des résultats obtenus pour ce problème.

2.4.3 Graphe de solution et ambiguïtés

Formellement, le graphe de solution est défini comme suit.

Définition 22

Soient (G^*, M^*, ω, m) un graphe d'échafaudage avec multiplicités et S une solution de ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM dans (G^*, M^*, ω, m). Le graphe de solution de S est défini par le tuple (G^*_s, M^*, ω, m') pour lequel :

— G^*_s est le sous-graphe de G^* tel que $V(G^*_s) = V(G^*)$ et $E(G^*_s) = E(G^*_s) \cap$

$E(G^*)$;

— $m' : E(G_s^*) \mapsto \mathbb{N}$ est la fonction de multiplicité telle que pour toute arête e, $m'(e)$ est le nombre d'occurrences de e dans S.

Notons que le couplage est conservé par rapport au graphe d'échafaudage et que pour toute arête de couplage, on a $m(e) = m'(e)$. Un graphe de solution peut être vu comme un graphe d'échafaudage pour lequel on aurait enlevé le bruit constitué par les mauvaises arêtes et pour lequel on aurait fixé la multiplicité des arêtes n'appartenant pas au couplage. Le graphe de solution est en fait une manière de représenter toutes les solutions utilisant le même ensemble d'arêtes. On peut noter la propriété structurele suivante.

Propriété 3

Soit (G_s^*, M^*, ω, m') un graphe de solution, pour toute arête non couplate $uv \in E(G_s^*) \setminus M^*$, nous avons $m'(uv) \leq m'(uM^*(u))$ et $m'(uv) \leq m'(vM^*(v))$.

Cela est dû au fait qu'un graphe de solution est construit à partir d'une solution de ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM. En effet, s'il existait une arête non couplante uv telle que $m'(uv) > m'(uM^*(u))$, cela signifierait que la solution qui construit ce graphe contient soit une marche alternée qui commence par une arête non couplante soit que l'arête de couplage $uM^*(u)$ est empruntée un nombre de fois supérieur à sa multiplicité, ce qui n'est pas correct dans les deux cas.

Soit S une solution de ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM, on dit que S est *ambiguë* s'il existe une collection de marches alternées différente S' telle que S et S' ont le même graphe de solution. Notons que S et S' ne possèdent pas nécessairement le même nombre de marches ouvertes ou fermées, comme le montre la figure 2.9. Pour constater qu'une solution est ambiguë, il n'est cependant pas nécessaire de trouver une autre collection possédant le même graphe de solution. En effet, l'observation du graphe de solution est suffisante. La notion d'ambiguïté est alors caractérisée par la définition suivante.

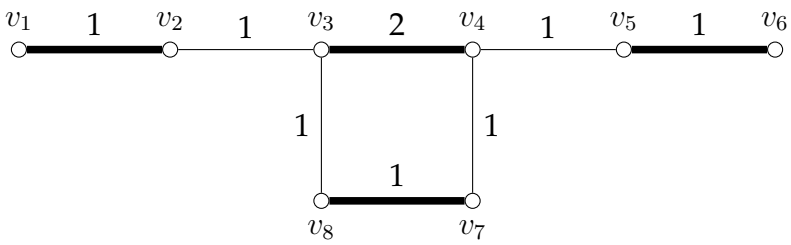


FIGURE 2.9 – *Exemple de graphe de solution ambigu. Il existe deux collections de permettant de construire ce graphe de solution* $S_1 = (v_1, v_2, v_3, v_4, v_7, v_8, v_3, v_4, v_5, v_6)$ *et* $S_2 = (v_1, v_2, v_3, v_4, v_5, v_6), (v_7, v_8, v_3, v_4, v_7)$. S_1 *est constituée d'une marche alternée ouverte et* S_2 *est constituée d'une marche alternée ouverte et d'une marche alternée fermée. On peut repérer l'ambiguïté du graphe de solution par la présence du chemin ambigu* (v_3, v_4).

Définition 23

Soit A un élément alterné dans (G_s^*, M^*, ω, m'). Si toutes les arêtes de A ont pour multiplicité μ, alors A est appelé μ-*uniforme* (ou simplement *uniforme*, si μ n'est pas connu). A est appelé *ambigu* si l'un des deux cas suivants est vrai :

— si A est un cycle alterné μ-uniforme et $\mu > 1$;

— si A est un chemin alterné tel que pour chaque extrémité, il existe une arête incidente de multiplicité strictement inférieure à μ.

Dans la définition précédente, on parle bien de chemins ou cycles alternés et non de marches ouvertes ou fermées. On parlera plus spécifiquement de chemin ambigu ou de cycle ambigu. L'ambiguïté dans un chemin ambigu vient du fait qu'il y a plusieurs façons de relier des séquences à chaque extrémité de chemins ambigus. Pour un cycle ambigu, l'ambiguïté vient du fait qu'il n'est pas possible de savoir combien de tours va effectuer une marche alternée fermée à l'intérieur du cycle : par exemple si l'on cherche trois marches alternées fermées dans un cycle 6-uniforme, on peut le décomposer en trois marches faisant deux tours chacune ou en deux marches faisant un tour et une marche faisant quatre tours. Il est intéressant de noter que la présence ou non de chemins ambigus ou de cycles ambigus dans un graphe de solution est suffisante pour montrer l'ambiguïté d'une collection, comme le montre le résultat suivant.

Théorème 3

Une solution de ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM n'est pas ambiguë si et seulement si son graphe de solution ne contient pas d'éléments alternés ambigus.

La preuve du théorème 3 sera donnée dans la sous-section 2.4.5.

2.4.4 Éliminer les ambiguïtés

Un graphe de solution issu d'une solution optimale est une manière d'encoder toutes les collection de même poids d'un graphe d'échafaudage. Cependant, même si un graphe de solution délivre des informations intéressantes, il n'est pas utilisable pour des applications biologiques, qui pour la plupart ont besoin de séquences linéaires. Il faut donc linéariser au préalable le graphe de solution en séquences. Si un graphe de solution (G_s^*, M^*, ω, m') contient des éléments alternés ambigus, plusieurs stratégies sont possibles.

Ignorer. Choisir arbitrairement une solution ayant (G_s^*, M^*, ω, m') comme graphe de solution. Toutefois, comme ce choix peut mener à produire une séquence chimérique, sans aucun moyen de l'évaluer, cette stratégie est à proscrire d'un point de vue bio-informatique.

Croisement de données. Utiliser un autre critère pour choisir la solution (par exemple N50[1]). La solution retournée est alors une solution ayant (G_s^*, M^*, ω, m') comme

1. N50 est une mesure statistique sur la taille des contigs : étant donné un ensemble de contigs, la

graphe de solution et maximisant ce critère. Toutefois, on trouvera également des cas où plusieurs solutions peuvent maximiser ce critère, et donc on ne pourra pas exclure la production de chimères.

Brutale. Détruire tous les éléments alternés ambigus. Pour chaque chemin alterné ambigu, enlever toutes les arêtes non couplantes incidentes aux extrémités du chemin. Pour chaque cycle alterné ambigu, enlever une arête n'appartenant pas au couplage. Cela transforme tous les éléments alternés ambigus en chemins uniformes isolés. Cette méthode permet d'éviter la production de chimères mais fait perdre beaucoup de poids à la solution, ce qui n'est pas intéressant d'un point de vue bio-informatique.

Semi-Brutale. Choisir un ensemble « approprié » d'extrémités de chemins alternés ambigus et enlever toutes les arêtes non couplantes incidentes à celles-ci. Pour chaque cycle alterné ambigu, enlever une arête n'appartenant pas au couplage.

Nous nous concentrerons sur la méthode semi-brutale. La méthode semi-brutale permet d'enlever les arêtes de façon plus parcimonieuse que la méthode brutale et donc de préserver davantage le poids de la solution. Toutefois, tandis que la stratégie brutale peut être réalisée rapidement, la stratégie semi-brutale demande davantage de temps de calcul pour choisir un ensemble de sommets « approprié ». Soit v une extrémité d'un chemin ambigu ou un sommet d'un cycle ambigu. Le terme *couper* v signifie enlever du graphe de solution toutes les arêtes non couplantes incidentes à v, et dans ce cas v est dénoté comme une *coupe*. Formellement, la méthode semi-brutale peut être définie comme une solution du problème suivant.

COUPE SEMI-BRUTALE

Entrée : un graphe de solution (G_s^*, M^*, ω, m')
Sortie : un ensemble de coupes X permettant de détruire tous les éléments alternés ambigus et tel que le score de X est minimum.

Dans la sous-section 2.4.6, nous verrons comment simplifier l'intitulé du problème. Notons qu'il n'est pas possible d'unifier ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM et COUPE SEMI-BRUTALE en un seul problème, car encore une fois, cela peut mener à créer une séquence chimérique, comme le montre la figure 2.10. Plusieurs scores à optimiser peuvent être considérés.

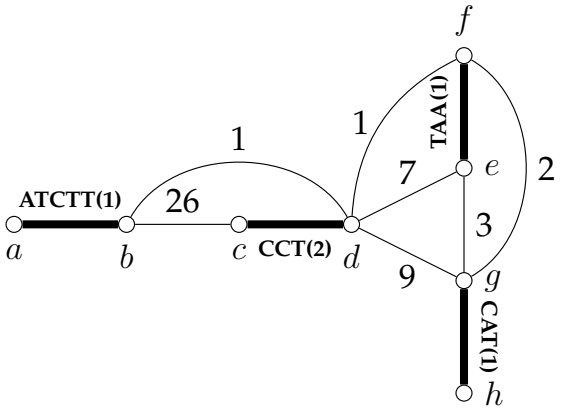
Score de coupe. Minimiser le nombre de coupes : $score(X) = |X|$.

Score de liaison. Minimiser le nombre d'arêtes enlevées du graphe : $score(X) = \sum\{m'(uv) \mid uv \in E(G_s^*) \setminus M^* \wedge \{u, v\}\} \cap X = \varnothing\}$.

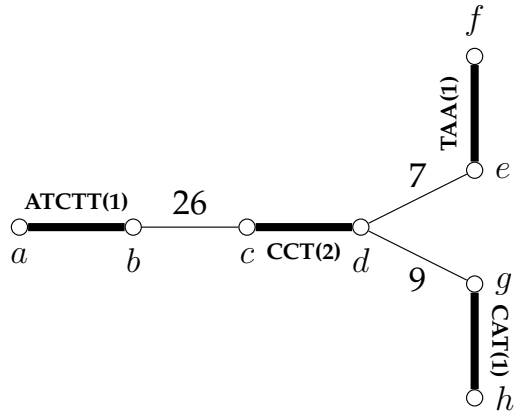
Score de poids. Minimiser le poids total des arêtes retirées : $score(X) = \sum\{m'(uv) \cdot \omega(uv) \mid uv \in E(G_s^*) \setminus M^* \wedge \{u, v\}\} \cap X = \varnothing\}$.

Notons que d'un point de vue de complexité de calcul, le score de liaison est un cas spécial de score de poids, car il suffit de donner à toutes les arêtes n'appartenant pas au couplage le poids un pour que les scores soient équivalents. C'est pourquoi, dans le

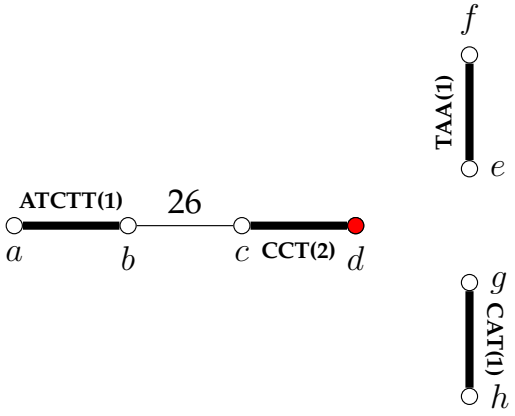
mesure N50 est définie comme la longueur du plus court contig utilisé pour couvrir 50% du génome.



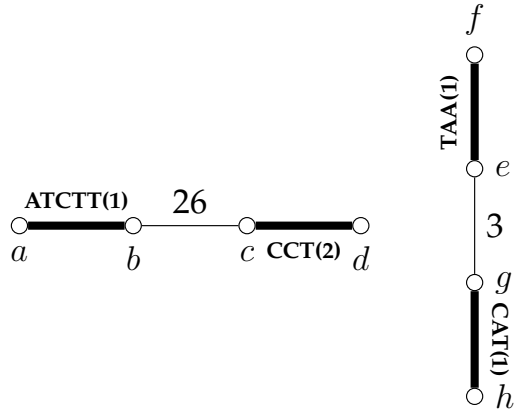
(a) *Graphe d'échafaudage. Ce graphe illustre les relations entre quatre contigs, représentés par les arêtes de couplage ab, cd, ef et gh, dessinées en gras. Les labels sur les arêtes de couplage représentent les séquences des contigs et leurs multiplicités (entre parenthèse). L'arête cd, dont la séquence est* CCT*, a une multiplicité de deux. Les autres contigs ont une multiplicité de un. Les labels des arêtes n'appartenant pas au couplage représentent les poids de ces arêtes.*



(b) *Graphe de solution après avoir résolu* ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM*. Ce graphe de solution est obtenu à partir d'une solution contenant deux marches alternées ouvertes de poids 42. Toute les arêtes n'appartenant pas au couplage sont de multiplicité une. L'arête de couplage labélée* CCT *est un chemin ambigu, ce qui mène à deux ensemble de séquences possibles* {ATCCT..CCT..TAA, CCT..CATG} *et* {ATCCT..CCT..CATG, CCT..TAA}*.*



(c) *Linéarisation du graphe de solution en utilisant la méthode semi-brutale. La méthode brutale donne un ensemble de quatre séquences indépendantes, avec un poids total de zéro. La méthode semi-brutale pour le score de coupe donne un ensemble unique de quatre séquences* {ATCCT..CCT, CCT, TAA, CATG}*, de poids total 26 (avec un score de poids de 16). Le sommet d est coupé afin de détruire le chemin ambigu constitué de l'arête cd.*



(d) *Linéarisation directe du graphe d'échafaudage. Chercher directement deux marches alternées ouvertes de poids maximum mène à garder la liaison chimérique entre les deux contigs* TTA *et* CATG*.*

FIGURE 2.10 – *Exemple d'une instance hypothétique constituée des deux chromosomes* ATCTT..CCT..TAA *et* CCT..CATG*.*

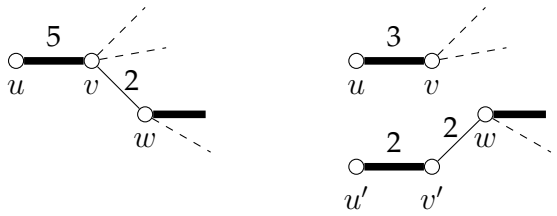


FIGURE 2.11 – *Exemple d'application de la règle 1 sur le chemin alterné 5-uniforme $(u, \ldots, v)$. Les numéros sur les arêtes représentent les multiplicités. Les arêtes de couplage sont dessinées en gras.*

reste de ce tapuscrit, nous ne parlerons que des scores de coupe et de poids. Comme nous le verrons par la suite, trouver des solutions optimales pour ces deux scores s'avère être un problème NP-difficile.

2.4.5 Linéariser le graphe de solution

Dans cette partie, nous montrons formellement qu'un graphe de solution (G_s^*, M^*, ω, m') peut être déconstruit de façon unique en marches alternées s'il ne contient pas d'éléments ambigus. Autrement dit, s'il n'existe pas deux solutions différentes ayant le même graphe de solution. On dira alors que (G_s^*, M^*, ω) peut être « linéarisé de façon unique ». À cette fin, nous introduisons la règle de réduction suivante sur (G_s^*, M^*, ω, m').

Règle 1

Soit $p = (u, \ldots, v)$ un chemin alterné μ-uniforme entre les sommets u et v tel que $deg(u) = 1$. Soit vw une arête n'appartenant pas au couplage. Effectuer les opérations suivantes :

1. créer le chemin alterné $m'(uv)$-uniforme $p' = (u', \ldots, v')$;
2. ajouter une arête $v'w$ telle que $m'(v'w) = m(vw)$;
3. enlever l'arête vw ;
4. décroître la multiplicité des arêtes de p de $m(vw)$.

Voir la figure 2.11 pour un exemple. Prouvons l'exactitude de cette règle, en montrant la propriété suivante.

Lemme 1

Soit $G_1 = (G_{s_1}^*, M_1^*, \omega_1, m_1')$ un graphe de solution et soit $(G_{s_2}^*, M_2^*, \omega_2, m_2')$ le graphe de solution résultant de l'application de la règle 1 sur G_1. G_1 peut être linéarisé de façon unique si et seulement G_2 peut être linéarisé de façon unique.

Preuve.

Soient $\mathcal{W}_1$ et $\mathcal{W}_2$ les ensembles de décompositions en marches alternées possibles pour les G_1 et G_2, respectivement. Considérons la fonction $\tau : \mathcal{W}_1 \mapsto \mathcal{W}_2$ associant chaque décomposition de $\mathcal{W}_1$ à une décomposition de $\mathcal{W}_2$. On associe une décomposition D_1 à une décomposition D_2 si le remplacement de toutes les occurrences de $(u, \dots, v)$ (ou $(v, \dots, u)$) dans les marches de D_1 par $(u', \dots, v')$ (ou $(v', \dots, u')$) est égal à D_2. Clairement, deux décompositions différentes de $\mathcal{W}_1$ ne peuvent pas être associées à une même décomposition de $\mathcal{W}_2$ et donc τ est injective. Pour montrer que τ est surjective, supposons qu'il existe une décomposition $D_2 \in \mathcal{W}_2$ qui n'a pas d'antécédent dans $\mathcal{W}_1$. Notons que toute marche alternée $M \in D_2$ contenant la sous-marche $(u', \dots, v')$ contient également l'arête $v'w$ car $m(v'w) = m(M^*(v')v')$ et une marche alternée ne peut pas commencer ou finir par une arête n'appartenant pas au couplage. Remplacer toutes les occurrences de $(u', \dots, v')$ (ou $(v', \dots, u')$) dans les marches de D_2 par $(u, \dots, v)$ (ou $(v, \dots, u)$) mène à une décomposition de D_1 de G_1 et dans ce cas, on a $\tau(D_1) = D_2$, ce qui est une contradiction. Ainsi toutes les décompositions de $\mathcal{W}_2$ ont un antécédent et donc τ est bijective. Cette bijection implique que le nombre de décompositions possibles pour G_1 est égal au nombre de décompositions possibles pour G_2 et donc la propriété du lemme est vraie. □

Nous pouvons à présent prouver le théorème 3.

Théorème 3

Une solution de ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM n'est pas ambiguë si et seulement si son graphe de solution ne contient pas d'éléments alternés ambigus.

Preuve.

Montrons que (G_s^*, M^*, ω, m') peut être linéarisé de façon unique si et seulement si il ne contient pas d'éléments ambigus.

⇒ Supposons par l'absurde que (G_s^*, M^*, ω, m') contient un élément ambigu et que (G_s^*, M^*, ω, m') peut être linéarisé de façon unique. Supposons dans un premier temps que (G_s^*, M^*, ω, m') contient un cycle ambigu c. Notons μ la multiplicité des arêtes de c. Pour tout entier $k \leq \mu$, on dénote par c^k la marche alternée fermée utilisant k fois chaque arête de c. Le cycle ambigu c peut être décomposé en deux ensembles de marches alternées $\{c^\mu\}$ et $\{c^{\mu-1}, c^1\}$, ce qui contredit l'unicité d'une décomposition.

Supposons à présent que (G_s^*, M^*, ω, m') contient un chemin ambigu $p = (u, \dots, v)$ et soit D une décomposition de (G_s^*, M^*, ω, m') en marches alternées. Soit $\mu > 1$ la multiplicité des arêtes de p et soient xu et yv deux arêtes non couplantes incidentes à u et v, respectivement. Par définition, on a $m'(xu) < \mu$ et $m'(yv) < \mu$. Aucune marche alternée de D ne peut commencer par une arête interne de couplage e de p car sinon cela voudrait dire que e est adjacente à une arête n'appartenant pas au couplage qui est

empruntée un nombre inférieur de fois à μ dans D ou qu'une marche alternée de D commencerait par une arête n'appartenant pas au couplage. C'est pourquoi, quand une marche alternée traverse k fois l'arête $uM^*(u)$, cette marche traverse k fois tout le chemin p. On peut distinguer trois cas.

- S'il existe une marche alternée (m_1, p, m_2, p, m_3) dans D passant deux fois par p, il est possible de construire une autre décomposition de (G_s^*, M^*, ω, m') en remplaçant $\{(m_1, p, m_2, p, m_3)\}$ par $\{(m_1, p, m_3), (p, m_2)\}$ dans D (où (p, m_2) est un cycle).
- Sinon soit $\overline{p}$, la sous-marche parcourant les sommets de $\overline{p}$ dans son sens inverse. S'il existe une marche alternée $(m_1, p, m_2, \overline{p}, m_3)$ dans D, passant une fois par p et une autre fois par $\overline{p}$, alors soit $\overline{m}_2$ la sous-marche parcourant les sommets de m_2 dans son sens inverse. Il est possible de construire une autre décomposition de (G_s^*, M^*, ω, m') en remplaçant $(m_1, p, m_2, \overline{p}, m_3)$ par $(m_1, p, \overline{m}_2, \overline{p}, m_3)$ dans D.
- Sinon, comme $\mu > 1$, il existe deux marches alternées (m_1, p, m_2) et (m_1', p, m_2') empruntant au moins une fois p, il est possible de construire une autre décomposition de (G_s^*, M^*, ω, m') en remplaçant (m_1, p, m_2) et (m_1', p, m_2') par (m_1, p, m_2') et (m_1', p, m_2) dans D.

Ainsi, dans tous les cas, on arrive à une contradiction et donc on peut conclure que (G_s^*, M^*, ω, m') peut être linéarisé de façon unique s'il ne contient pas d'élément ambigu.

$\Leftarrow$ Montrons que si (G_s^*, M^*, ω, m') ne contient pas d'élément ambigu, alors il peut être linéarisé de façon unique. On suppose que la règle 1 a été appliquée à (G_s^*, M^*, ω, m') autant de fois que possible. Si (G_s^*, M^*, ω, m') est vide, alors l'unique ensemble de marches décomposant (G_s^*, M^*, ω, m') est l'ensemble vide. Si (G_s^*, M^*, ω, m') contient un cycle alterné uniforme, alors comme c n'est pas ambigu, c est 1-uniforme et donc ne peut être décomposé que par l'ensemble $\{c\}$. Si (G_s^*, M^*, ω, m') n'est pas constitué uniquement de cycles, considérons $p = (u, \ldots, v)$ un chemin alterné μ-uniforme maximal. Tous les sommets internes de p sont de degré deux. Supposons, sans perte de généralité, que $deg(u) \leq deg(v)$. Si $deg(u) = 1$ et $deg(v) = 2$, alors la règle 1 s'applique. Si $1 < deg(u) \leq deg(v)$, alors par maximalité de p, il existe deux arêtes xu et yv telles que $m'(xu) < \mu$ et $m'(yu) < \mu$ et donc p est un chemin ambigu, ce qui constitue une contradiction. Ainsi, on a nécessairement $deg(u) = deg(v) = 1$ et donc p est un chemin isolé qui ne peut être décomposé que par l'ensemble contenant μ fois la marche alternée $(u, \ldots, v)$.

□

On peut utiliser la preuve précédente afin de formuler l'algorithme 5 qui permet de linéariser un graphe de solution en un ensemble de marches alternées.

Algorithme 5 : Algorithme permettant de déconstruire un graphe de solution ne contenant pas d'élément ambigu en marches alternées.

Entrée : Un graphe de solution (G_s^*, M^*, ω, m').
Sortie : Une déconstruction D de (G_s^*, M^*, ω, m') en marches alternées.
// G.
1 Appliquer récursivement la règle 1 jusqu'à ce que (G_s^*, M^*, ω, m') soit constitué de cycles alternés 1-uniformes et de chemins alternés uniformes;
2 $D \leftarrow \varnothing$;
3 **Pour tout** *cycle alterné* 1*-uniforme* c **faire**
4 $\quad D \leftarrow D \cup \{c\}$;
5 **Pour tout** *chemin alterné* μ*-uniforme* p **faire**
6 $\quad$ **Pour tout** $i \in [1, \mu]$ **faire**
7 $\quad\quad D \leftarrow D \cup \{p\}$;
8 **retourner** D;

2.4.6 Simplification du problème

Nous introduisons à présent un ensemble de règles de réduction qui, en plus de la règle 1, permet de simplifier l'intitulé de COUPE SEMI-BRUTALE. Pour toutes ces règles, nous considérons un graphe de solution (G_s^*, M^*, ω, m'). Tout d'abord, nous pouvons enlever les parties triviales du graphe de solution, ce qui se traduit par les deux règles suivantes.

Règle 2

Soit c un cycle isolé μ-uniforme. Si $\mu = 1$, alors enlever c du graphe de solution, sinon transformer c en un chemin alterné μ-uniforme en lui enlevant son arête de poids le plus faible.

Règle 3

Enlever tous les chemins alternés uniformes isolés du graphe de solution.

Ensuite, comme on ne modifie pas les arêtes internes des chemins alternés uniformes, on peut simplifier le graphe de solution en contractant chacun des chemins uniformes en une unique arête de couplage.

Règle 4

Soit $p = (u, \ldots, v)$ un chemin alterné μ-uniforme maximal. Effectuer les opérations suivantes :

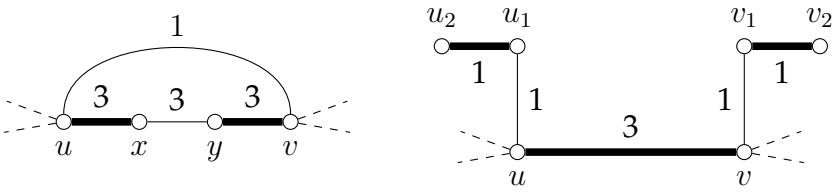


FIGURE 2.12 – *Exemple d'application de la règle 4 sur le chemin alterné 3-uniforme (u, x, y, v), les numéros sur les arêtes représentent les multiplicités. Les arêtes de couplage sont dessinées en gras.*

1. enlever tous les sommets internes de p ;
2. créer l'arête de couplage uv avec $m'(uv) = \mu$;
3. s'il existe une arête n'appartenant pas au couplage e entre u et v, alors
   — créer deux arêtes de couplage u_1u_2 et v_1v_2 ayant pour multiplicité $m'(e)$,
   — enlever l'arête e,
   — créer les arêtes de poids nul uu_1 et vv_1.

Preuve (Exactitude la règle 4).

Tout d'abord, comme aucun sommet interne de p ne peut être coupé, replacer p par une unique arête de couplage ne change pas le score d'une solution, et ce pour les deux fonctions de score. Il reste à montrer l'exactitude quand l'arête n'appartenant pas au couplage $e = uv$ existe. On a $m(e) < \mu$ car sinon $p \cup \{e\}$ consisterait en un cycle alterné μ-uniforme isolé et serait enlevé par la règle 2. Dans ce cas, le p est un chemin ambigu et toute solution X pour COUPE SEMI-BRUTALE contient nécessairement soit le u, soit v (pour pouvoir enlever l'arête e). Après l'application de cette règle, p est toujours ambigu et une des deux arêtes parmi $\{uu_1, vv_1\}$ doit être enlevée dans une solution. Les deux sommets u_1 et v_1 ne peuvent pas être coupés, car ils n'appartiennent pas à un chemin ambigu. Ainsi l'ensemble X est une solution dans le graphe de départ, alors il est également une solution après avoir appliqué la règle. X possède le même score de coupe dans les deux graphes, par définition. Pour le score de poids, le score de X décroît exactement de $\omega(e)$ et donc la distance entre deux solutions différentes est la même dans les deux graphes. □

La règle suivante transforme chaque chemin ambigu, après sa contraction en une unique arête couplante par la règle précédente, de façon à ce qu'il ne soit rattaché qu'à un seul chemin ambigu.

Règle 5

Soit uv une arête de couplage qui n'appartient pas à un chemin ambigu et telle que $deg(u) \geq 2$ et $deg(v) \geq 2$. Effectuer les opérations suivantes :

1. enlever l'arête uv ;

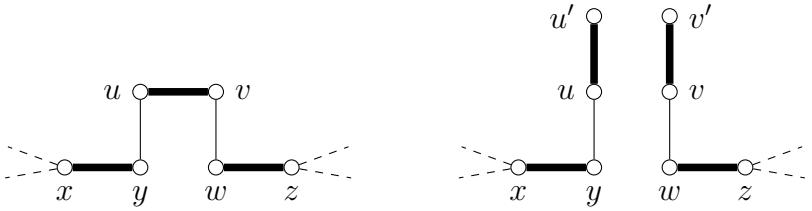


FIGURE 2.13 – *Exemple d'application de la règle 5 sur l'arête de couplage uv. Les arêtes de couplage sont dessinées en gras et toutes les arêtes ont pour multiplicité un.*

2. ajouter deux sommets u' et v' ;
3. ajouter deux arêtes de couplage uu' et vu' de multiplicité $m'(uv)$.

Preuve (Exactitude de la règle 5).

Aucun chemin ambigu n'est changé, créé ou détruit par l'application de la règle. De plus comme les deux sommets u' et v' sont de degré un, aucune solution ne peut contenir u' ou v'. Comme le score d'une solution ne dépend que des arêtes n'appartenant pas au couplage, et qu'aucune n'a été modifiée, toute solution a le même score avant et après l'application de la règle dans la règle, pour les deux fonctions de score. □

Toutes les règles de réduction présentées ici peuvent être appliquées en temps linéaire. De plus, après l'application de toutes ces règles, il s'avère que toute arête de couplage appartient soit à un chemin ambigu, soit possède un sommet de degré un. Dans ce dernier cas, on dit que l'arête de couplage est *propre*.

Propriété 4

Un graphe de solution, pour lequel toutes les règles de réduction présentées ont été appliquées, ne contient pas de chemin ambigu si et seulement si toutes ses arêtes de couplage sont propres.

Preuve.

⇒ Considérons une arête de couplage uv qui n'est pas propre. Comme (G_s^*, M^*, ω, m') ne contient pas de chemin ambigu, la règle 5 peut s'appliquer sur uv, ce qui constitue une contradiction.

⇐ Considérons un chemin ambigu $(u, \ldots, v)$, par définition il existe une arête n'appartenant pas au couplage incidente à chacune des extrémités u et v. Et donc, clairement, l'arête de couplage $uM^*(u)$ ne peut pas être propre.

□

Ainsi, par la précédente propriété, la fonction de multiplicité n'est plus importante, car le fait de savoir si toutes les arêtes de couplage sont propres suffit pour vérifier une solution. On peut donc reformuler le problème COUPE SEMI-BRUTALE de la façon

suivante.

COUPE SEMI-BRUTALE

Entrée : un graphe de solution (G_s^*, M^*, ω)
Sortie : un ensemble de coupes X permettant de rendre propre toutes les arêtes de M^* et telle que le score de X est minimum.

La notion de chemin uniforme n'a également plus de sens, car ceux-ci ont été contractés en une seule arête par la règle 4. Par la suite, on parlera donc d'arêtes ambiguës et d'arêtes non ambiguës. Seules les arêtes ambiguës peuvent avoir une extrémité coupée dans une solution.

2.4.7 Autres propriétés

Enfin, pour finir sur le problème de la linéarisation des graphes de solution, nous introduisons ici quelques propriétés qui seront utilisées dans le chapitre 4.

Propriété 5

Si v est un sommet d'une arête non ambiguë, alors $deg(v) \leq 2$.

L'exactitude de cette propriété provient de la règle 1 : si une arête non ambiguë ne respecte pas la propriété, alors on peut lui appliquer la règle 1. Pour la fonction de poids, il peut être utile de caractériser le nombre de coupes dans une solution en fonction du nombre d'arêtes ambiguës. C'est pourquoi nous introduisons un type de solution appelé *solution normalisée*. Une solution normalisée est une solution qui contient exactement une coupe par arête de couplage ambiguë. Le lemme suivant montre que nous pouvons transformer n'importe quelle solution en une solution normalisée ayant au moins le même score de poids.

Lemme 2

Soit X une solution de COUPE SEMI-BRUTALE dans le graphe de solution (G_s^*, M^*, ω). Il est possible de construire une solution normalisée X' dans (G_s^*, M^*, ω) telle que $score(X') \leq score(X)$ pour le score de poids.

Preuve.

Comme X est une solution, après avoir enlevé toutes les arêtes non couplantes incidentes à un sommet de X, toutes les arêtes de couplage sont propres. On construit X' en choisissant exactement un sommet de degré un par arête ambiguë après l'application de X. Clairement, X' est une solution et comme chaque arête enlevée par X' est également enlevée par X, on a $score(X') \leq score(X)$. □

2.4.8 État de l'art et contribution

Avant le début des travaux réalisés dans cette thèse, un seul article avait été publié sur le problème de la linéarisation [88]. Cet article explique le contexte du problème et présente quelques résultats de complexité. Un résumé de ces résultats est présenté dans le tableau 2.2.

TABLE 2.2 – *Résumé de l'état de l'art pour* COUPE SEMI-BRUTALE

Topologie	Score	Complexité	Borne inférieure
Chemins	Tous	Linéaire	Ouvert
Arbres			
Général		NP-difficile	
Planaire et $\Delta \leq 4$	Coupe		1.3606 sous P $\neq$ NP $2-\epsilon, \forall \epsilon > 0$ sous UGC Pas sous-exponentiel
Général			

Dans le chapitre 4, nous étendrons ces résultats au niveau de la complexité dans la hiérarchie polynomiale (voir tableau 2.3) et au niveau des bornes supérieures et inférieures d'approximation en temps polynomial (voir tableau 2.4). Certains de ces résultats ne sont qu'une extension de ceux présentés dans [88] : par exemple pour les arbres, seul un algorithme optimisant le score de poids a été présenté. Nous ferons formellement la preuve de l'exactitude de cet algorithme et montrerons comment nous pouvons l'adapter pour le score de coupe. Ces tableaux ne sont pas non plus exhaustifs, quelques propriétés sur la complexité paramétrée peuvent être dérivés à partir des résultats principaux de complexité.

TABLE 2.3 – *Résumé des résultats de complexité pour* COUPE SEMI-BRUTALE.

Topologies	Score	Complexité	
Complets	Coupe	Linéaire	Théorème 13
Bipartis complets			Théorème 14
Cographes			Théorème 15
Arbres	Tous		Théorème 11
$\Delta \leq 2$			Théorème 12
$\mathcal{G}'$[1]		NP-difficile	Théorème 16
Supergraphes de $\mathcal{G}'$	Poids		Corollaire 7
Graphes scindés	Coupe		Théorème 18

1. Graphes bipartis, planaires et sous-cubiques

Enfin, nous présenterons deux algorithmes exacts permettant de trouver des solutions optimales pour les deux fonctions de score dans le cas général.

TABLE 2.4 – *Résumé des bornes d'approximation pour* COUPE SEMI-BRUTALE.

Topologies	Score	Hypothèse	Borne inférieure		Borne supérieure.
$\mathcal{G}''$[1]	Coupe	P $\neq$ NP	1.00009…	Théorème 19	4-approx. Théorème 25
Sous-cubiques			1.00041…	Théorème 21	
$\Delta \leq 4$			1.0069…	Théorème 24	
$\Delta \leq 5$			1.0128…		
$\Delta \leq 6$			1.0138…		
Graphes scindés			1.360…	Théorème 23	
		UGC	$2-\epsilon$		
Général	Poids		1.01887…	Théorème 20	2-approx. Théorème 26
		P $\neq$ NP	1.01515…		
Sous-cubiques			1.00017…	Théorème 19	

1. Graphes bipartis et sous-cubiques

2.5 Implémentation et expérimentation

Enfin, finissons ce chapitre en indiquant quelques informations sur la façon dont les tests ont été effectués. En effet, certains résultats qui sont présentés dans les chapitres 3 et 4 amènent à la description d'algorithmes. Il peut donc être intéressant d'implémenter ces algorithmes et de tester leurs performances sur des instances réelles. Dans cette section, nous décrirons dans un premier temps l'outil `scaftool` dans lequel ont été implémentés ces algorithmes puis nous indiquerons comment les instances réelles ont été générées afin de pouvoir effectuer les tests. Tous les tests effectués dans cette thèse ont été réalisé à l'aide d'un ordinateur personnel possédant $16GO$ de mémoire vive et un processeur $i7$ ayant une cadence de $1.90GHz$ sur huit cœurs.

2.5.1 L'outil scaftool

L'outil `scaftool` est un logiciel écrit en C++ qui permet de répondre au problème de l'échafaudage et de la linéarisation. Ce logiciel s'appuie notamment sur la bibliothèque `Boost` [1], notamment pour les structures de données qui y sont utilisées. Pour décrire une instance réelle pour le problème de l'échafaudage ou de la linéarisation, on envoie en entrée deux fichiers.

— Un fichier *.dot* permettant de donner la structure d'un graphe. Un fichier *.dot* contient la liste des arêtes d'un graphe. On peut également ajouter des informations supplémentaires sur les sommets ou les arêtes. Pour les graphes d'échafaudage ou de solution, on indiquera par exemple le poids et la multiplicité des arêtes et quelles arêtes appartiennent au couplage.

— Un fichier *.fasta* permettant de stocker des séquences nucléiques ou protéiques. On s'en servira afin de donner les séquences associées aux graphes d'échafaudage ou au graphe de solution. Ce fichier est toutefois optionnel car les algorithmes décrits n'ont besoin que de la description du graphe d'échafaudage ou de solution pour s'exécuter.

En sortie du programme, on obtient également un fichier *.dot* et éventuellement un fichier *.fasta* si un tel fichier a également été donné en entrée. Le fichier *.dot* peut conte-

nir soit une collection de cycles et de chemins alternés, un graphe de solution ou un graphe de solution linéarisé, en fonction des arguments passés en paramètre. À l'heure actuelle, l'application scaftool a été découpée en quatre exécutables :

— **samtodot :** cet exécutable permet de créer un graphe d'échafaudage dans un fichier *.dot* à partir d'un fichier d'alignement de séquence. Il a notamment été utilisé afin de générer des instances pour le jeu de tests. Nous y reviendrons plus en détails sur ce rôle par la suite.

— **exact-scaffolding :** cet exécutable permet de répondre aux problèmes ÉCHAFAUDAGE MAXIMUM et ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM grâce à une formulation linéaire en nombres entiers ;

— **meta-scaffolding :** cet exécutable permet de répondre au problème ÉCHAFAUDAGE MAXIMUM grâce à une approche heuristique faisant appel à des algorithmes évolutionnistes ;

— **apx-scaffolding :** cet exécutable permet de répondre au problème ÉCHAFAUDAGE MAXIMUM de façon approchée grâce à un algorithme glouton ;

— **linearisation :** cet exécutable permet de linéariser un graphe de solution.

Sans compter les algorithmes issus des travaux de cette thèse, l'outil `scaftool` a majoritairement été développé par Annie Chateau et Mathias Weller, seul l'exécutable *meta-scaffolding* a été entièrement écrit par Corentin Boennec. Les résultats qui seront présentés par la suite ont permis d'enrichir les méthodes proposées par les deux exécutables *apx-scaffolding* et *linearisation*. Nous reviendrons plus en détails dans les sections traitant des expérimentations dans les chapitres suivants.

2.5.2 Génération des instances

2.5.2.1 Instances pour ÉCHAFAUDAGE

Nous présentons à présent comment les instances ont été générées afin de pouvoir tester les algorithmes. Chacune des instances est créée en suivant les étapes suivantes :

1. un génome de référence est choisi sur une base de données publique, par exemple le site du NCBI dans la base de données Nucleotides ;
2. des lectures pairées sont simulées à partir de ce génome ;
3. un assemblage *de novo* est réalisé à partir de ce jeu de lectures ;
4. des séquences consensus sont extraites grâce à l'alignement des lectures ;
5. un graphe d'échafaudage est généré à partir de cet alignement ;

Le tableau 2.5 présente le jeu de données choisi. Celui-ci a été sélectionné afin d'avoir une variété de génomes en termes de tailles et en termes d'espèces représentées. La base de données est accessible ici : `http://www.ncbi.nlm.nih.gov/`.

La génération des lectures pairées est effectuée à l'aide de l'outil `wgsim` [6, 62]. Les paramètres utilisés lors de cette étape comprennent une longueur des lectures égale à 100 paires de bases et une taille d'insert moyenne de 500 paires de bases (c'est-à-dire la distance entre deux lectures pairées).

TABLE 2.5 – *Jeu de données sélectionné*

Espèce	Sorte	Alias	Taille (bp)	Type	Numéro d'accession
Anopheles Gambiae str. PEST	Insecte	anopheles	41963435	Chromosome 3L	NT_078267.5
Bacillus anthracis str. Sterne	Bactérie	anthrax	5228663	Chromosome	NC_005945.1
Zaire ebolavirus	Virus	ébola	18959	Génome complet	NC_002549.1
Gloeobacter violaceus PCC 7421	Bactérie	gloeobacter	4659019	Chromosome	NC_005125.1
Lactobacillus acidophilus NCFM	Bactérie	lactobacillus	1993560	Chromosome	NC_006814.3
Danaus plexippus	Insecte	monarch	15314	Mitochondrie	NC_021452.1
Pandoravirus salinus	Virus	pandora	2473870	Génome complet	NC_022098.1
Pseudomonas aeruginosa PAO1	Bactérie	pseudomonas	6264404	Chromosome	NC_002516.2
Oryza sativa Japonica	Plante	rice	134525	Chloroplast	X15901.1
Saccharomyces cerevisiae	Levure	sacchr3	316613	Chromosome 3	X59720.2
Saccharomyces cerevisiae	Levure	sacchr12	1078177	Chromosome 12	NC_001144.5

TABLE 2.6 – *Statistiques d'assemblage en prenant* $k = 29$.

Donnée	Temps (s)	Nombre de contigs	Longueur max. des contigs (bp)
anopheles	4346	42045	10043
anthrax	183	4055	9655
ebola	0.5	17	4735
gloebacter	165	4517	9500
lactobacillus	70	1898	7976
monarch	0.5	14	1517
pandora	103	2451	7930
pseudomonas	238	5248	9430
rice	2.5	84	12666
sacchr3	7	296	6317
sacchr12	35	889	7147

L'assemblage *de novo* est assuré avec l'outil `minia` [5, 24]. Cet outil réalise l'assemblage grâce à une méthode basée sur un graphe particulier appelé graphe de De Bruijn. L'idée est de décomposer les séquences des lectures en toutes les sous-séquences possibles de taille k. Ces sous-séquences sont appelées k-mers. Sans rentrer dans les détails, pour avoir un assemblage efficace, il est nécessaire d'avoir une taille k des k-mers ni trop grande, ni trop petite. Pour faire simple, avoir une faible taille de k-mers permet de réduire l'espace de stockage utilisé mais diminue la quantité d'informations et augmente les cas ambigus. Une taille importante de k-mers en revanche rend plus facile la résolution de l'assemblage mais risque de produire des séquences discontinues. L'avantage de l'outil `minia` est qu'il utilise le filtre Bloom qui est une structure de données non-exacte permettant de stocker efficacement les k-mers : l'assemblage pour les génomes choisis ne prend que quelques minutes sur un ordinateur ordinaire. On peut donc tester plusieurs tailles différentes et choisir celle adaptée au génome. Le tableau 2.6 présente quelques statistiques sur le jeu de données précédent pour une taille de k-mers donnée.

L'alignement des lectures en contigs est ensuite effectué avec `bwa` [61, 2]. Le choix de cet outil est motivé par la méta-étude réalisée dans [42] indiquant que c'est l'un des outils les plus adaptés pour l'échafaudage de génome. Cet outil utilise un tableau de suffixes pour indexer les séquences de contigs. Une requête sur cette structure permet de trouver les localisations des lectures sur les contigs.

La dernière étape est la construction du graphe d'échafaudage. Cette étape est directement implémentée dans `scaftool` dans l'exécutable `samtodot`. Dans cette étape, on va construire une arête de couplage par contig. Si deux contigs partagent des lectures pairées, alors on va créer une arête non couplante joignant les deux arêtes de couplage correspondantes. La position des lectures sur les contigs va déterminer les sommets auxquels cette arête non couplante va être incidente.Le poids d'une arête non couplante est égal au nombre de paires de lecture associées à cette arête. La représentation graphique de l'instance `monarch` par le logiciel `graphviz` est donnée par la figure 2.14, cela correspond au plus petit des graphes d'échafaudage du jeu de données.

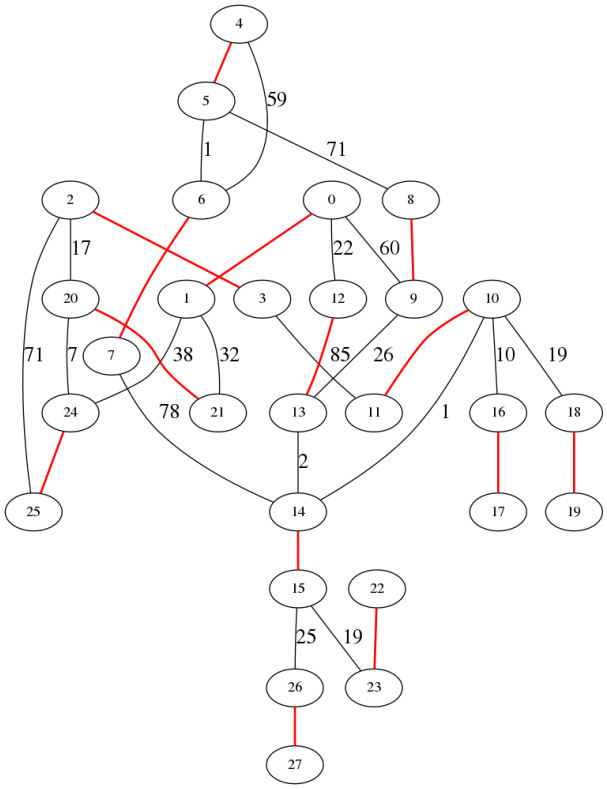


FIGURE 2.14 – *Graphe d'échafaudage correspondant à l'instance* `monarch`*. Le couplage correspond aux arêtes rouges et les numéros sur les arêtes correspondent à leurs poids.*

2.5.2.2 Instance pour COUPE SEMI-BRUTALE

Pour pouvoir générer des graphes de solution comprenant des chemins ambigus, il faut au préalable connaître les multiplicités des contigs sur le génome. Pour cela, nous avons utilisé l'outil `megablast` [69, 4]. Cet outil permet d'aligner les contigs sur un génome de référence, le nombre de fois qu'un contig peut être aligné sur des sous-séquences disjointes correspond à la multiplicité de ce contig.

La résolution de ÉCHAFAUDAGE AVEC MULTIPLICITÉS MAXIMUM est effectuée grâce à une formulation ILP qui est implémentée dans `scaftool` [89]. Notons que dans le jeu de données sélectionné, les génomes `monarch` et `ebola` ne contiennent pas de chemins ambigus et donc sont exclus des instances testées dans le chapitre 4.

III

Échafaudage

Dans ce chapitre, nous présentons des résultats sur le problème de ÉCHAFAUDAGE MAXIMUM défini dans la section 2.3. La section 3.1 est dévolue à des résultats de complexité et d'inapproximation. Nous montrerons ensuite un algorithme d'approximation glouton dans la section 3.2 et nous montrerons comment l'utiliser dans une classe de graphes appelée « graphes de clusters connectés » et dans le cas général grâce à une formulation SAT. Enfin, des tests permettant de comparer différentes versions de l'algorithme glouton seront présentés dans la section 3.3.

3.1 Complexité et borne d'inapproximation

Comme nous l'avons vu dans la sous-section 2.3.2, la complexité du problème d'échafaudage a été étudiée en prenant en compte différents paramètres ou en modifiant les contraintes du problème. Nous apporterons notre pierre à l'édifice en montrant un résultat de NP-complétude pour un cas restreint ainsi que deux résultats d'appartenance à des classes d'inapproximation, là aussi pour des cas restreints.

3.1.1 Appartenance à la classe NP-complet

Il a été montré dans [20] que le problème de décision ÉCHAFAUDAGE était NP-complet dans le cas général mais qu'il pouvait être résolu en temps polynomial dans les graphes d'échafaudage où $2\sigma_c + \sigma_p = |M^*|$. Cela est dû au fait que dans une collection de cycles et de chemins alternés solution au problème, chaque chemin alterné est constitué par une seule arête de couplage et chaque cycle alterné est constitué de deux arêtes de couplage (ce qui est la plus petite taille possible pour un cycle alterné). La recherche d'une solution devient alors plus facile car on connaît précisément la taille de chacun des éléments alternés à chercher. Une interrogation naturelle issue de ce ré-

sultat est de savoir si l'on peut augmenter la taille des éléments alternés recherchés sans que le problème bascule dans la classe NP-complet. On répondra négativement à cette interrogation en montrant que si l'on force chaque chemin alterné à être constitué d'une seule arête de couplage et chaque cycle alterné à être constitué par k arêtes de couplage, le problème devient NP-complet. Pour cela, on étudie le problème ÉCHAFAUDAGE en regardant la valeur de la maille alternée du graphe d'échafaudage, qui, rappelons-le, est la taille du plus petit cycle alterné du graphe d'échafaudage. Dans les graphes d'échafaudage pour lesquels $g^* \cdot \sigma_c + \sigma_p = |M^*|$, une collection de σ_c cycles et de σ_p chemins alternés, on a tous les chemins alternés constitués par une seule arête de couplage et tous les cycles alternés constitués par g^* arêtes de couplage. Nous montrons que si $g^* = 3$, alors ÉCHAFAUDAGE est NP-complet en créant une réduction à partir de ENSEMBLE INDÉPENDANT. Pour construire cette réduction, nous avons besoin de restreindre au préalable ENSEMBLE INDÉPENDANT aux graphes sous-cubiques et de maille au moins quatre. Pour cela, nous utiliserons ce résultat de Lozin et Milanič.

Théorème 4 – [65]

Soit $\mathcal{F}$ un ensemble fini de graphes et soit X la classe de graphes planaires sous-cubiques ne contenant aucun graphes de $\mathcal{F}$ comme sous-graphe induit. Si $\mathcal{F}$ ne contient aucun arbre de trois feuilles, alors ENSEMBLE INDÉPENDANT est NP-difficile dans X.

En prenant $\mathcal{F}$ contenant le cycle de trois sommets, nous obtenons que ENSEMBLE INDÉPENDANT est NP-difficile dans les graphes planaires sous-cubiques et de maille au moins quatre. Considérons à présent la construction suivante.

Construction 2

À partir d'un graphe planaire sous-cubique G tel que la maille de G est de valeur au moins quatre, nous produisons un graphe d'échafaudage (G^*, M^*, ω) de la façon suivante :

- pour chaque arête $e_i \in E(G)$, construire une arête de couplage $u_i\overline{u}_i$;
- pour chaque sommet $v_t \in V(G)$, créer un cycle alterné C_{v_t} contenant toutes les arêtes de couplage $u_i\overline{u_i}$ correspondantes aux arêtes incidentes à v_t dans G et rajouter à ce cycle des arêtes de couplage de façon à ce que C_{v_t} soit de longueur trois. Le cycle C_{v_t} est appelé *cycle-sommet*. Formellement :
  - si $deg(v_t) = 1$, soit e_i l'arête incidente à v_t dans G. Introduire les arêtes de couplage $v_t^1\overline{v}_t^1$ et $v_t^2\overline{v}_t^2$ et construire le cycle-sommet $C_{v_t} = \{v_t^1, \overline{v}_t^1, v_t^2, \overline{v}_t^2, u_i, \overline{u}_i\}$;
  - si $deg(v_t) = 2$, soient e_i et e_j les arêtes incidentes à v_t dans G. Introduire l'arête de couplage $v_t^1\overline{v}_t^1$ et construire le cycle-sommet $C_{v_t} = \{v_t^1, \overline{v}_t^1, u_i, \overline{u}_i, u_j, \overline{u}_j\}$;
  - si $deg(v_t) = 3$, soient e_i, e_j et e_k les arêtes incidentes à v_t dans G. Construire le cycle-sommet $C_{v_t} = \{u_i, \overline{u}_i, u_j, \overline{u}_j, u_k, \overline{u}_k\}$.

Notons que le graphe résultant est sous-cubique et biparti. La bipartition est donnée par $P_1 = \{u_i \mid e_i \in E(G)\} \cup \{v_t^1, v_t^2 \mid v_t \in V(G)\}$ et $P_2 = V(G^*) \setminus P_1$. Si le graphe

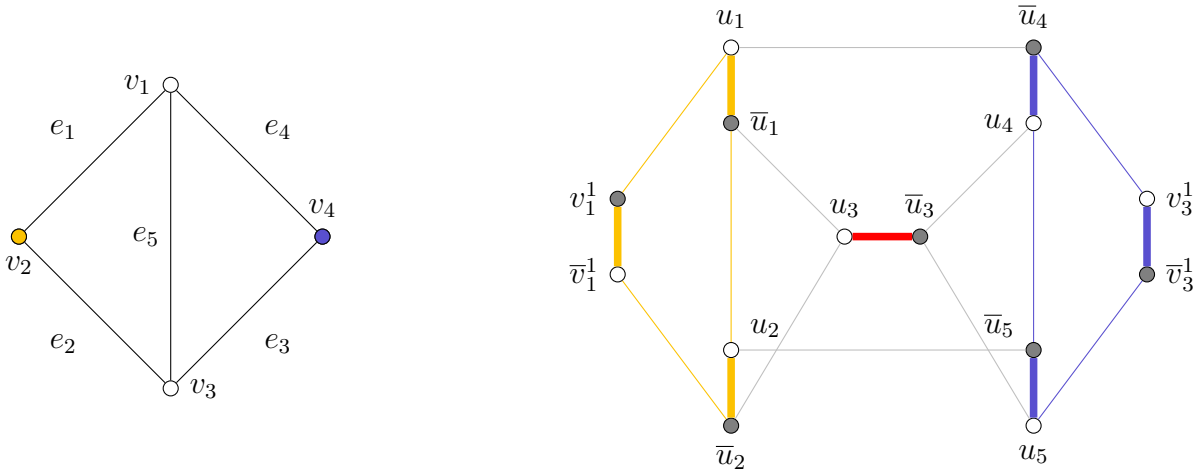


FIGURE 3.1 – *Exemple de graphe d'échafaudage produit par la construction 2.* ***Gauche :*** *graphe donné en entrée de la construction avec un stable contenant deux sommets en jaune et bleu.* ***Droite :*** *graphe résultant avec une collection contenant deux cycles alternés, dessinés en jaune et en bleu et un chemin alterné, dessiné en rouge. Une bipartition est donnée par les sommets gris et blancs. Un exemple de cycle-sommet est le cycle* $C_{v_1} = \{\overline{v}_1^1, v_1^1, \overline{u}_1, u_1, \overline{u}_2, u_2\}$.

d'entrée est planaire, il est possible de rendre la construction planaire en abandonnant la contrainte de bipartition. Un exemple de graphe d'échafaudage résultant de la construction 2 est exposé par la figure 3.1. Un graphe d'échafaudage issu de cette construction possède les propriétés suivantes.

Lemme 3

Soient G un graphe cubique et (G^*, M^*, ω) son graphe d'échafaudage produit par la construction 2. Soit S une solution de ÉCHAFAUDAGE dans (G^*, M^*, ω) avec $\sigma_c = k$ et $\sigma_p = |E(G)| - 3k$. Nous avons les trois propriétés suivantes :

(a) $g^* = 3$;

(b) tout cycle alternant de S est un cycle-sommet ;

(c) soient C_{v_t} et $C_{v_{t'}}$ deux cycle-sommets dans S, les sommets v_t et $v_{t'}$ ne sont pas adjacents dans G.

Preuve.

(a) Par construction, chaque cycle-sommet est composé de trois arêtes de couplage. On a donc $g^* \leq 3$. Supposons qu'il existe un cycle alterné constitué de deux arêtes de couplage e et e'. Les arêtes e et e' appartiennent à des cycle-sommets distincts, sinon on aurait un cycle-sommet composé de deux arêtes de couplage. Appelons ces deux cycle-sommets C_t et $C_{t'}$. Les arêtes de couplage e et e' appartiennent toutes les deux à C_t et $C_{t'}$ sinon elles ne seraient pas adjacentes. Par construction, cela signifie qu'il existe deux arêtes e_i et e_j incidentes à v_t et $v_{t'}$ dans G. Cela constitue une contradiction car il ne peut exister qu'une seule arête entre deux mêmes sommets. Ainsi, il n'existe pas de cycle alterné contenant deux arêtes de couplage dans (G^*, M^*, ω) et on a bien $g^* = 3$.

(b) Par le lemme 3(a), on a $|M^*| = g^* \cdot \sigma_c + \sigma_p$, ce qui implique que tout cycle alterné dans S a une longueur de six. Soit C un cycle de S et soit $u_i\overline{u}_i$ une arête de couplage de C. S'il existe une arête $v_t^k\overline{v}_t^k$ dans G, alors par construction, la dernière arête de couplage de C est l'arête $v_t^{k'}\overline{v}_t^{k'}$, si $deg(v_t) = 1$, ou une arête $u_j\overline{u}_j$ telle que e_j est incidente à v_t dans G. Dans ce cas, C est le cycle-sommet C_{v_t}. Supposons qu'il n'existe pas d'arête de couplage $v_t^k\overline{v}_t^k$ dans C. Alors par contsruction, pour chaque paire d'arête e_i et e_j de G associées aux arêtes de couplage de C, on a e_i et e_j incidentes à un même sommet. Comme G ne contient pas de cycle de longueur trois, alors les trois arêtes de G associées aux arêtes de couplage de C sont incidentes à un même sommet et donc C est un cycle-sommet.

(c) Soit e_i une arête de G reliant les sommets v_t et $v_{t'}$. L'arête de couplage $u_i\overline{u}_i$ appartient aux deux cycle-sommets C_{v_t} et $C_{v_{t'}}$, et donc S ne peut pas contenir à la fois C_{v_t} et $C_{v_{t'}}$.

□

L'idée de la réduction polynomiale est de simuler les sommets du stable par les cycle-sommets. Si on trouve une collection contenant k cycle-sommets, alors comme deux cycle-sommets sont issus de sommets non voisins, il est possible de trouver un stable de taille k. Ce qui amène au résultat suivant.

Théorème 5

ÉCHAFAUDAGE est NP-complet même dans les graphes d'échafaudage (G^*, M^*, ω) où G^* est cubique biparti (ou cubique planaire) et $|M^*| = g^* \cdot \sigma_c + \sigma_p$.

Preuve.

Tout d'abord, comme ÉCHAFAUDAGE a déjà été montré dans NP dans le cas général, il reste à montrer la complétude pour ce cas restreint. Soit G un graphe sous-cubique planaire de maille supérieur à trois et soit (G^*, M^*, ω) son graphe d'échafaudage produit par la construction 2. Ce graphe peut être produit en temps polynomial. Montrons que G possède un stable de taille k si et seulement si il existe une solution positive pour ÉCHAFAUDAGE dans (G^*, M^*, ω) en prenant $\sigma_c = k$ et $\sigma_p = |E(G)| - 3k$.

⇒ Soit I un stable de taille k dans G. On construit la solution S suivante. Pour chaque sommet v_t de I, on crée le cycle alterné C_{v_t}. Puis, pour chaque arête de couplage $u_i\overline{u}_i$ restante, on crée le chemin alterné constitué de cette seule arête. On obtient une solution S constituée de k cycles alternés et $|E(G)| - 3k|$ chemins alternés.

⇐ Soit S une solution dans (G^*, M^*, ω) contenant k cycles alternés et $|E(G)| - 3k$ chemins alternés et soit I l'ensemble de sommets $\{v_t \mid C_{v_t} \in S\}$. Par le lemme 3(b), chaque cycle alterné de S est un cycle-sommet et donc $|I| = k$. De plus, par le lemme 3(c), deux sommets de I ne peuvent pas être voisins. Ainsi I est un stable de taille k.

On a montré qu'il existe une réduction du problème ENSEMBLE INDÉPENDANT vers le problème ÉCHAFAUDAGE et donc ÉCHAFAUDAGE est NP-complet même si $|M^*| = g^* \cdot \sigma_c + \sigma_p$. □

Le résultat précédent a été montré vrai si $g^* = 3$, on peut le prolonger en montrant qu'il est vrai pour toute maille alternée g^* de valeur supérieure à trois. Pour cela, il suffit de modifier la construction précédente en ajoutant dans chaque cycle-sommet un chemin alterné possédant k arêtes de couplage. En faisant cela, la maille alternée du graphe issu de la construction sera égale à $3 + k$. De plus, il est nécessaire que le graphe à partir duquel est effectuée la construction soit de maille au moins $g^* + 1$. En faisant cela, on conserve les propriétés lemme 3(b) et lemme 3(c). Par le théorème 4, en prenant l'ensemble $\mathcal{F}$ contenant tous les cycles de longueur inférieure ou égale à g^*, ENSEMBLE INDÉPENDANT reste NP-difficile dans les graphes de maille au moins $g^* + 1$. Ainsi le théorème 5 reste vrai pour toute valeur de g^* supérieure à trois.

3.1.2 Appartenance à la classe APX-complet

Comme indiqué précédemment, la construction 1 a été utilisée pour montrer que ÉCHAFAUDAGE est NP-complet dans [20]. Ce résultat réduit le problème CHEMIN HAMILTONIEN vers ÉCHAFAUDAGE. On peut modifier cette construction afin d'y intégrer des poids et montrer que le problème ÉCHAFAUDAGE MAXIMUM est APX-complet. Ainsi, nous pourrons faire une S-réduction à partir de VOYAGEUR DE COMMERCE ASYMÉTRIQUE MAXIMUM. Notons que cette réduction a déjà été décrite de façon informelle par Chen et al. [22], mais nous le ferons de façon formelle ici. La construction utilisée est la suivante.

Construction 3

À partir d'une instance de VOYAGEUR DE COMMERCE ASYMÉTRIQUE MAXIMUM (G, c), nous produisons le graphe d'échafaudage (G^*, M^*, ω) suivant :

— pour chaque sommet $v_i \in V(G)$, construire une arête de couplage $v_i^{in} v_i^{out}$;

— pour chaque paire de sommets (v_i, v_j), ajouter l'arête $v_i^{out} v_j^{in}$ avec $w(v_i^{out} v_j^{in}) = c(v_i, v_j)$.

De façon un peu informelle, on peut dire que la construction consiste à remplacer chaque sommet de l'instance de départ par une arête de couplage. Ainsi, si l'on trouve un cycle alterné de poids k passant par toutes les arêtes du couplage du graphe d'échafaudage, on peut trouver un cycle de coût k dans l'instance de départ en contractant toutes les arêtes de couplage en un unique sommet, ce qui constitue une réduction stricte. Comme Papadimitriou et Yannakakis ont montré que VOYAGEUR DE COMMERCE ASYMÉTRIQUE MAXIMUM [71] est APX-difficile, on a le résultat suivant.

Théorème 6

ÉCHAFAUDAGE MAXIMUM est APX-complet dans les graphes complets, même dans le cas où $\sigma_p = 0$ et $\sigma_c = 1$.

La preuve ayant déjà était faite pour démontrer le théorème 1, nous ne la ferons pas à nouveau.

3.1.3 Appartenance à la classe poly-APX-difficile

Dans la partie précédente, on a considéré le cas où le graphe d'échafaudage était un graphe complet. En pratique, les instances réelles ne correspondent pas à des graphes complets. Cependant dans le cas général, comme le problème ÉCHAFAUDAGE est NP-complet, cela implique directement le résultat suivant.

Théorème 7

Il n'existe pas d'algorithme d'approximation en temps polynomial pour ÉCHAFAUDAGE MAXIMUM, sauf si P =NP.

Preuve.

Si un tel algorithme existait, alors cela signifierait qu'il pourrait répondre en temps polynomial au problème de décision ÉCHAFAUDAGE, ce qui est une contradiction. □

Le résultat précédent ne tient pas si l'on se restreint aux classes pour lesquelles la réponse à ÉCHAFAUDAGE peut être calculée en temps polynomial. On peut se demander à quel point ÉCHAFAUDAGE MAXIMUM est approximable dans ce cas. Nous montrons que, dans l'hypothèse où P $\neq$ NP, il n'est pas possible d'approximer ÉCHAFAUDAGE MAXIMUM avec un facteur d'approximation meilleur qu'une fonction polynomiale en la taille de l'instance pour ce type de graphes. En s'inspirant de la construction 2, on peut créer une réduction stricte entre ENSEMBLE INDÉPENDANT MAXIMUM et ÉCHAFAUDAGE MAXIMUM de la façon suivante.

Construction 4

À partir d'un graphe simple G, nous produisons le graphe d'échafaudage (G^*, M^*, ω) suivant.

- Pour chaque arête $e_i = v_t v'_t \in E(G)$, construire un cycle $\{u_i^t, \overline{u}_i^t, u_i^{t'}, \overline{u}_i^{t'}, e_i, \overline{e}_i\}$ avec $u^t\overline{u}_i^t, u^{t'}\overline{u}_i^{t'}, e_i\overline{e}_i \in M^*$.
- Pour chaque sommet v_t, construire une clique $\{v_1^t v_2^t, v_3^t, v_4^t\}$, avec $v_1^t v_2^t, v_3^t v_4^t \in M^*$.
- Pour chaque sommet v_t, soit $\mathcal{A}(v_t)$ la liste des arêtes incidentes à v_t dans G. Créer un cycle alterné contenant tous les sommets $\{v_1^t, \overline{v}_1^t, v_2^t, \overline{v}_2^t\} \cup \{u_i^t, \overline{u}_i^t \mid e_i \in \mathcal{A}(v_t)\}$. Formellement :

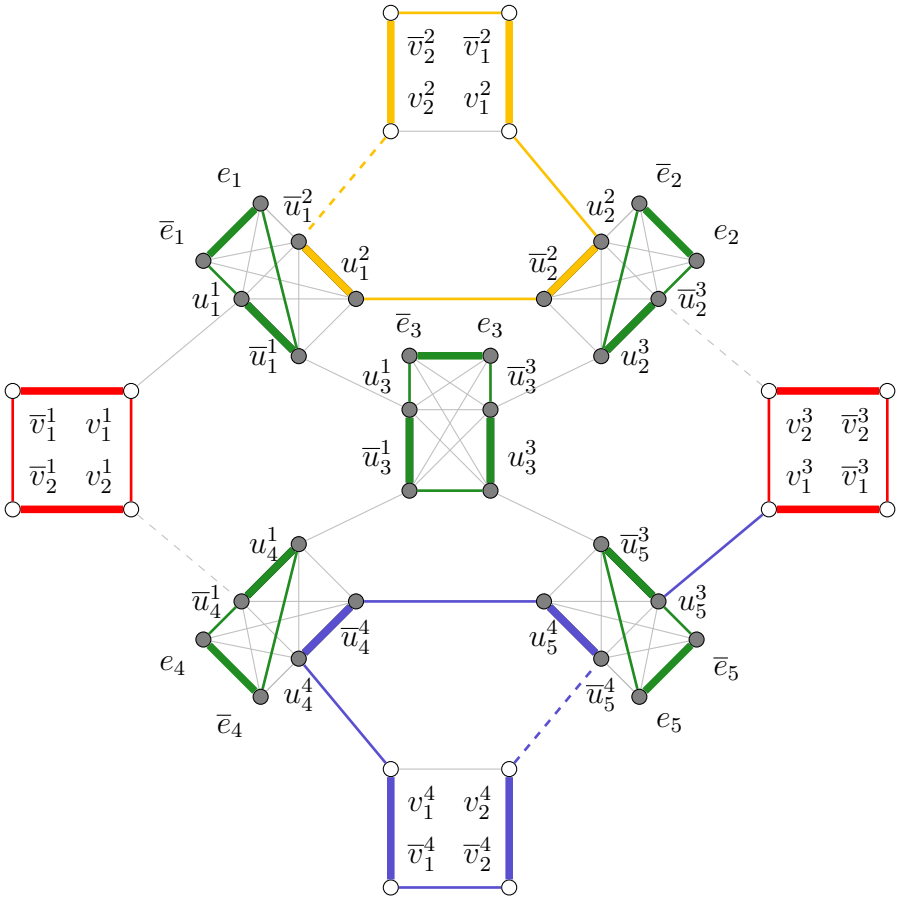


FIGURE 3.2 – *Exemple de graphe d'échafaudage produit par la construction 4. Le graphe donné en entrée est le même que celui de la figure 3.1. Les sommets gris appartiennent aux cliques associées aux arêtes et les sommets blancs appartiennent aux cycles associés aux sommets. Les arêtes de poids un sont dessinées en pointillés. Le long cycle-sommet $C(v_2)$ correspond aux sommets $\{\overline{v}_1^2, v_1^2, u_2^2, \overline{u}_2^2, u_1^2, \overline{u}_1^2, v_2^2, \overline{v}_2^2\}$. La solution dessinée sur la figure correspond au stable contenant les sommets bleu et jaune : cela correspond aux deux longs cycle-sommets bleu et jaune. Les cycles alternés des gadgets arêtes sont dessinées en vert et les cycles alternés des gadgets sommets correspondants aux sommets n'appartenant pas au stable sont dessinés en rouge.*

- pour tout $k < deg(v_t)$, soient e_i et e_j les $k^{ème}$ et $k+1^{ème}$ arêtes de $\mathcal{A}(v_t)$, ajouter une arête entre les sommets $\overline{u}_i^t$ et u_i^t;
- soient e_i et e_j la première et dernière arête de $\mathcal{A}(v_t)$, ajouter les arêtes $v_1^t u_i^t$ et $v_2^t \overline{u}_j^t$.

— La fonction de poids ω est construite de façon suivante. Pour chaque sommet v_t, soit e_j la dernière arête de $\mathcal{A}$, donner à l'arête $v_2^t \overline{u}_j^t$ un poids de un. Pour toutes les autres arêtes non couplantes, donner un poids nul.

Un exemple de graphe d'échafaudage résultant de la construction 4 est exposé par la figure 3.2. Pour chaque sommet v_t de G, on appelle *long cycle-sommet* de v_t le cycle $\{v_1^t, \overline{v}_2^t, v_2^t, \overline{v}_2^t\} \cup \{u_i^t, \overline{u}_i^t \mid e_i = v_t v_{t'} \in E(G)\}$ et l'on dénote ce cycle par $C(v_t)$. On pourra noter qu'un long cycle-sommet a un poids égal à un. Un graphe d'échafaudage résul-

tant de la construction précédente possède les propriétés suivantes.

Lemme 4

Soit G un graphe simple et soit (G^*, M^*, ω) son graphe résultant de la construction 4. Soit S une collection de $|V(G)| + |E(G)|$ cycles alternés dans (G^*, M^*, ω). Les deux propriétés suivantes sont vérifiées.

(a) Soit C un cycle de S, si C n'est pas de poids nul, alors C est un long cycle-sommet.

(b) Soient $C(v_t)$ et $C(v_{t'})$ deux longs cycle-sommets de S, les sommets v_t et $v_{t'}$ ne sont pas adjacents dans G.

Preuve.

(a) Dans un premier temps, montrons qu'il n'existe pas de cycle dans S contenant à la fois une arête de couplage $e_i\overline{e}_i$ et une arête de couplage $v_1^t\overline{v}_1^t$. Supposons par contradiction qu'il existe un cycle C qui ne satisfait pas cette condition. Comme le nombre de cycles dans S est égal à la somme du nombre d'arêtes $e_i\overline{e}_i$ et du nombre d'arêtes $v_1^t\overline{v}_1^t$, cela signifie qu'il existe un cycle alterné C' ne contenant aucune arête $e_i\overline{e}_i$ et aucune arête $v_1^t\overline{v}_1^t$. Soit $u_j^\ell\overline{u}_j^\ell$ une arête de couplage de C', cette arête ne peut pas appartenir à C', sinon l'arête de couplage $e_j\overline{e}_j$ se retrouverait isolée dans S et formerait un chemin alterné. Ainsi, par construction, toutes les arêtes de couplage de C' appartiennent au long cycle-sommet $C(v_\ell)$. Le graphe induit par les sommets de $C(v_\ell) \setminus \{v_1^\ell, \overline{v}_1^\ell\}$ est un chemin ce qui constitue une contradiction, et donc un tel cycle C ne peut pas exister. À présent, supposons, toujours par contradiction, qu'il existe un cycle alterné C dans S de poids non nul qui n'est pas un long cycle-sommet. C contient au moins une arête de poids non nul $v_2^t\overline{u}_i^t$ et donc contient également l'arête de couplage $v_2^t\overline{u}_2^t$. Comme C n'est pas un long cycle-sommet, il existe une arête de couplage $u_j^t\overline{u}_j^t$ appartenant à C telle que l'arête de couplage $u_j^{t'}u_j^{t'}$ appartient aussi à C. Ainsi, soit l'arête $e_j\overline{e}_j$ appartient à C, soit $e_j\overline{e}_j$ est isolée et forme un chemin alterné dans S. Dans les deux cas, on obtient une contradiction. On en conclut qu'un tel cycle C ne peut pas exister et donc que tout cycle alterné de poids non nul dans S est un long cycle-sommet.

(b) Soit $e_i = v_tv_{t'}$ une arête de G. Supposons que S contient à la fois $C(v_t)$ et $C(v_{t'})$. Dans ce cas l'arête de couplage $e_i\overline{e}_i$ se retrouve isolée et S contient un chemin, ce qui est une contradiction avec le fait que S est une solution. Ainsi deux longs cycle-sommets ne peuvent pas appartenir à S si leurs sommets correspondants sont voisins dans G.

□

Nous allons à présent, en utilisant le lemme précédent, montrer l'appartenance à la classe *poly*-APX-difficile pour le problème ÉCHAFAUDAGE MAXIMUM lorsque l'on se restreint aux classes de graphes pour lesquelles le problème ÉCHAFAUDAGE peut être résolu en temps polynomial. Rappelons que le problème ENSEMBLE INDÉPENDANT est *poly*-APX-complet [11]. En utilisant la construction précédente, nous allons créer

une S-réduction à partir de ENSEMBLE INDÉPENDANT et vers ÉCHAFAUDAGE MAXIMUM, ce qui permet de transférer l'appartenance à la classe d'approximation $poly$-APX-difficile.

L'idée est un peu la même que pour la preuve du théorème 5 : si l'on trouve une solution contenant k longs cycle-sommets, alors comme deux longs cycle-sommets sont issus de sommets non voisins, il est possible de trouver un stable de taille k dans le graphe d'entrée. Comme le poids de la solution est égal au nombre de longs cycle-sommets dans la collection, on a bien une S-réduction entre ces deux problèmes.

Théorème 8

ÉCHAFAUDAGE MAXIMUM est poly-APX-difficile même dans les classes de graphes où ÉCHAFAUDAGE peut être résolu en temps polynomial.

Preuve.

Soit G un graphe et soit (G^*, M^*, ω) son graphe produit par la construction 4. Cette construction s'obtient clairement en temps polynomial. Montrons qu'elle constitue une S-réduction en prouvant que G contient un stable de taille k si et seulement si (G^*, M^*, ω) contient une solution S pour ÉCHAFAUDAGE MAXIMUM de poids k en prenant $\sigma_p = 0$ et $\sigma_c = |V(G)| + |E(G)|$.

- $\Rightarrow$ Soit I un stable de taille k dans G. Construisons une solution S dans (G^*, M^*, ω) :
  - — pour chaque sommet $v_t \in I$, construire le long cycle-sommet $C(v_t)$ dans S ;
  - — pour chaque sommet $v_t \in V(G) \setminus I$, construire le cycle $\{v_1^t \overline{v}_1^t, v_2^t, \overline{v}_2^t\}$ dans S ;
  - — pour chaque arête $e_i = v_t v_{t'}$, si e_i n'est pas incidente à un sommet de I, construire le cycle $\{u_i^t, \overline{u}_i^t, u_i^{t'}, \overline{u}_i^{t'}, e_i, \overline{e}_i\}$.

  Comme chaque long cycle-sommet a un poids de un, on obtient bien une solution S avec $\omega(S) = k$.
- $\Leftarrow$ Soit S une solution dans (G^*, M^*, ω) telle que $\omega(S) = k$. Soit I l'ensemble des sommets $\{v_t \mid C(v_t) \in S\}$ (*i.e* l'ensemble des sommets dont le long sommet-cycle correspondant est dans S). Par le lemme 4(a) on a $|I| = k$ et par le lemme 4(b), on s'assure qu'il n'existe pas deux sommets adjacents dans I. On a ainsi construit un stable de taille k.

On a construit une S-réduction de ENSEMBLE INDÉPENDANT MAXIMUM vers ÉCHAFAUDAGE MAXIMUM. Comme cette réduction préserve l'appartenance à la classe poly-APX, alors ÉCHAFAUDAGE MAXIMUM est poly-APX-difficile, même pour les classes de graphes où l'on peut trouver une solution pour ÉCHAFAUDAGE en temps polynomial. □

Soit ω_{max} la valeur maximum des poids pour une instance (G^*, M^*, ω) et soient S_{opt} une solution optimale et S n'importe quelle solution non nulle de (G^*, M^*, ω). On peut remarquer que $\omega S_{opt} \leq \omega_{max} \cdot \omega(S)$. Mais cela n'est pas un facteur d'approximation po-

lynomial en la taille de l'instance. En effet, le poids des arêtes n'est pas borné et donc peut être arbitrairement grand. Pour encoder le poids des arêtes, nous avons besoin d'un espace mémoire de taille logarithmique par rapport à leurs valeurs ce qui implique que la valeur ω_{max} n'est pas proportionnelle à la taille de l'instance. Ainsi, on ne peut pas conclure de cette façon que le problème ÉCHAFAUDAGE MAXIMUM est poly-APX-complet dans le cas où une solution faisable est calculable en temps polynomial.

On peut dériver quelques résultats du théorème 8. Tout d'abord, on peut retrouver indépendamment un résultat qui avait été montré dans [85] montrant que ÉCHAFAUDAGE MAXIMUM était $W[1]$-difficile lorsqu'il était paramétré par le poids de la solution. Comme ENSEMBLE INDÉPENDANT est $W[1]$-complet lorsqu'on le paramètre par la taille du stable et comme la taille du stable est égale au poids d'une solution dans la construction précédente (c'est-à-dire que les deux paramètres sont égaux), on a donc créé une FPT-réduction. On a donc directement le résultat suivant.

Corollaire 1

Il est $W[1]$-difficile de trouver une solution de ÉCHAFAUDAGE MAXIMUM ayant un poids supérieur ou égal au paramètre k.

On sait également que selon l'hypothèse, ETH, ENSEMBLE INDÉPENDANT n'admet pas d'algorithme sous-exponentiel en fonction du nombre d'arêtes du graphe [46]. La taille de la construction est linéaire en fonction du nombre d'arêtes du graphe de départ : on ajoute au plus quatorze sommets et au plus vingt-sept arêtes à la construction par arête du graphe original. Cela implique le résultat suivant.

Corollaire 2

Il n'existe pas d'algorithme permettant de résoudre ÉCHAFAUDAGE MAXIMUM en temps $\mathcal{O}(2^{o(|E(G^*|))})$ ou $\mathcal{O}(2^{o(|V(G^*|))})$ pour tout graphe d'échafaudage (G^*, M^*, ω), sauf si ETH est fausse.

3.2 Approximation

Dans la section précédente, nous avons établi des bornes inférieures d'approximation sous l'hypothèse P $\neq$ NP. Nous nous intéressons à présent aux résultats positifs d'approximation, c'est-à-dire à établir des bornes supérieures d'approximation pour le problème de l'échafaudage. Précisément, nous étudierons un algorithme d'approximation glouton sur une classe particulière de graphes d'échafaudage et également dans le cas général.

3.2.1 Algorithme glouton

L'algorithme d'approximation glouton est décrit par l'algorithme 6. Le principe de cet algorithme est d'évaluer les arêtes par ordre décroissant de poids et de les ajouter au fur et à mesure dans une solution partielle S, (*i.e.* une solution « en construction »). L'instruction centrale de cet algorithme est la fonction FAISABILITÉ (lignes 4 et 7). Cette fonction, étant donné un graphe d'échafaudage et une solution partielle S indique s'il est possible de construire une collection contenant σ_p chemins alternés et σ_c cycles alternés contenant les arêtes de S. Si ce n'est pas le cas, alors la dernière arête considérée est enlevée de S. Si une fonction de faisabilité est fournie, l'algorithme renvoie une solution approchée (si une telle solution existe) à la fin de son exécution, ce qui est décrit par la propriété suivante.

Propriété 6

Soit f une fonction de FAISABILITÉ ayant une complexité en temps égale à $\mathcal{O}(t)$. Si c'est possible, l'algorithme 6 renvoie une solution approchée pour ÉCHAFAUDAGE MAXIMUM avec pour complexité en temps $\mathcal{O}(t \times |E(G^*)| \times \sigma_c^2)$.

Preuve.

Si ce n'est pas possible de construire une solution dans le graphe d'échafaudage donné en entrée, alors le premier appel de la fonction FAISABILITÉ renvoie FAUX et l'algorithme s'arrête. Sinon, à chaque fois qu'une arête e est ajoutée à la solution partielle S, toutes les arêtes incidentes à e et n'appartenant pas au couplage sont supprimées. Ainsi la solution produite est nécessairement constituée de chemins alternés et de cycles alternés. De plus, comme à chaque étape, la condition de faisabilité est vérifiée, quand l'algorithme se termine, il a produit une solution contenant σ_p chemins alternés et σ_c cycles alternés. □

La solution partielle S donnée en entrée de la fonction de FAISABILITÉ est appelée *solution initiante*. Pour pouvoir utiliser cet algorithme sur une certaine classe de graphe, il faut produire une fonction de FAISABILITÉ dédiée à cette classe. Cet algorithme a déjà été étudié sur les graphes complets [20, 27], menant au résultat d'approximation suivant.

Théorème 9 – [20, 27]

L'algorithme 6 produit une solution 3-approchée dans les graphes complets.

Malheureusement, les instances réelles sont assez loin de ressembler à des graphes complets. Pour pouvoir utiliser malgré tout cet algorithme sur ces instances réelles, il est nécessaire de transformer ces instances en graphes complets. Cela se fait par l'ajout d'arête de poids nuls. Cette transformation est définie formellement comme suit.

Définition 24

Soient (G_1^*, M^*, ω_1) et (G_2^*, M^*, ω_2) deux graphes d'échafaudage tels que G_1^* est un sous-graphe partiel de G_2^*. (G_1^*, M^*, ω_2) est *complété* de (G_1^*, M^*, ω_1) si pour toute arête non couplante $e \in E(G_2^*) \setminus M^*$, on a $\omega_2(e) = \omega_1(e)$ si $e \in E(G_1^*)$ et $\omega_2(e) = 0$ sinon.

Comme nous le verrons dans la section 3.3, on obtient en pratique de bons résultats quand on exécute l'algorithme glouton sur les graphes complets complétés d'instances réelles : c'est-à-dire qu'on obtient des collections possédant un poids proche de la solution optimale. Simplement, la collection renvoyée peut contenir des arêtes qui ont été ajoutées pour compléter le graphe et donc qui n'appartiennent pas à l'instance réelle. Cela fait que, en pratique, les collections calculées ne sont pas vraiment des solutions pour le problème ÉCHAFAUDAGE MAXIMUM. Une idée intéressante derrière l'étude d'autres classes de graphes pour la fonction de faisabilité est que, si le nombre d'arêtes ajoutées pour compléter le graphe est moindre, alors la probabilité qu'une arête de poids nul se retrouve dans la collection calculée semble diminuée.

Algorithme 6 : Algorithme Glouton pour ÉCHAFAUDAGE MAXIMUM

Entrée : Un graphe d'échafaudage (G^*, M^*, ω) et deux entiers σ_p et σ_c.
Sortie : Une collection S contenant σ_p chemins alternés et σ_c cycles alternés ou FAUX si une telle collection ne peut exister.

```
// Phase d'initialisation.
```
$S \leftarrow M^*$;
$E \leftarrow E(G^*) \setminus M^*$;
Trier E par ordre décroissant de poids;
Si $\neg$FAISABILITÉ$(G^*, M^*, S, \sigma_p, \sigma_c)$ **alors**
    retourner FAUX
```
// boucle principale
```
Pour tout $e \in E$ **faire**
    Si FAISABILITÉ$(G^*, M^*, S \cup \{e\}, \sigma_p, \sigma_c)$ **alors**
        $S \leftarrow S \cup \{e\}$;
        $I \leftarrow$ ensemble d'arêtes incidentes à e;
        $E \leftarrow E \setminus I$;
retourner S;

Pour vérifier cette hypothèse, nous étudierons dans la suite de cette section deux fonctions de faisabilité. La première fonction de faisabilité est définie dans une classe de graphes particulière appelée « graphes de clusters connectés ». La seconde fonction de faisabilité est définie dans le cas général à l'aide d'une réduction vers le problème SATISFAISABILITÉ, ce qui en fait une fonction non polynomiale. Notons que de toute façon, comme dans le cas général ÉCHAFAUDAGE est NP-complet, il n'est pas possible de créer une fonction de faisabilité s'exécutant en temps polynomial dans le cas général.

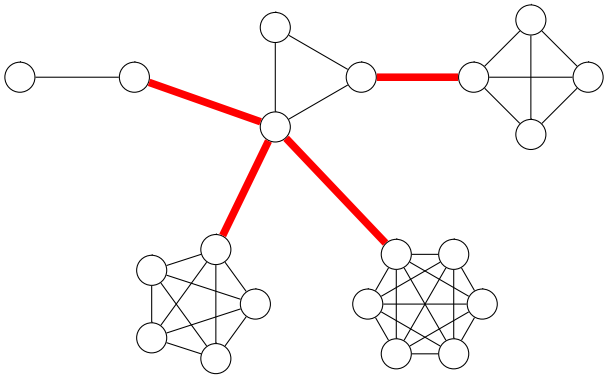


FIGURE 3.3 – *Exemple de graphe de clusters connectés. Les arêtes dessinées en rouge font partie de la partition B.*

3.2.2 Graphes de clusters connectés

3.2.2.1 Définition de la classe

La classe de graphes dans laquelle nous allons travailler est la classe des graphes de clusters connectés qui est définie de la façon suivante.

Définition 25

Un *graphe de clusters connectés* est un graphe G qui admet une décomposition de ses arêtes en deux ensembles E' et B tels que :

- le graphe G' induit par E' est une union disjointe de graphes complets ;
- chaque arête de B est incidente à deux composantes connexes de G' ;
- chaque arête de B est un isthme de G.

Un exemple de graphe de clusters connectés est montré par la figure 3.3. Soit G un graphe de clusters connectés. Par la suite, par simplicité, nous désignerons par le terme « cliques » les composantes connexes du graphe partiel comprenant les arêtes de E'.

La structure d'un graphe de clusters connectés est similaire à celle d'un arbre : si l'on contracte les cliques de G en un seul sommet, on obtient un arbre T, où l'ensemble des sommets $V(T)$ correspond aux cliques de G et l'ensemble des arêtes $E(T)$ correspond à la partition B. Nous utiliserons donc un vocabulaire similaire à celui utilisé pour un arbre. Dans un graphe de clusters connectés *enraciné*, une clique r est désignée comme *racine*. Le *parent* d'une clique c est la clique p adjacente à c dans le chemin de c vers r ; c est alors désignée comme l'*enfant* de p. Une clique sans enfant est appelée *feuille*. On utilisera également un vocabulaire plus spécifique aux graphes de clusters connectés. Un sommet v pour lequel il existe une arête de B incidente est appelé *porte*. La porte v d'une clique c adjacente au parent de c est appelée *porte supérieure* de c. Soit vv' l'arête de couplage incidente à la porte supérieure de c, le sous-graphe induit par $V(c) \setminus \{v, v'\}$ est appelé sous-clique de c. Soit v un sommet de G, on dénote par c_v la clique de G contenant le sommet v. Par la suite, nous nous intéresserons aux graphes d'échafaudage (G^*, M^*, ω) tels que G^* est un graphe de clusters connectés et $B \cap M^* = \varnothing$.

3.2.2.2 Préliminaires

L'algorithme de faisabilité que nous allons construire pour les graphes de clusters connectés est un algorithme dynamique ascendant : cela veut dire que l'on va construire et mettre à jour progressivement des entrées de table en partant des feuilles et en se déplaçant vers la racine. Dans tout le reste de cette sous-section, nous nous référerons à (G^*, M^*, ω) pour désigner le graphe donné en entrée de cette fonction de faisabilité. Une solution partielle est appelée *solution locale* si elle n'est définie que pour un sous-graphe de (G^*, M^*, ω). À noter que même si le terme est semblable à celui utilisé dans l'optique d'une recherche locale, il est utilisé de façon différente ici : la localité provient ici du fait que l'on se restreint à une partie du graphe et non de la proximité avec d'autres solutions. Soit S une solution initiante et S' une solution locale définie dans le sous-graphe G', on dit que S' *contient* S (ou que S est *contenue* dans S') si $S \cap E(G') \subseteq S'$. Nous aurons parfois besoin de créer une solution locale à partir de deux autres solutions locales en utilisant certaines opérations qui sont définies comme suit.

Définition 26

Soit S une solution initiante et soient G_1 et G_2 deux sous-graphes sommets-disjoints de G^*. Soient S_1 et S_2 deux solutions locales dans G_1 et G_2, respectivement. Soit S' une solution locale dans $G^*[V(G_1) \cup V(G_2)]$. S est une *composition* de S_1 et S_2 si exactement une de ses opérations a été effectuée.

Fusion : un chemin alterné de S_1 a été fusionné avec un chemin alterné de S_2. On a alors $S' = S_1 \cup S_2 \cup \{uu'\}$ où u et u' sont des extrémités de chemins alternés dans S_1 et S_2, respectivement.

Fermeture : un chemin alterné de S_1 et un chemin alterné de S_2 ont été fusionnés pour former un cycle alterné. On a alors $S' = S_1 \cup S_2 \cup \{uu', vv'\}$ où u et v sont des extrémités d'un chemin alterné dans S_1 et u' et v' sont des extrémités d'un chemin alterné de S_2.

Absorption : remplacer une arête de S_2 n'appartenant pas à S par un chemin alterné de S_1. On a $S' = S_1 \cup (S_2 \setminus \{vv'\}) \cup \{uv, u'v'\}$ où u et u' sont des extrémités d'un chemin alterné de S_1. On appelle l'arête uv' une *arête absorbante*.

Juxtaposition : S' est l'union disjointe de S_1 et S_2, aucune des opérations précédentes n'est effectuée. On a $S' = S_1 \cup S_2$.

Notons qu'une composition de deux solutions locales n'est pas toujours possible : par exemple dans une solution locale ne contenant que des cycles alternés, aucun chemin alterné ne peut être fusionné avec un autre chemin alterné. Dans l'algorithme présenté ici, nous manipulerons des ensembles de solutions locales : un nouvel ensemble de solutions locales peut être créé à partir de deux autres ensembles $\mathcal{S}_1$ et $\mathcal{S}_2$ en utilisant une certaine opération si pour tous les couples de solutions (S_1, S_2) avec $S_1 \in \mathcal{S}_1$ et $S_2 \in \mathcal{S}_2$, les deux solutions S_1 et S_2 peuvent être combinées avec cette opération.

Définition 27

Soient G_1 et G_2 deux sous-graphes sommets-disjoints de (G^*, M^*, ω) et soient $\mathcal{S}_1$ et $\mathcal{S}_2$ deux ensembles de solutions locales des sous-graphes G_1 et G_2, respectivement. Soit $\mathcal{S}$ un ensemble de solutions locales dans le graphe induit par $V(G_1) \cup V(G_2)$. $\mathcal{S}$ est une *composition complète* de $\mathcal{S}_1$ et $\mathcal{S}_2$ si les deux conditions suivantes sont réunies :

— $\forall S_1 \in \mathcal{S}_1, \forall S_2 \in \mathcal{S}$, la composition de S_1 et S_2 appartient à $\mathcal{S}$;

— $\forall S \in \mathcal{S}, \exists S_1 \in \mathcal{S}_1, \exists S_2 \in \mathcal{S}_2$ tels que S est une composition de S_1 et S_2.

Pour nous assurer de pouvoir construire une composition complète à partir de deux ensembles de solutions locales, il est nécessaire de caractériser une solution locale à partir des opérations que l'on peut lui appliquer. On définira pour cela les types de solutions suivants.

Définition 28

Soit S une solution initiante dans (G^*, M^*, ω). Soient G' un sous-graphe de (G^*, M^*, ω) et S' une solution locale de G'. Soit G'' le sous-graphe induit par les sommets $V(G^*) \setminus V(G')$.

1. S' est *extensible* si S' contient chemin alterné ayant une extrémité v telle que v a un voisin dans G''.
2. S' est *fermable* si S contient un chemin alterné $(u, \ldots, v)$ et qu'il est possible de créer un chemin alterné $(M^*(u), \ldots, M^*(v))$ dans $G'' \cup \{uM^*(u), vM^*(v)\}$.
3. S' est *absorbante* si S' contient un chemin alterné contenant une arête uv n'appartenant pas à S telle que u et v soient voisins de u' et v' et qu'il est possible de créer un chemin alterné $(u', \ldots, v')$ dans G''.
4. S' est appelé *gelée* si aucune des propriétés précédentes n'est vraie.

On peut remarque que toutes les solutions fermables sont également extensibles. Si une solution est extensible, alors on peut y prolonger un chemin alterné avec des arêtes de G''. Si une solution est fermable, alors on peut créer un cycle alterné en ajoutant à un chemin alterné des arêtes de G''. Si une solution est absorbante, alors on peut remplacer une arête absorbante par un chemin alterné. Autrement dit, une solution extensible assure que l'on puisse faire une opération de fusion, une solution fermable assure que l'on puisse faire une opération de fermeture et une solution absorbante assure que l'on puisse faire une opération d'absorption. Ces opérations sont illustrées par la figure 3.4.

Comme le nombre de solutions locales peut être exponentiel, nous n'allons stocker dans les entrées de table calculées que les cardinalités des solutions locales, ce qui est suffisant pour répondre à la question de faisabilité. Nous introduisons à présent la sémantique des entrées de table utilisée dans l'algorithme.

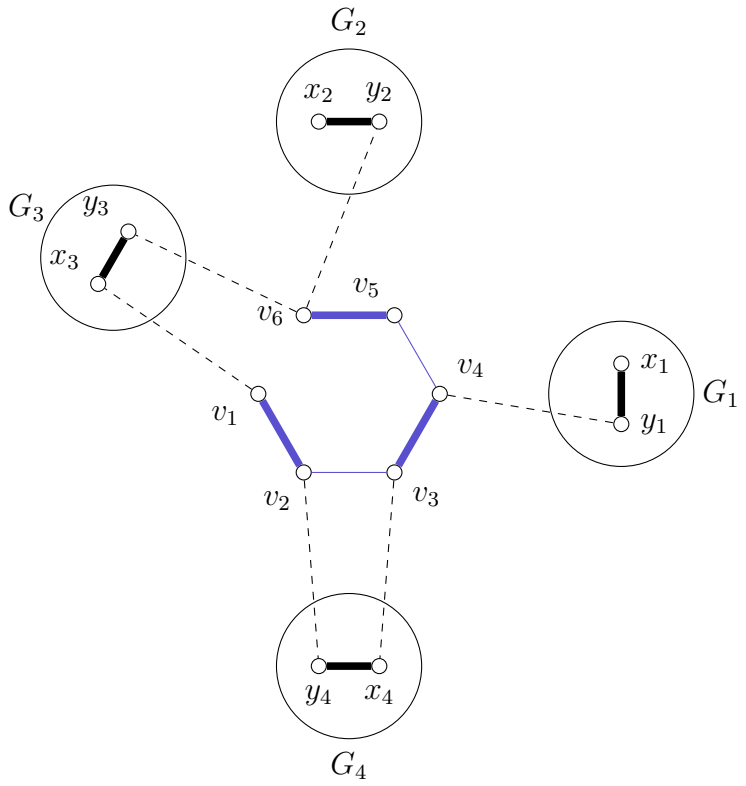


FIGURE 3.4 – *Exemple d'opérations possibles. Soit G' le graphe induit par les sommets $\{v_1, \ldots, v_6\}$. Soit S une solution locale dans G' composée d'un seul chemin alterné, dessiné en bleu. S est extensible dans $G' \cup G_2$: on peut prolonger le chemin alterné en utilisant les arêtes v_6y_2 et y_2x_2. S est fermable dans $G' \cup G_3$: on peut fermer le chemin alterné en utilisant les arêtes v_1x_3, x_3y_3 et y_3v_6. S est absorbante dans $G' \cup G_4$: on peut augmenter la longueur du chemin en enlevant l'arête v_2v_3 et en ajoutant les arêtes v_2y_4, y_4x_4 et x_4v_3 dans S. Finalement, S est gelée dans $G' \cup G_1$.*

Sémantique

Soit $\mathcal{S}$ un ensemble de solutions locales et i un entier naturel. Une entrée de table $[\mathcal{S}, i]$ est l'ensemble des entiers naturels j tels qu'il existe une solution $S \in \mathcal{S}$ avec $\sigma_c(S) = i$ et $\sigma_p(S) = j$. Formellement, soit $S_i = \{S \mid S \in \mathcal{S} \wedge \sigma_c(S) = i\}$, le sous-ensemble de $\mathcal{S}$ contenant toutes les solutions ayant exactement i cycles alternés. On définit $[\mathcal{S}, i] = \bigcup_{S \in \mathcal{S}} \{\sigma_p(S)\}$.

Pour mieux comprendre la sémantique, mettons en avant quelques valeurs particulières pour $[\mathcal{S}, i]$. Pour un ensemble contenant uniquement la solution vide, on a $[\{\varnothing\}, 0] = \{0\}$ et pour tout $i > 0$, on a $[\{\varnothing\}, i] = \varnothing$. Pour une solution S constituée d'un seul chemin alterné, on a $[\{S\}, 0] = \{1\}$ et pour tout $i > 0$, on a $[\{S\}, i] = \varnothing$. Finalement, pour une solution S constituée d'un seul cycle alterné, on a $[\{S\}, 1] = \{0\}$ et pour tout $i \neq 1$, on a $[\{S\}, i] = \varnothing$. Par simplicité, on notera $[\mathcal{S}]$ le vecteur $([\mathcal{S}, 0], \ldots, [\mathcal{S}, \sigma_c])$. Pour récupérer la valeur des cardinalités des solutions locales créées, il suffit de faire des opérations arithmétiques simples à partir des valeurs stockées dans la table. Cela est décrit par le résultat suivant.

Lemme 5

Soient G_1 et G_2 deux sous-graphes sommets-disjoints de (G^*, M^*, ω). Soient $\mathcal{S}_1$ et $\mathcal{S}_2$ deux ensembles de solutions locales de G_1 et G_2, respectivement. Soit $\mathcal{S}$ un ensemble de solutions locales de $G^*[V(G_1) \cup V(G_2)]$ telle que $\mathcal{S}$ est une composition complète de $\mathcal{S}_1$ et $\mathcal{S}_2$.

1. Si $\mathcal{S}$ est l'ensemble des solutions composées avec une opération de fusion, alors
   $\forall i, j, [\mathcal{S}, i + j] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j] + \{-1\}$.
2. Si $\mathcal{S}$ est l'ensemble des solutions composées avec une opération de fermeture, alors
   $\forall i, j [\mathcal{S}, i + j + 1] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j] + \{-2\}$.
3. Si $\mathcal{S}$ est l'ensemble des solutions composées avec une opération d'absorption, alors
   $\forall i, j [\mathcal{S}, i + j] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j] + \{-1\}$.
4. Si $\mathcal{S}$ est l'ensemble des solutions composées avec une opération de juxtaposition, alors
   $\forall i, j [\mathcal{S}, i + j] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j]$.

Preuve.

Dans cette démonstration, pour toute solution $S \in \mathcal{S}$, on dénotera par S_1 et S_2 les solutions locales de respectivement $\mathcal{S}_1$ et $\mathcal{S}_2$, telles que S est composée par S_1 et S_2.

1. et 3. $\forall S \in \mathcal{S}$, comme un seul chemin alterné de S est constitué à la fois d'un chemin alterné de S_1 et d'un chemin alterné de S_2, on a $\sigma_p(S) = \sigma_p(S_1) + \sigma_p(S_2) - 1$. De plus aucun cycle alterné supplémentaire n'est formé par l'opération de fusion ou d'absorption, on a donc $\sigma_c(S) = \sigma_c(S_1) + \sigma_c(S_2)$. Ainsi, on a bien $\forall i, j [\mathcal{S}, i + j] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j] + \{-1\}$.
2. $\forall S \in \mathcal{S}$, comme un seul cycle alterné de S est composé à la fois d'un chemin alterné de S_1 et d'un chemin alterné de S_2, on a $\sigma_c(S) = \sigma_c(S_1) + \sigma_c(S_2) + 1$. Ces deux chemins alternés ne comptent plus comme des chemins alternés dans S, on a donc $\sigma_p(S) = \sigma_p(S_1) + \sigma_p(S_2) - 2$. Ainsi, on a bien $\forall i, j [\mathcal{S}, i + j + 1] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j] + \{-2\}$.
4. $\forall S \in \mathcal{S}$, comme chaque chemin alterné (resp. cycle alterné) de $S_1 \cup S_2$ est également un chemin alterné (resp. cycle alterné) de S, on a $\forall i, j [\mathcal{S}, i + j] = [\mathcal{S}_1, i] + [\mathcal{S}_2, j]$.

□

On utilisera le résultat donné par le lemme 5 pour définir quatre applications $fusion_t$ (algorithme 8), $fermeture_t$ (algorithme 9), $absorption$ (algorithme 8) et $juxtaposition$ (algorithme 7) pour construire les entrées de table. Les entrées de table renvoyées par ces algorithmes correspondent aux compositions complètes d'une opération particulière. Bien que les résultats du lemme 5 ne soient définis que pour deux ensembles de solutions, nous utiliserons des versions généralisées permettant de prendre plus que deux ensembles en entrée. De plus, les fonctions $fusion_t$ et

$fermeture_t$ possèdent un paramètre t supplémentaire, indiquant le nombre de chemins alternés à fusionner ou à fermer. Par exemple, si nous avons trois solutions locales S_1, S_2 et S_3, il est parfois possible de construire un chemin alterné p dans une composition résultante tel que p est constitué d'un chemin alterné de S_1, d'un chemin alterné de S_2 et d'un chemin alterné de S_3. Dans ce cas, nous utiliserons la fonction $fusion_3(\{S_1\}, \{S_2\}, \{S_3\})$ pour indiquer cette opération. On s'autorisera également à fermer un unique chemin alterné en un cycle alterné et pour cela nous nous servirons de la fonction $fermeture_1$. Toutefois, avant d'utiliser une de ces opérations, il faudra s'assurer qu'elle est possible.

Algorithme 7 : Fonction $juxtaposition$

Entrée : $\mathcal{S}^1 = \{\mathcal{S}_1^1, \mathcal{S}_2^1, \ldots\}, \ldots, \mathcal{S}^k = \{\mathcal{S}_1^k, \mathcal{S}_2^k, \ldots\}$: ensembles de solutions locales.

Si $k = 0$ **alors**
  $[\mathcal{S}] \leftarrow 0$;
$[\mathcal{I}] \leftarrow juxtaposition(S^2, \ldots, S^k)$;
Pour tout $i \in [0, \sigma_c]$ **faire**
  Pour tout $j \in [0, \sigma_c - i]$ **faire**
    $[\mathcal{S}, i+j] \leftarrow [\mathcal{S}, i] + \bigcup_{S \in \mathcal{S}^1} [S, j]$
retourner $[\mathcal{S}]$

Algorithme 8 : Fonctions $fusion_t$ ou $absorption$

Entrée : $\mathcal{S}^1 = \{\mathcal{S}_1^1, \mathcal{S}_2^1, \ldots\}, \ldots, \mathcal{S}^k = \{\mathcal{S}_1^k, \mathcal{S}_2^k, \ldots\}$: ensembles de solutions locales et t le nombre de chemins alternés à fusionner ($t = 2$ dans la fonction absorption).

Pour tout $i \in [0, \sigma_c]$ **faire**
  Pour tout $j \in [0, \sigma_c - i]$ **faire**
    $[\mathcal{S}, i+j] \leftarrow \bigcup_{S \in \mathcal{S}^1} [S, i] + \bigcup_{S' \in \mathcal{S}^2} [S', j] + \{-(t-1)\}$
Si $k \neq 2$ **alors**
  $[\mathcal{S}] \leftarrow juxtaposition(\{S\}, S^3, \ldots, S^k)$;
retourner $[\mathcal{S}]$

Algorithme 9 : Fonction $fermeture_t$

Entrée : $\mathcal{S}^1 = \{\mathcal{S}_1^1, \mathcal{S}_2^1, \ldots\}, \ldots, \mathcal{S}^k = \{\mathcal{S}_1^k, \mathcal{S}_2^k, \ldots\}$: ensembles de solutions et t le nombre de chemins alternés à fermer en un cycle.

Pour tout $i \in [0, \sigma_c]$ **faire**
  Pour tout $j \in [0, \sigma_c - i]$ **faire**
    $[\mathcal{S}, i+j+1] \leftarrow \bigcup_{S \in \mathcal{S}^1} [S, i] + \bigcup_{S' \in \mathcal{S}^2} [S', j] + \{-t\}$
Si $k \neq 2$ **alors**
  $[\mathcal{S}] \leftarrow juxtaposition(\{S\}, S^3, \ldots, S^k)$;
retourner $[\mathcal{S}]$

3.2.2.3 L'algorithme

On présente à présent l'algorithme de faisabilité pour les graphes de clusters connectés. À certains moments de l'algorithme, nous aurons besoin de considérer des parties de chemins alternés. Pour cela nous introduisons un nouveau type d'élément alterné : un *élément alterné partiel* est une partie d'un élément alterné contenu dans une unique clique. Soit c une clique et e un élément alterné, les arêtes de $E(c) \cap E(e)$ constituent un élément alterné partiel de c. Notons que e peut contenir plusieurs éléments alternés partiels, s'il est contenu dans plusieurs cliques. Dans l'algorithme de faisabilité, nous traverserons quatre types de sous-graphes différents, pour chaque sommet v, élément alterné partiel e, clique c ou sous-clique c', on définit les sous-graphes $G^*(v), G^*(e), G^*(c)$ ou $G^*(c')$ comme suit.

— Soit v un sommet de G^*, on définit par $enf(v)$ l'ensemble des enfants de $c(v)$ (possiblement vide) adjacents à v. Le sous-graphe $G^*(v)$ est l'union de v et des branches incidentes à v. Formellement, $G^*(v) = G^*[\{v\} \cup \bigcup_{c \in enf(v)} (G^*(c))]$.

— Soit e un élément alterné partiel, le sous-graphe $G^*(e)$ est l'union de e et des branches adjacentes aux sommets de e. Formellement, $G^*(e) = G^*[\bigcup_{v \in V(e)} V(G^*(v))]$.

— Soit c une clique ou une sous-clique de G^*. Le sous-graphe $G^*(c)$ est l'union de c et de toutes les branches aux sommets de c. Formellement $G^*(e) = G^*[\bigcup_{v \in V(c)} V(G^*(v))]$.

Pour chaque sous-graphe traversé, nous utilisons quatre ensembles de solutions locales.

Définition 29

Soit x un sommet, un élément alterné partiel, une sous-clique ou une clique et soit S une solution locale de $G^*(x)$. On définit les ensembles suivants :

— $S \in \mathcal{F}(x)$ si et seulement si S est fermable ;

— $S \in \mathcal{E}(x)$ si et seulement si S est extensible et non fermable ;

— $S \in \mathcal{A}(x)$ si et seulement si S est absorbante ;

— $S \in \mathcal{G}(x)$ si et seulement si S est gelée.

Des exemples illustrant la définition précédente seront distillés au fur et à mesure de la description de l'algorithme. À présent, nous pouvons créer la fonction de faisabilité. Les sous-sections suivantes sont dédiées aux descriptions des algorithmes servant à calculer les entrées de table des quatre types de sous-graphes définis plus haut. On dénote par S la solution initiante donnée en entrée de la fonction de faisabilité, l'algorithme construit ici doit indiquer s'il est possible de construire une solution S' contenant S à la fin de l'algorithme.

3.2.2.4 Sommets

Soit v un sommet de G^*. Dans cette sous-section, nous montrons comment calculer les entrées de table pour les ensembles $\mathcal{G}(v)$ et $\mathcal{E}(v)$, contenant les solutions gelées et

extensibles, respectivement, du sous-graphe $G^*(v)$. Notons que comme l'arête reliant $G^*(v)$ et $G^* \setminus G^*(v)$ est un isthme, il n'existe pas de solutions fermables, ni de solutions absorbables. Il n'est donc pas nécessaire de calculer les entrées de table pour les ensembles $\mathcal{F}(v)$ et $\mathcal{A}(v)$. L'idée est de construire les entrées de table en fusionnant successivement les entrées de table des enfants incidents à v. Pour construire une solution locale gelée dans $G^*(v)$, alors il faut juxtaposer des solutions locales gelées de tous les enfants. Pour construire une solution locale extensible dans $G^*(v)$, alors il faut utiliser une solution locale extensible d'un des enfants et la juxtaposer avec des solutions locales gelées des autres enfants. Si une arête de $S \cap G^*(v)$ est incidente à v, alors l'enfant dans lequel la solution est extensible est imposée par S. De plus, dans ce cas, toutes les solutions locales dans $G^*(v)$ contenant S sont extensibles.

Algorithme 10 : Fonction *calculer_sommet*

Entrée : Un graphe d'échafaudage (G^*, M^*), une solution initiante S et un sommet v.

1 $[\mathcal{G}(v)] \leftarrow \varnothing$; $[\mathcal{E}(v)] \leftarrow \varnothing$; $[\mathcal{G}(v), 0] \leftarrow \{0\}$;
2 $enf \leftarrow \{c_1, \ldots, c_k\}$: liste des enfants reliés à v;
3 **Pour tout** $c_t \in C(v)$ **faire**
4 $\quad calculer\_clique(c_t)$;
5 $\quad [\mathcal{G}'] \leftarrow [\mathcal{G}(v)]$;
6 $\quad [\mathcal{E}'] \leftarrow [\mathcal{E}(v)]$;
7 $\quad$ **Si** $\exists uv \in E(G^*(v)) \cap S$ **alors**
8 $\quad\quad$ **Si** $u \in c_t$ **alors**
9 $\quad\quad\quad [\mathcal{E}(v)] \leftarrow fusion_1(\{\mathcal{E}'\}, \{\mathcal{E}(c_t)\})$
10 $\quad\quad$ **sinon**
11 $\quad\quad\quad [\mathcal{E}(v)] \leftarrow juxtaposition(\{\mathcal{E}'\}, \{\mathcal{G}(c_t), \mathcal{E}(c_t)\})$
12 $\quad$ **sinon**
13 $\quad\quad [\mathcal{G}(v)] \leftarrow juxtaposition(\{\mathcal{G}'\}, \{\mathcal{G}(c_t), \mathcal{E}(c_t)\})$
14 $\quad\quad [\mathcal{E}(v)] \leftarrow juxtaposition(\{\mathcal{E}'\}, \{\mathcal{G}(c_t), \mathcal{E}(c_t)\}) \cup fusion_1(\{\mathcal{G}'\}, \{\mathcal{E}(c_t)\})$

Lemme 6

Pour tout sommet v, les valeurs des entrées de table calculées par l'algorithme 10 sont correctes.

Preuve.

Tout d'abord, s'il n'existe pas d'enfants reliés à v, $G^*(v)$ est constitué du seul sommet v. Dans ce cas, l'unique solution locale possible est la solution vide qui est gelée. Les assignations initiales de la première ligne sont correctes. Supposons que les valeurs calculées par la fonction *calculer_clique* sont correctes. Soit S' une solution locale de $G^*(v)$ contenant S, distinguons deux cas.

1er cas : il existe une arête $uv \in S \cap G^*(v)$. Dans ce cas, S' est extensible. La solution locale S' est composée par la fusion d'une solution locale extensible

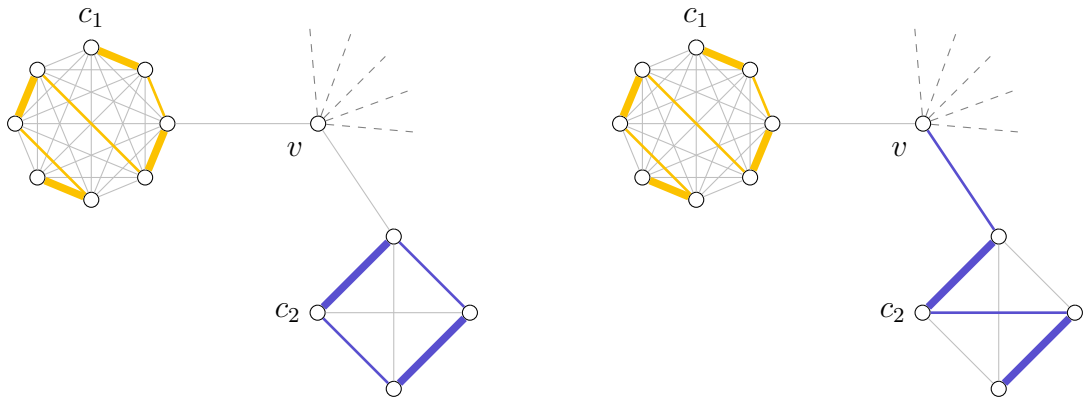


FIGURE 3.5 – *Deux exemples de solutions locales dans un sous-graphe $G^*(v)$ où v est un sommet. Les deux cliques c_1 et c_2 sont les enfants de $c(v)$. Les deux solutions locales contiennent deux éléments alternés dessinés en bleu et jaune. La solution locale de gauche est gelée et appartient donc à $\mathcal{G}(v)$. La solution locale de droite est extensible car elle contient une arête incidente à v et donc appartient à $\mathcal{E}(v)$. Le chemin alterné bleu de cette solution locale pourra être prolongé dans $c(v)$.*

TABLE 3.1 – *Valeurs des entrées de table correspondantes au graphe de la figure 3.5.*

Nombre de cycles	Nombres de chemins					
	$\mathcal{G}(G^*(c_1))$	$\mathcal{E}(G^*(c_1))$	$\mathcal{G}(G^*(c_2))$	$\mathcal{E}(G^*(c_2))$	$\mathcal{G}(G^*(v))$	$\mathcal{E}(G^*(v))$
0	$\{1,2,3\}$	$\{1,\dots,4\}$	$\{1\}$	$\{1,2\}$	$\{2,\dots,6\}$	$\{2,\dots,6\}$
1	$\{0,1,2\}$	$\{1,2\}$	$\{0\}$	$\varnothing$	$\{2,\dots,4\}$	$\{2,\dots,4\}$
2	$\{0\}$	$\varnothing$	$\varnothing$	$\varnothing$	$\{0,\dots,2\}$	$\{1,2\}$
3	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\{0\}$	$\varnothing$

de $G^*(c_u)$ et des juxtapositions de n'importe quelle solution locale dans les autres enfants reliés à v. Ainsi, les assignations des lignes 9 et 11 sont correctes.

2nd cas : il n'existe pas d'arête $uv \in S \cap G^*(v)$. S' est gelée si et seulement si elle ne contient pas d'arête incidente à v. Comme pour tout enfant c_t de relié à v, aucune arête incidente à v n'existe dans $G^*(c_t)$, S' est composée par les juxtapositions de n'importe quelle solution pour chaque enfant, l'assignation ligne 13 est donc correcte. Si S' est extensible, alors il existe un unique enfant c_t tel qu'un chemin alterné de $S' \cap G^*(c_t)$ a été prolongé jusqu'à v, et donc la solution locale de $G^*(c_t)$ composant S' est extensible. Ainsi S' est composée de la fusion d'une solution locale extensible d'un unique enfant et des juxtapositions de n'importe quelle solution locale pour chacun des autres enfants, et donc l'assignation ligne 14 est correcte.

□

Deux exemples de solutions locales extensibles et gelées sont données par figure 3.5 et les valeurs des entrées de table pour ces exemples sont exposées dans le tableau 3.1.

3.2.2.5 Élément alterné partiel

Soit c une clique et soit e un élément alternant de c ne contenant pas la porte supérieure de c. Dans ce paragraphe, nous montrons comment calculer les entrées de table pour les ensembles $\mathcal{E}(e), \mathcal{F}(e)$ et $\mathcal{G}(e)$, contenant les solutions locales extensibles, fermables et gelées, respectivement, du sous-graphe $\mathcal{G}(e)$. L'idée de cet algorithme est de juxtaposer les solutions gelées des sommets internes puis de fusionner les entrées de table obtenues avec les entrées de table des extrémités de e.

Algorithme 11 : Fonction $calculer\_élément\_alterné\_partiel$

Entrée : Un graphe d'échafaudage (G^*, M^*), une solution initiante S et un élément alterné partiel e ayant pour sommets $\{v_0, v_1, \ldots, v_k\}$.

Pour tous $v \in p$ **faire** $calculer\_sommet(v)$;
Si *e est un cycle alterné* **alors**
  $[\mathcal{G}(e)] \leftarrow juxtaposition(\{e\}, \{\mathcal{G}(v_0)\}, \ldots, \{\mathcal{G}(v_k)\})$;
  $[\mathcal{F}(e)] \leftarrow \varnothing$;$[\mathcal{A}(e)] \leftarrow \varnothing$; $[\mathcal{E}(e)] \leftarrow \varnothing$;
sinon
  $[\mathcal{I}_e] \leftarrow juxtaposition(\{e\}, \{\mathcal{G}(v_1)\}, \ldots, \{\mathcal{G}(v_{k-1})\})$;
  $[\mathcal{F}(e)] \leftarrow juxtaposition(\{\mathcal{G}(v_0)\}, \{\mathcal{G}(v_k)\}, \{\mathcal{I}_e\})$;
  $[\mathcal{G}(e)] \leftarrow fusion_3(\{\mathcal{E}(v_0)\}, \{\mathcal{E}_v(v_k)\}, \{\mathcal{I}_e\})$;
  $\cup\ fermeture_1(\{\mathcal{G}(v_0)\}, \{\mathcal{G}(v_k)\}, \{\mathcal{I}_e\})$;
  $[\mathcal{E}(e)] \leftarrow fusion_2(\{\mathcal{E}(u)\}, \{\mathcal{G}(v)\}, \{\mathcal{I}_e\})$;
  $\cup\ fusion_2(\{\mathcal{G}(u)\}, \{\mathcal{E}(v)\}, \{\mathcal{I}_e\})$;

Notons qu'il est possible de créer une solution locale absorbante S' de $G^*(e)$ lorsque e est un chemin alterné, en fermant celui-ci. Cela pourrait permettre par la suite d'absorber un chemin alterné e' dans la clique contenant e. Cependant la solution résultante pourrait également être obtenue en laissant le chemin alterné e dans une solution fermable et en effectuant ensuite une opération de fermeture impliquant les deux chemins alternés e et e'. Pour éviter cette redondance, la valeur de l'entrée $[\mathcal{A}(e)]$ n'est pas calculée (et n'est pas utilisée par la suite). On considérera que la fermeture d'un chemin alterné en un cycle produit une solution gelée.

Lemme 7

Pour tout élément alterné partiel e, les valeurs des entrées de table calculées par algorithme 11 sont correctes.

Preuve.

Supposons que les valeurs calculées par la fonction $calculer\_sommet$ sont correctes. Soit S' une solution locale de $G^*(e)$ contenant S. Dans un premier temps, notons que pour chaque sommet interne v de e, les solutions locales composant

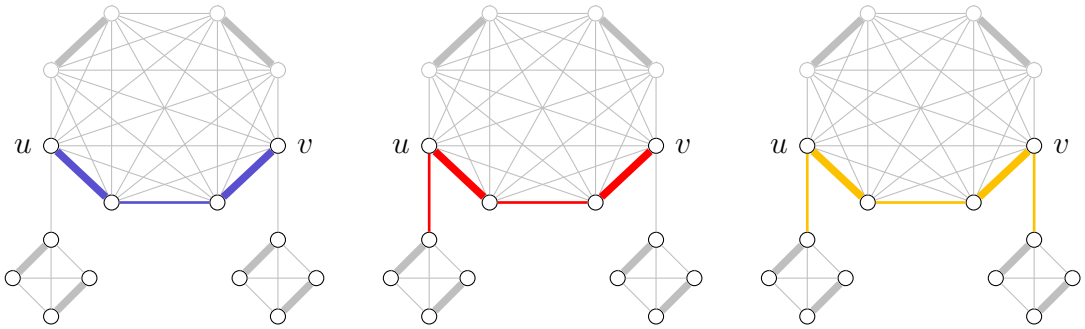


FIGURE 3.6 – *Trois exemples de solutions locales dans un sous-graphe $G^*(p)$ où p est un chemin alterné entre u et v. À gauche, une solution locale dessinée en bleu est fermable (et extensible). Au centre, une solution locale dessinée en rouge est extensible. À droite, une solution locale dessinée en jaune est gelée. Dans les solutions locales en bleu et en rouge, on peut étendre le chemin alterné dans la clique en ajoutant une arête de la clique à une des extrémités. Dans la solution locale bleue, le chemin alterné peut être refermé en un cycle alterné en joignant les deux extrémités u et v avec des arêtes de la clique, dans ce cas la solution est gelée.*

S' dans $G^*(v)$ sont gelées. La solution locale S' contient donc les juxtapositions des solutions locales gelées des sommets internes de e. Dans le cas d'un cycle alterné, e ne contient que des sommets internes et S' est gelée et donc S' est composée des juxtapositions de solutions locales gelées des sommets de e et du cycle e. Comme dans ce cas toutes les solutions dans $G^*(e)$ sont gelées, les assignations lignes 3 et 4 sont correctes. Traitons à présent le cas où e est un chemin alterné interne. Soit I_e l'ensemble des solutions locales composées des juxtapositions de e avec de solutions locales gelées des sommets internes de e.

— S' est fermable si et seulement si les degrés des extrémités v_0 et v_k sont égaux à un. Dans ce cas les solutions locales composant S' dans $G^*(v_0)$ et $G^*(v_k)$ sont gelées et donc l'assignation ligne 7 est correcte.

— S' est gelée si et seulement si les degrés des extrémités v_0 et v_k sont égaux à deux. C'est le cas si (1) les solutions locales composant S' dans $G^*(v_0)$ et $G^*(v_k)$ sont extensibles ou si (2) les solutions locales composant S' dans $G^*(v_0)$ et $G^*(v_k)$ sont gelées et e est refermé en un cycle. Ainsi, l'assignation ligne 8 est correcte.

— Une solution est extensible et non fermable si exactement une des extrémités v_0 ou v_k a un degré un. Soient S_0 et S_k les solutions locales composant S dans $G^*(v_0)$ et $G^*(v_k)$, respectivement. Exactement une solution locale parmi S_0 et S_k est extensible. Ainsi l'assignation ligne 9 est correcte.

□

Des exemples de solutions locales extensibles, gelées ou fermables sont données par figure 3.6 et les valeurs des entrées de table pour ces exemples sont exposées dans le tableau 3.2.

TABLE 3.2 – *Valeurs des entrées de table correspondantes au graphe de la figure 3.6.*

Nombre de cycles	Nombres de chemins		
	$\mathcal{E}(G^*(v))$	$\mathcal{G}(G^*(p))$	$\mathcal{F}(G^*(p))$
0	$\{2,\ldots,4\}$	$\{1,\ldots,3\}$	$\{3,\ldots,5\}$
1	$\{1,2\}$	$\varnothing$	$\{1,\ldots,3\}$
2	$\varnothing$	$\varnothing$	$\{1\}$

3.2.2.6 Sous-Clique

Soit c' une sous-clique de G^*. Nous montrons dans cette partie comment calculer les entrées de table pour les ensembles $\mathcal{F}(c'), \mathcal{G}(c'), \mathcal{A}(c')$ et $\mathcal{E}(c')$ comprenant les solutions locales fermables, gelées, absorbantes et extensibles, respectivement, du sous-graphe $G^*(c')$. Les valeurs des entrées de table sont calculées en parcourant les éléments alternés partiels et en fusionnant successivement leurs entrées de table. Pour cela nous utiliserons à chaque étape t un graphe intermédiaire G_t et deux ensembles intermédiaires $\mathcal{A}_+$ et $\mathcal{E}_+$. Le graphe G_i est défini récursivement de la façon suivante. G_0 est le graphe vide. Soit e_i l'élément alterné partiel considéré à l'étape i, on a $G_i = G^*[V(G_{i-1}) \cup V(G^*(e_i))]$, c'est-à-dire le sous-graphe induit par les sommets des branches déjà parcourues. À chaque étape t, une solution locale S' de G_i appartient à $\mathcal{A}_+$ (resp. $\mathcal{E}_+$) si et seulement si les deux conditions suivantes sont réunies :

— S' contient un ensemble C non nul de chemins alternés fermables ;

— $S' \setminus C$ n'est pas extensible (resp. est extensible) ;

Ces deux ensembles contiennent donc des solutions qui sont fermables. La raison pour laquelle on n'utilise pas l'ensemble $\mathcal{F}(c')$ dans le graphe intermédiaire est la suivante : dans une solution fermable contenant un unique chemin alterné fermable, en effectuant une opération de fermeture sur celle-ci, la solution produite peut être soit absorbante, soit extensible (voir la figure 3.7). On a donc besoin d'une information supplémentaire afin de connaître l'ensemble auquel appartient cette solution produite.

Lemme 8

Pour toute sous-clique c', les valeurs des entrées de table calculées par algorithme 12 sont correctes.

Preuve.

Supposons que les valeurs des entrées calculées par la fonction *calculer_élément_alterné_partiel* sont correctes. Montrons par récurrence que les valeurs calculées à chaque étape t sont correctes pour le graphe G_t. Tout d'abord, G_0 est un graphe vide, et donc l'unique solution possible est celle ne contenant aucun chemin ou cycle alterné et cette solution est gelée. Les assignations des lignes 1 à 3 sont donc correctes. À présent, considérons l'élément alterné partiel considéré à l'étape t et supposons par hypothèse de récurrence que les valeurs calculées aux étapes précédentes sont correctes. Les valeurs calculées aux étapes précédentes sont enregistrées dans les entrées $\mathcal{G}', \mathcal{A}', \mathcal{E}', \mathcal{G}'_+, \mathcal{A}'_+, \mathcal{E}'_+$. Soient S_1 une solution locale dans G_{t-1}, S_2 une solution locale dans $G^*(e_t)$ et S' une composition de S_1 et S_2. On a les propriétés suivantes

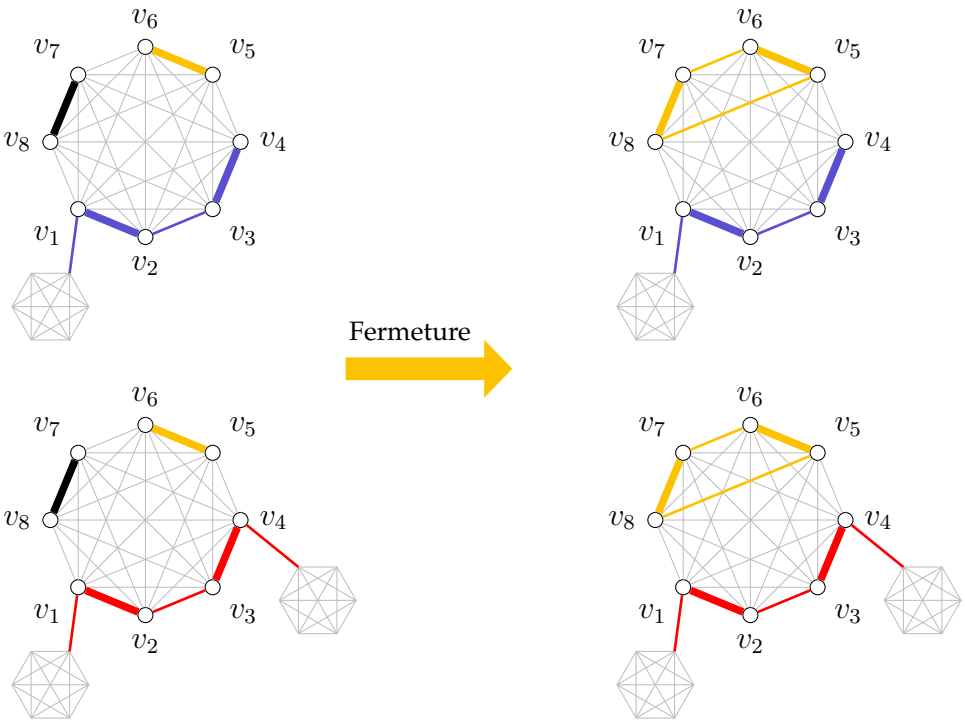


FIGURE 3.7 – *Exemple de deux solutions fermables (**gauche**) et deux solutions résultantes avec une opération de fermeture (**droite**) dans une sous-clique $c = \{v_1, \ldots, v_8\}$. À gauche, pour les deux solutions, le chemin alterné en jaune est fermable et après l'opération de fermeture, à droite, ce chemin alterné a été refermé en cycle alterné avec l'arête de couplage v_7v_8. Pour la solution fermable du haut, le chemin partiel dessiné en bleu est extensible et pour la solution fermable du bas, le chemin alterné partiel dessiné en rouge est gelé. Ainsi, la solution résultante en haut est extensible tandis que la solution résultante en bas est gelée.*

:

- si S' est obtenue par une opération de juxtaposition, alors S_1 appartient à $\mathcal{G}', \mathcal{A}', \mathcal{E}', \mathcal{A}'_+$ ou $\mathcal{E}'_+$ et S_2 appartient à $\mathcal{F}(e_t), \mathcal{G}(e_t)$ ou $\mathcal{E}(e_t)$;
- si S' est obtenue par une opération de fusion, alors S_1 appartient à $\mathcal{E}', \mathcal{A}'_+$ ou $\mathcal{E}'_+$ et S_2 appartient à $\mathcal{F}(e_t)$ ou $\mathcal{E}(e_t)$;
- si S' est obtenue par une opération d'absorption, alors S_1 appartient à $\mathcal{A}'$ ou $\mathcal{A}'_+$ et S_2 appartient à $\mathcal{F}(e_t)$;
- si S' est obtenue par une opération de fermeture, alors S_1 appartient à $\mathcal{A}'_+$ ou $\mathcal{E}'_+$ et S_2 appartient à $\mathcal{F}(e_t)$.

Il existe ainsi vingt-cinq compositions complètes à considérer. Si $S_2 \in \mathcal{F}(e_t)$ (resp. $\mathcal{P}(e_t)$) et S' est obtenue par une opération de fermeture (resp. opération de fusion), alors S' est fermable (resp. extensible) si S_1 contient davantage qu'un chemin alterné fermable (resp. extensible) ou S' est absorbante dans le cas contraire. Ainsi, une composition complète obtenue avec une opération de fermeture ou de fusion n'est pas incluse dans un unique ensemble, parmi ceux définis. Cela pose une difficulté pour le calcul des entrées de table, car on ne va pas pouvoir assigner une telle composition complète à un des ensembles. Toutefois, on

Algorithme 12 : Fonction $calculer\_sous\_clique$

Entrée : Un graphe d'échafaudage (G^*, M^*), une solution initiante S et une sous-clique c'.

$[\mathcal{G}(c')] \leftarrow \varnothing$; $[\mathcal{E}(c')] \leftarrow \varnothing$; $[\mathcal{A}(c')] \leftarrow \varnothing$;
$[\mathcal{E}_+] \leftarrow \varnothing$; $[\mathcal{A}_+] \leftarrow \varnothing$;
$[\mathcal{G}(c'), 0] \leftarrow \{0\}$;
$E \leftarrow \{e_1, \ldots, e_k\}$: liste des éléments alternés partiels de c';
Pour tout $e_t \in E$ **faire**
  $calculer\_élément\_alterné\_partiel(e_t)$;
  $[\mathcal{G}'] \leftarrow [\mathcal{G}(c')]$; $[\mathcal{E}'] \leftarrow [\mathcal{E}(c')]$; $[\mathcal{A}'] \leftarrow [\mathcal{A}(c')]$;
  $[\mathcal{E}'_+] \leftarrow [\mathcal{E}_+]$; $[\mathcal{A}'_+] \leftarrow [\mathcal{A}_+]$;

  $[\mathcal{G}(c')] \leftarrow juxtaposition(\{\mathcal{G}'\}, \{\mathcal{G}(e_t)\})$

  $[\mathcal{A}(c')] \leftarrow juxtaposition(\{\mathcal{A}'\}, \{\mathcal{G}(e_t)\})$
  $\cup\ fusion_2(\{\mathcal{E}'\}, \{\mathcal{E}(e_t)\})$
  $\cup\ absorption(\{\mathcal{A}'\}, \{\mathcal{F}(e_t)\})$
  $\cup\ fermeture_2(\{\mathcal{A}'_+\}, \{\mathcal{F}(e_t)\})$

  $[\mathcal{A}_+] \leftarrow juxtaposition(\{\mathcal{G}', \mathcal{A}', \mathcal{A}'_+\}, \{\mathcal{F}(e_t)\})$
  $\cup\ juxtaposition(\{\mathcal{A}'_+\}, \{\mathcal{G}(e_t)\})$
  $\cup\ fusion_2(\{\mathcal{E}'_+\}, \{\mathcal{E}(e_t)\})$
  $fusion_2(\{\mathcal{A}'_+\}, \{\mathcal{F}(e_t)\})$

  $[\mathcal{E}(c')] \leftarrow juxtaposition(\{\mathcal{E}'\}, \{\mathcal{G}(e_t), \mathcal{E}(e_t)\})$
  $\cup\ juxtaposition(\{\mathcal{G}', \mathcal{A}'\}, \{\mathcal{E}(e_t)\})$
  $\cup\ fusion_2(\{\mathcal{E}'_+\}, \{\mathcal{E}(e_t)\})$
  $\cup\ fusion_2(\{\mathcal{E}'\}, \{\mathcal{F}(e_t)\})$
  $\cup\ fermeture_2(\{\mathcal{E}'_+\}, \{\mathcal{F}(e_t)\})$

  $[\mathcal{E}_+] \leftarrow juxtaposition(\{\mathcal{E}'_+\}, \{\mathcal{G}(e_t), \mathcal{E}(e_t), \mathcal{F}(e_t)\})$
  $\cup\ juxtaposition(\{\mathcal{A}'_+\}, \{\mathcal{E}(e_t)\})$
  $\cup\ juxtaposition(\{\mathcal{E}'\}, \{\mathcal{F}(e_t)\})$
$[\mathcal{F}(c')] \leftarrow [\mathcal{A}_+] \cup [\mathcal{E}_+]$

peut résoudre ce problème et également réduire le nombre de compositions complètes à considérer en ignorant certaines d'entre elles : S' peut être ignorée par l'algorithme s'il existe une autre solution locale dans G_t avec la même cardinalité pouvant être obtenue avec une autre composition.

1. Supposons que S' est obtenue par une opération de fermeture (resp. opération de fusion) et S_1 contient plus qu'un chemin alterné fermable (resp. extensible). Soient p_1 et p_2 deux chemins alternés fermables (resp. extensibles) de S_1. Il existe une solution S'_1 similaire à S_1, à l'exception que les chemins alternés p_1 et p_2 ont été fermés en un cycle (resp. fusionnés en un unique chemin alterné) dans une étape précédente. On peut obtenir une solution locale de G_t de même cardinalité que S' en faisant une opération de juxtaposition entre S'_1 et S_2. Ainsi, après une opération de fermeture,

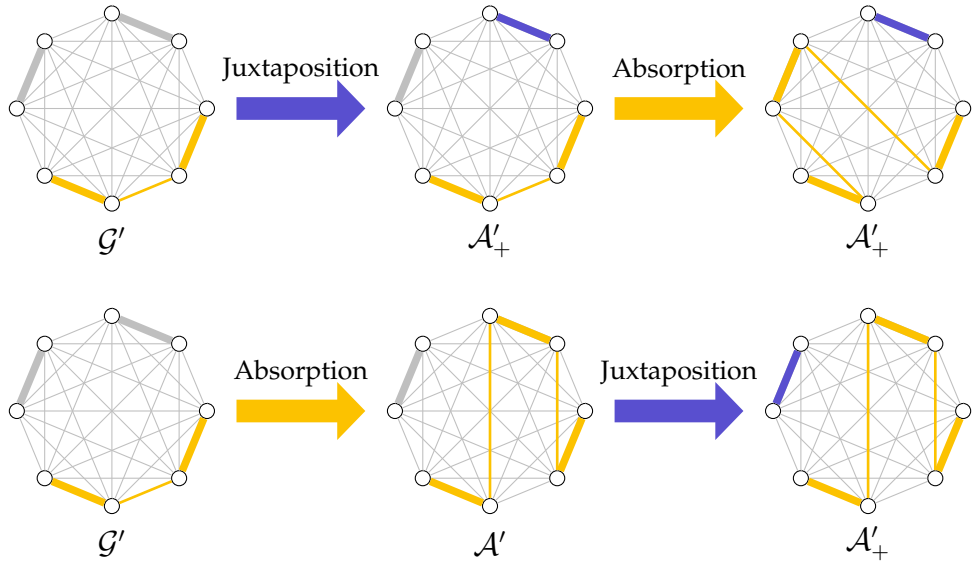


FIGURE 3.8 – *Exemple de cas non considéré par l'algorithme. En haut, la solution obtenue est obtenue après la juxtaposition d'un chemin alterné fermable p_1 et l'absorption d'un chemin alterné fermable p_2. La solution intermédiaire appartient à l'ensemble $\mathcal{A}'_+$ durant la deuxième étape. En bas, une solution de même cardinalité, est obtenue après avoir absorbé p_1 et juxtaposé p_2 et dans ce cas, la solution intermédiaire appartient à $\mathcal{A}'$. La solution du haut n'est pas considérée par l'algorithme car la solution du bas possède la même cardinalité que celle-ci.*

on peut supposer que la solution locale obtenue n'appartient pas aux ensembles $\mathcal{A}_+$ ou $\mathcal{E}_+$. De même, après une opération de fermeture entre une solution locale de $\mathcal{E}'$ ou $\mathcal{E}_+$, et une solution locale de $\mathcal{E}(e_t)$, la solution locale obtenue n'appartient ni à $\mathcal{E}(c')$, ni à $\mathcal{E}_+$.

2. Supposons qu'une des conditions suivantes soit vraie. (1) $S_1 \in \mathcal{A}'_+$, $S_2 \in \mathcal{E}(e_t)$ et S' est obtenue avec une opération de fusion, (2) $S_2 \in \mathcal{E}'_+, S_2 \in \mathcal{F}(e_t)$ et S' est obtenue par une opération de fusion, (3) $S_1 \in \mathcal{A}'_+, S_2 \in \mathcal{F}(e_t)$ et S' est obtenue avec une opération d'absorption. Soit p un chemin alterné fermable de S_1 qui a été absorbé ou fusionné durant l'opération composant S'. Il existe une solution S'_1 similaire à S_1, à l'exception que toutes les arêtes n'appartenant pas au couplage de p ont été absorbées ou fusionnées durant les étapes précédentes. On peut obtenir une solution locale de G_t de même cardinalité que S' en faisant une opération de juxtaposition entre S'_1 et S_2. Ainsi on peut ignorer les trois cas de solutions locales produites par les conditions (1), (2) et (3). Le cas (3) est illustré par la figure 3.8.

Il reste à présent vingt-deux compositions complètes à considérer, chacune de ces compositions complètes n'appartient qu'à un seul des six ensembles de solutions locales parmi $\mathcal{G}(c'), \mathcal{A}(c'), \mathcal{E}(c'), \mathcal{G}'_+, \mathcal{A}'_+$ et $\mathcal{E}'_+$.

— S' est gelée. Dans ce cas, la seule opération possible pour obtenir S' est la juxtaposition, car l'ajout d'une arête n'appartenant pas à la solution initiante de la sous-clique dans S' rendrait la solution absorbante. Les solutions locales S_1 et S_2 sont nécessairement gelées car sinon leur juxtaposition ne serait pas gelée. Ainsi, l'affectation de la ligne 10 est correcte.

— S' est absorbante. Dans ce cas, S' contient au moins une arête de la sous-

clique n'appartenant pas à la solution initiante.

- Si S_2 est gelée, alors comme la seule opération possible est la juxtaposition, S_1 doit être absorbante.
- Si S_2 est extensible, alors une opération de fusion doit être effectuée. Si l'opération est faite à partir d'un chemin alterné fermable, alors la solution résultante est extensible. Ainsi S_1 est également extensible.
- Si S' est obtenue avec une opération d'absorption, alors S_1 est absorbante et S_2 est fermable.
- Si S' est obtenue avec une opération de fermeture, alors S_1 et S_2 sont fermables. Comme la solution résultante est absorbante, S_1 appartient à l'ensemble $\mathcal{A}'_+$.

Ainsi, l'affectation de la ligne 11 est correcte.

- S' appartient à l'ensemble $\mathcal{A}_+$, cela veut dire qu'elle est fermable et qu'elle ne contient pas de chemins alternés extensibles.
  - Si S' est issue d'une juxtaposition, alors S_2 est soit gelée, soit fermable. Dans le premier cas, S_1 doit être fermable mais ne pas contenir de chemin alterné extensible et donc S_1 appartient à $\mathcal{A}'_+$. Dans le second cas, S_1 ne doit pas contenir de chemin alterné extensible et donc S_1 appartient à $\mathcal{G}', \mathcal{A}'$ ou $\mathcal{A}'_+$.
  - Si S' est issue d'une fusion, alors S_1 est fermable et S_2 est soit extensible, soit fermable. Dans le premier cas, le chemin alterné de S_1 doit être fusionné avec un chemin alterné extensible de S_1 pour ne pas que la solution résultante contienne un chemin extensible et donc S_1 appartient à $\mathcal{E}_+$. Dans le second cas, S_1 ne peut pas contenir de chemin alterné extensible car sinon S' en contiendrait aussi et donc S_1 appartient à $\mathcal{A}_+$.

  Ainsi, l'affectation de la ligne 12 est correcte.
- S' est extensible. Dans ce cas, soit S_1 contient un chemin alterné extensible, soit S_2 est extensible.
  - Si S_1 est extensible et S' est issue d'une opération de juxtaposition, alors S_2 ne peut pas être fermable car sinon la solution résultante serait également fermable et donc S_2 est soit gelée, soit extensible.
  - Si S_1 est extensible et S' est issue d'une opération de fusion, alors rappelons que comme indiqué plus haut, on ne considère que les solutions extensibles de $\mathcal{E}'$ contenant un unique chemin alterné extensible. Ainsi, S_2 ne peut pas être extensible car sinon la solution résultante serait absorbante et donc S_2 est fermable.
  - Si S_1 appartient à $\mathcal{E}'_+$, alors comme S' ne doit pas être fermable, le chemin fermable de S_1 doit soit être fusionné avec un chemin alterné extensible, soit être fermé en un cycle avec un chemin alterné fermable. Ainsi, dans le premier cas S_2 est extensible et dans le deuxième cas S_2 est fermable.

TABLE 3.3 – *Valeurs des entrées de table correspondantes au graphe de la figure 3.9. On rappelle qu'une solution initiante comprenant les arêtes* $\{v_3v_{10}, v_4, v_9, v_6v_7\}$ *est donnée.*

Nombre de cycles	Nombres de chemins			
	$\mathcal{E}(G^*(c'))$	$\mathcal{G}(G^*(c'))$	$\mathcal{F}(G^*(c'))$	$\mathcal{A}(G^*(c'))$
0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
1	$\{2,\ldots,6\}$	$\{1,\ldots,5\}$	$\{3,\ldots,7\}$	$\varnothing$
2	$\{1,\ldots,4\}$	$\{1,\ldots,3\}$	$\{2,\ldots,5\}$	$\{2,\ldots,6\}$
3	$\{1,2\}$	$\{1\}$	$\{1,\ldots,3\}$	$\{1,\ldots,4\}$
4	$\varnothing$	$\varnothing$	$\varnothing$	$\{0\}$

— Si S_2 est extensible et que S_1 ne contient aucun chemin alterné extensible ou fermable, alors S est obtenue avec une opération de juxtaposition et dans ce cas S_1 est soit gelée, soit absorbante.

Ainsi, l'affectation ligne 13 est correcte.

— S' appartient à $\mathcal{E}_+$, cela veut dire qu'elle est fermable et qu'elle contient un chemin alterné extensible. Rappelons que comme indiqué précédemment, on ne considère pas la fusion d'une solution de $\mathcal{E}_+$ avec une solution fermable. Ainsi, S' est obtenue avec une juxtaposition et soit S_1, soit S_2 contient un chemin alterné extensible.

— Si S_1 appartient à $\mathcal{E}'_+$, alors S_2 peut être indifféremment gelée, extensible ou fermable.

— Si S_1 appartient à $\mathcal{A}'_+$ ou $\mathcal{E}'_+$, alors pour que S' contienne un chemin alterné extensible, la solution S_2 doit être extensible.

— Si S_1 est extensible, alors pour que S' contienne un chemin alterné fermable, la solution S_2 doit être fermable.

Ainsi, l'affectation de la ligne 14 est correcte.

Comme après ces affectations, chacune des solutions locales de G_t appartient à un seul de ces six ensembles et est une composition d'une solution locale de G_{t-1} et $G^*(e_t)$, cela suffit pour conclure que les valeurs des six entrées de table relatives à ces six ensembles sont correctes pour G_t.
Finalement, après l'exécution de la boucle parcourant les éléments alternés partiels de c', les valeurs des entrées de table des ensembles $\mathcal{G}(c'), \mathcal{A}(c')$ et $\mathcal{E}(c')$ pour le sous-graphe $G_k = G^*(c')$ sont correctes. Il reste à calculer la valeur de l'entrée de table pour l'ensemble $\mathcal{F}(c')$. Comme les ensembles contenant des chemins alternés fermables sont exactement les ensembles $\mathcal{A}_+$ et $\mathcal{E}_+$, il suffit de faire l'union de leurs entrées de table pour obtenir l'entrée de table de $\mathcal{F}(c')$. Ainsi l'affectation de la ligne 15 est correcte. □

Des exemples de solutions locales dans une sous-clique sont données par la figure 3.9 et les valeurs des entrées de table pour ces exemples sont exposées dans le tableau 3.3.

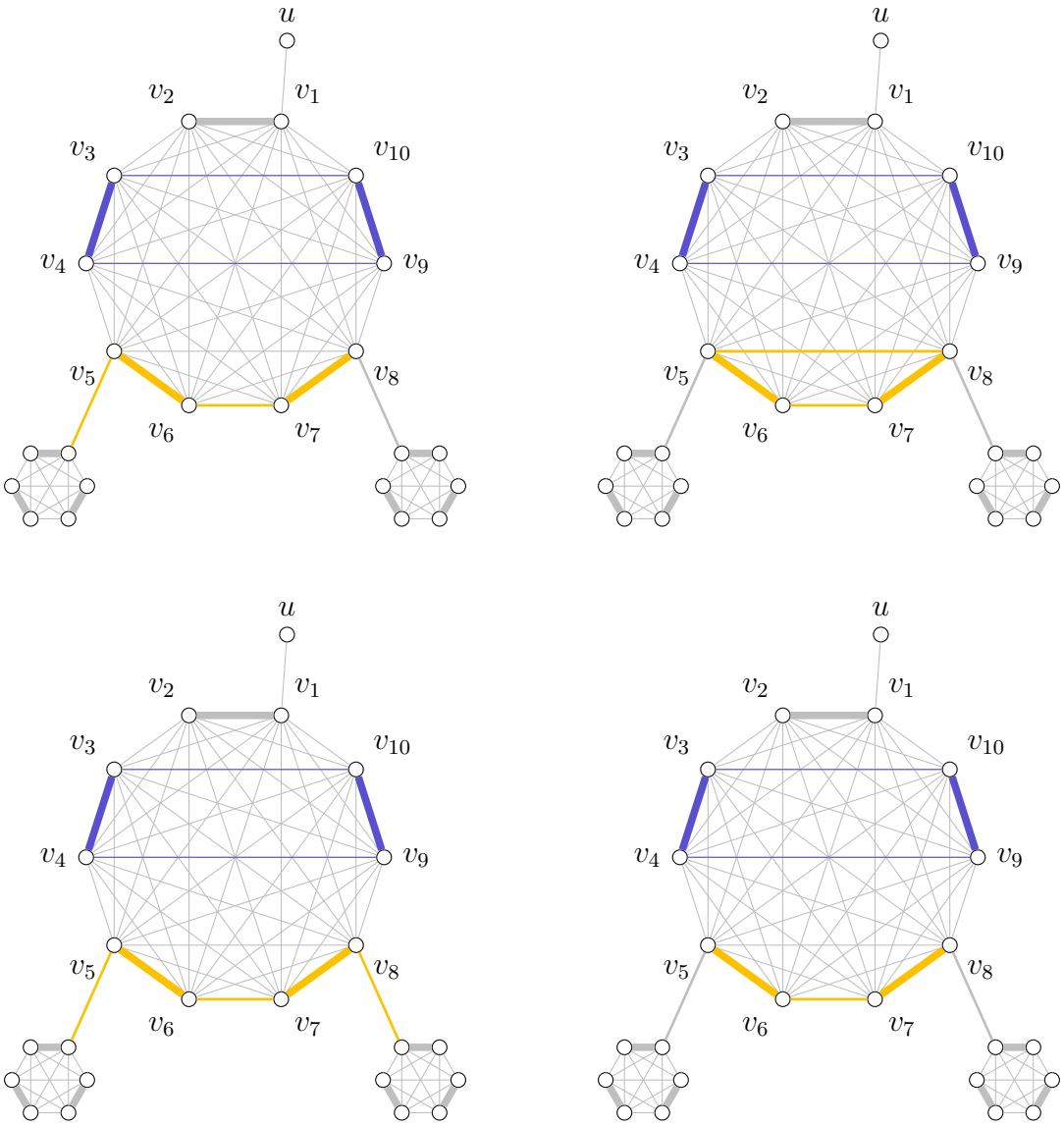


FIGURE 3.9 – *Quatre exemples de solutions locales dans un sous-graphe $G^*(c')$ où c' est une sous-clique comprenant les sommets $\{v_3, \ldots, v_{10}\}$, la porte supérieure de la clique est le sommet v_1. On suppose qu'une solution initiante comprenant les arêtes $\{v_3v_{10}, v_4, v_9, v_6v_7\}$ est donnée. La solution en haut à gauche est extensible, on peut étendre le chemin alterné partiel jaune en ajoutant une arête entre le sommet v_8 et un des sommets de v_1v_2. La solution en haut à droite est absorbante, on peut enlever l'arête v_5v_8 et la remplacer par le chemin $\{v_3, v_2, v_1, v_{10}\}$ car elle n'appartient pas à la solution initiante. La solution en bas à gauche est gelée car toutes les arêtes non couplantes appartiennent à la solution initiante. La solution en bas à droite est fermable, on peut fermer le chemin alterné jaune en ajoutant le chemin $\{v_3, v_2, v_1, v_{10}\}$.*

3.2.2.7 Clique

Soit c une clique de G^* et d la porte supérieure de c. Nous montrons dans ce paragraphe comment calculer les entrées de table pour les ensembles $\mathcal{G}(c)$ et $\mathcal{E}(c)$, comprenant les solutions gelées et extensibles, respectivement, du sous-graphe $G^*(c)$. Notons que comme l'arête reliant d et le parent de c est un isthme, il n'est pas possible d'avoir des solutions absorbantes ou fermables et donc les ensembles $\mathcal{A}(c)$ et $\mathcal{E}(F)$ sont vides. Soit e l'élément alterné partiel de e contenant la porte supérieure d. L'idée est de fu-

sionner les entrées de table de $G^*(e)$ avec celles obtenues pour la sous-clique de c. Si e est un chemin alterné et d est une extrémité de celui-ci, nous utiliserons deux sous-ensembles $\mathcal{E}_d$ et $\mathcal{E}_{d'}$ pour partitionner l'ensemble $\mathcal{E}(e)$. Soit S' est une solution locale de $G^*(e)$, S' appartient à P_d si et seulement si $S' \in \mathcal{E}(e)$ et d est une extrémité d'un chemin alterné de S'. De même, $S' \in \mathcal{E}_{d'}$ si et seulement si $S' \in \mathcal{E}$ et d n'est pas une extrémité d'un chemin alterné de S'. Autrement dit, $\mathcal{E}_d$ (resp. $\mathcal{E}'_d$) contient les solutions locales pour lesquelles on peut étendre le chemin alterné d'extrémité d (resp. d') en ajoutant une arête non couplante incidente à d (resp. d'). Pour calculer ces deux ensembles, nous réutiliserons la valeur de $\mathcal{I}_e$, calculée par la fonction $calculer\_élément\_alterné\_partiel$.

Algorithme 13 : $calculer\_clique$

Entrée : Un graphe d'échafaudage (G^*, M^*), une solution initiante S et une clique c.

$d \leftarrow$ porte supérieure de c; $e \leftarrow$ élément alterné partiel de c contenant d;
$c' \leftarrow$ sous-clique de c;
$calculer\_sous\_clique(c')$; $calculer\_élément\_alterné\_partiel(e)$;
Si *e est un chemin alterné* ***et*** *d est une extrémité de e* **alors**
  $d' \leftarrow$ autre extrémité de e;
  $[\mathcal{E}_d] \leftarrow juxtaposition(\{\mathcal{G}(d)\}, \{\mathcal{E}(d')\}, \{\mathcal{I}_e\})$
  $[\mathcal{E}_{d'}] \leftarrow juxtaposition(\{\mathcal{E}(d)\}, \{\mathcal{G}(d')\}, \{\mathcal{I}_e\})$
  $[\mathcal{G}(c)] \leftarrow juxtaposition(\{\mathcal{G}(e), \mathcal{E}_{d'}\}, \{\mathcal{G}(c'), \mathcal{F}(c'), \mathcal{A}(c'), \mathcal{E}(c')\})$
    $\cup\ fusion_2(\{\mathcal{F}(e), \mathcal{E}_{d'}\}, \{\mathcal{F}(c'), \mathcal{E}(c')\})$
    $\cup\ absorption(\{\mathcal{F}(e)\}, \{\mathcal{A}(c')\})$
    $\cup\ fermeture_2(\{\mathcal{F}(e)\}, \{\mathcal{F}(c')\})$
  $[\mathcal{E}(c)] \leftarrow juxtaposition(\{\mathcal{F}(e), \mathcal{E}_d\}, \{\mathcal{G}(c'), \mathcal{F}(c'), \mathcal{A}(c'), \mathcal{E}(c')\})\}$
    $\cup\ fusion_2(\{\mathcal{F}(e)\}, \{\mathcal{F}(c'), \mathcal{E}(c')\})$
sinon
  $[\mathcal{G}(c)] \leftarrow juxtaposition(\{\mathcal{F}(c'), \mathcal{G}(c'), \mathcal{A}(c'), \mathcal{E}(c')\}, \{\mathcal{F}(e), \mathcal{G}(e), \mathcal{E}(e)\})$
    $\cup\ fusion_2(\{\mathcal{F}(c'), \mathcal{E}(c')\}, \{\mathcal{F}(e), \mathcal{E}(e)\})$
    $\cup\ absorption_1(\{\mathcal{A}(c')\}, \{\mathcal{F}(e)\})$
    $\cup\ fermeture_2(\{\mathcal{F}(c')\}, \{\mathcal{F}(e)\})$
  $[\mathcal{E}(c)] \leftarrow \varnothing$

Lemme 9

Pour toute clique c, les valeurs des entrées de table calculées par l'algorithme 13 sont correctes.

Preuve.

Soient c' la sous-clique de c, e l'élément alterné partiel de c contenant la porte supérieure d de c et S' une solution locale de $G^*(c)$. Distinguons deux cas.

1er cas : Supposons que e est un chemin alterné partiel et que d est une extrémité de e. Supposons que les valeurs calculées par la fonction $calculer\_élément\_alterné\_partiel$ sont correctes pour les ensembles $\mathcal{F}(e)$,

$\mathcal{G}(e)$ et $\mathcal{I}_e$. Calculons les valeurs des entrées de table pour les ensembles $\mathcal{E}_d$ et $\mathcal{E}_{d'}$. Rappelons que $\mathcal{I}_e$ est la juxtaposition des solutions locales gelées des sommets internes de e. Une solution S'' de $G^*(e)$ appartient à $\mathcal{E}_d$ (resp. $\mathcal{E}_d$) si et seulement si S'' est extensible, d est de degré un (resp. degré deux) et d' est de degré deux (resp. degré un). Dans ce cas, la solution locale composant S'' dans $G^*(d)$ est gelée (resp. extensible) et la solution locale composant S' dans $G^*(d')$ (resp. gelée) est extensible. Ainsi l'assignation ligne 6 (resp. ligne 7) est correcte. Montrons que les entrées de table calculées pour les ensembles $\mathcal{G}(c)$ et $\mathcal{E}(c)$ sont correctes.

— S' est gelée si et seulement si S' contient une arête incidente à d. Dans ce cas, une des conditions suivante est vraie : (1) la solution locale composant S' dans $G^*(e)$ est gelée ou appartient à l'ensemble $\mathcal{E}_d$, ou (2) S' est obtenue par une opération de fusion, d'absorption ou de fermeture. Ainsi l'assignation ligne 9 est correcte. S'

— S' est extensible si et seulement si S' ne contient pas une arête incidente à d. Dans ce cas, la solution composant S' dans $G^*(e)$ est gelée et S' est obtenue avec une opération de juxtaposition. Ainsi l'assignation ligne 11 est correcte.

2nd cas : À présent supposons que la porte supérieure d est un sommet interne de e. Alors, S' est gelée et donc S' est composée de n'importe quel type de solution locale dans $G^*(e)$ et $G^*(c')$ et avec n'importe quelle opération. Et donc l'assignation ligne 13 est correcte. De même comme aucune solution locale de $G^*(c)$ n'est extensible, l'assignation ligne 14 est correcte.

□

FIGURE 3.10 – *Deux exemples de solutions locales dans un sous-graphe $G^*(c)$ où c est une clique. La clique est reliée à son père par le sommet u. Les deux solutions locales contiennent un cycle alterné en jaune et un chemin alterné en bleu. À gauche la solution locale est gelée et à droite la solution locale est extensible.*

Des exemples de solutions locales dans une sous-clique sont données par figure 3.10 et les valeurs des entrées de table pour ces exemples sont exposées dans le tableau 3.4.

TABLE 3.4 – *Valeurs des entrées de table pour le graphe de la figure 3.10.*

Nombre de cycles	Nombres de chemins $\mathcal{G}(G^*(c))$	$\mathcal{E}(G^*(c))$
0	$\{1,\ldots,3\}$	$\{1,\ldots,4\}$
1	$\{0,1,2\}$	$\{1,2\}$
2	$\{0\}$	$\varnothing$
3	$\varnothing$	$\varnothing$

Algorithme 14 : Fonction de FAISABILITÉ

Entrée : Un graphe d'échafaudage (G^*, M^*) tel que G^* est un graphe de clusters connectés, une solution initiante S et deux entiers naturels σ_c et σ_p

1 $r \leftarrow$ racine de G^* ;
2 $calculer\_sous\_clique(r)$;
3 **retourner** $\sigma_p \in ([\mathcal{F}(r), \sigma_c] \cup [\mathcal{G}(r), \sigma_c] \cup [\mathcal{A}(r), \sigma_c] \cup [\mathcal{E}(r), \sigma_c])$

3.2.2.8 Fonction de faisabilité

Finalement, après avoir calculé toutes les entrées de table, on peut vérifier s'il est possible de construire une solution. Pour cela, il suffit de regarder dans les entrées de table calculées pour la clique racine du graphe d'échafaudage, ce qui est décrit par le corollaire suivant.

Corollaire 3

Étant donné une solution initiante S, l'algorithme 14 retourne VRAI si et seulement si il est possible de construire une solution composée de σ_c cycles alternés et σ_p chemins alternés et contenant toutes les arêtes de S. La complexité en temps de cet algorithme est $\mathcal{O}(|V(G^*)| \times \sigma_c^2)$.

Preuve.

Soit r la clique racine de G^*. Comme $G^*(r) = G^*$, il existe une solution S' contenant S et telle que $\sigma_c(S) = \sigma_c$ et $\sigma_p(S) = \sigma_p$ si et seulement si S appartient à $\mathcal{F}(r) \cup \mathcal{G}(r) \cup \mathcal{A}(r) \cup \mathcal{P}(r)$. Le retour de la fonction indique si une telle solution existe et donc l'algorithme est correct. Concernant la complexité en temps, chaque opération de composition dans la table (*i.e.* algorithme 7, algorithme 9 et algorithme 8) est en $\mathcal{O}(\sigma_c^2)$: les entrées de table peuvent être vues comme des tableaux de $\sigma_c + 1$ cases où chaque case est un intervalle (indiquant le nombre de chemins alternés). Dans chacun des algorithmes algorithme 10, algorithme 11, algorithme 12, une itération de boucle possède une complexité en temps de $\mathcal{O}(\sigma_c^2)$ et algorithme 13 a une complexité en temps total de $\mathcal{O}(\sigma_c^2)$. Dans l'algorithme 10, le nombre total d'itérations de boucle dépend du nombre de cliques dans G^*. Dans l'algorithme 11, le nombre total d'itérations de boucle dépend du nombre de sommets. Dans algorithme 12, le nombre total d'itérations de boucle dépend du nombre d'éléments alternés partiels. Ainsi, pour ces algorithmes précédents, le nombre total d'itérations de boucle dépend d'un nombre borné par

le nombre de sommets du graphe. On a donc bien une complexité en temps de $\mathcal{O}(|V(G^*)| \times \sigma_c^2)$ pour l'algorithme 14. □

Notons qu'il est possible de construire une fonction de faisabilité pour un graphe de clusters connectés non connexe : il suffit pour cela de calculer les entrées de table pour chacune des composantes connexes puis de faire une opération de juxtaposition sur ces entrées.

On a construit une fonction de faisabilité s'exécutant en temps polynomial pour les graphes de clusters connectés. En utilisant cette fonction de faisabilité dans l'algorithme glouton, on peut calculer une solution approchée dans cette classe de graphe d'échafaudage. On peut également montrer que le facteur entre le poids de la solution alors calculée et le poids d'une solution optimale est borné par une constante.

Théorème 10

L'algorithme 6 donne une solution pour ÉCHAFAUDAGE MAXIMUM dans les graphes de clusters connectés avec un facteur d'approximation de cinq.

Preuve.

Tout d'abord, par la propriété 6, l'algorithme 6 renvoie une solution. Soient (G^*, M^*, ω) un graphe d'échafaudage tel que G^* est un graphe de clusters connectés, S_{opt} une solution optimale de ÉCHAFAUDAGE MAXIMUM et S_{app} la solution renvoyée par l'algorithme. Il reste à montrer que $\omega(S_{opt}) \leq 5 \cdot \omega(S)$. Tout d'abord, notons que pour chaque chemin alterné, il y a une arête non couplante de moins par rapport au nombre d'arêtes appartenant au couplage. Dans un cycle alterné, il y a autant d'arêtes non couplantes que d'arêtes couplantes. Et comme il y a exactement une arête de couplage incidente à chaque sommet, le nombre d'arêtes n'appartenant pas au couplage dans une solution est égal à $|V(G^*)| - \sigma_p = n$. Dans ce qui suit, nous ne considérons pas les arêtes de couplage dans S_{opt} et S_{app}. Nous dénotons par $e_1^{opt}, \ldots, e_n$ les arêtes appartenant à S_{opt} et par $e_1^{app}, \ldots, e_n^{app}$ les arêtes appartenant à S_{app}. La suite de la preuve est dévolue à la construction d'une application $\varphi : S_{opt} \mapsto S_{app}$ telle que les deux inéquations suivantes soient vérifiées.

$$\forall e \in S_{opt}, \omega(e) \leq \omega(\varphi(e)) \tag{3.1}$$
$$\forall e \in S_{app}, |\varphi^{-1}(\{e\})| \leq 5 \tag{3.2}$$

L'inéquation (3.1) indique que pour toute arête de la solution optimale, il existe une arête de la solution approchée de poids au moins aussi grand dans la solution approchée. L'inéquation (3.2) indique que chaque arête de la solution approchée est associée à au plus cinq arêtes de la solution optimale.
Une arête e_i^{opt} de la solution optimale peut ne pas être choisie par l'algorithme pour trois raisons.

1. Elle est éliminée parce qu'elle appartient à l'ensemble I (ligne 8 de l'algorithme 6) quand une arête e_j^{app} est choisie. Dans ce cas, on a $\omega(e_j^{app}) \geq$

$\omega(e_i^{opt})$ car seules les arêtes apparaissant après e_j^{app} dans la liste ordonnée E peuvent appartenir à I. Quand une arête e_j^{app} est choisie, elle peut éliminer au plus deux arêtes de la solution optimale, comme le montre la figure 3.11. On assigne $\varphi(e_i^{opt}) = e_j^{app}$. L'inéquation (3.1) est vérifiée par construction et l'inéquation (3.2) reste vraie, en considérant uniquement les arêtes éliminées de cette façon.

2. Elle est éliminée car son addition déconnecterait le graphe et le nombre de chemins alternés et de cycles alternés nécessaires pour couvrir l'ensemble du graphe deviendrait trop grand. Deux cas peuvent être différenciés.
   (a) L'ajout de l'arête fermerait un chemin alterné en un cycle alterné. Dans ce cas, ce chemin alterné contient au moins une arête n'appartenant pas au couplage. Soit e_j^{app} la dernière de ce chemin alterné à avoir été ajoutée. Comme e_j^{app} a été choisie avant e_i^{opt}, on a $\omega(e_j^{app}) \geq \omega(e_i^{opt})$. On assigne $\varphi(e_i^{opt}) = e_j^{app}$. L'inéquation (3.1) est vérifiée par construction. Comme e_j^{app} est la dernière arête du chemin alterné à avoir été ajoutée, aucune autre arête de la solution optimale n'a été affectée à e_i^{opt} de cette façon et donc l'inéquation (3.2) reste également satisfaite.
   (b) Elle est éliminée car son addition supprimerait un isthme x et augmenterait de un le nombre de chemins alternés nécessaires. Dans ce cas, lors d'une étape précédente, un isthme x' a été éliminé par l'ajout d'une arête e_j^{app} et cela force l'isthme x à être dans la solution approchée. Comme e_j^{app} a été choisie avant e_i^{opt}, on a $\omega(e_j^{app}) \geq \omega(e_i^{opt})$. On assigne $\varphi(e_i^{opt}) = e_j^{app}$. L'inéquation (3.1) est vérifiée par construction. Comme x est incidente à au plus deux arêtes de la solution optimale, au plus deux arêtes peuvent avoir été éliminées de cette façon et donc l'inéquation (3.2) reste également satisfaite.
3. Elle est éliminée car son addition fusionnerait deux chemins alternés p_1 et p_2. Si e_i^{opt} n'est pas incidente à une porte et p_1 et p_2 sont constitués d'une seule arête de couplage, alors cela signifie que le nombre de chemins alternés et de cycles alternés est atteint et dans ce cas la solution partielle est égale à S_{app}. Comme le nombre d'arêtes est le même dans la solution optimale et la solution approchée, il existe une arête e_j^{app} dans la solution approchée telle que $|\varphi(e_j^{app})^{-1}| = 0$. On assigne $\varphi(e_i^{opt}) = e_j^{app}$. Comme e_j^{app} a été choisie avant e_i^{opt}, on a $\omega(e_j^{app}) \geq \omega(e_i^{opt})$. Les inéquations (3.1) et (3.2) sont vérifiées par construction. Sinon, l'algorithme élimine e_i^{opt} car au moins un des chemins alternés fusionnés doit être fermé en un cycle alterné pour atteindre le nombre correct de cycles alternés dans la solution. Dans ce cas, trois cas peuvent survenir (1) soit $\sigma_p = 0$ et le cas 2(b) ne peut pas intervenir ; (2) soit il existe un cycle alterné dans la solution approchée et aucune arête du cas 2(a) n'est assignée à l'arête qui ferme le cycle alterné ; (3) soit il existe un chemin alterné dans la solution approchée qui contient une arête n'ayant pas d'arête du cas 2(a) assigné sinon e_i^{opt} ne pourrait pas être éliminée pour cette raison. Ainsi, il existe une arête e_j^{app} choisie lors d'une étape précédente telle que $|\varphi(e_j^{app})^{-1}| \leq 4$. On assigne $\varphi(e_i^{opt}) = e_j^{app}$. Les inéquations (3.1) et (3.2) sont vérifiées par construction.

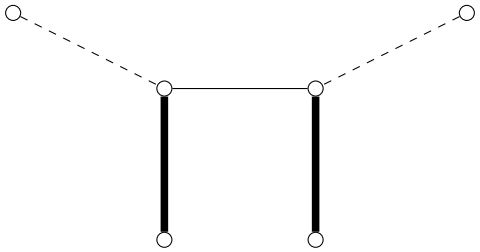


FIGURE 3.11 – *Une arête choisie par l'algorithme glouton peut éliminer au plus deux arêtes d'une solution optimale. Les arêtes en gras appartiennent au couplage, l'arête en traits pleins est l'arête choisie par l'algorithme glouton et les deux arêtes dessinées en pointillés appartiennent à une solution optimale.*

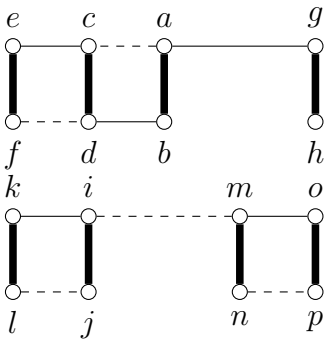


FIGURE 3.12 – *Le facteur d'approximation de cinq donné par le théorème 10 est serré. Les arêtes de couplage sont dessinées en gras, les arêtes en pointillés appartiennent à la solution approchée et les arêtes en traits pleins appartiennent à une solution optimale. G^* est composé des cliques $C_1 = \{a, b, c, d, e, f\}, C_2 = \{g, h\}, C_3 = \{i, j, k, l\}$ et $C_4 = \{m, n, o, p\}$. La clique C_2 est reliée à la clique C_3 par l'arête gk. Toutes les arêtes ont un poids nul, exceptées ac et les arêtes appartenant à la solution optimale. Dans cet exemple, on cherche une solution pour $\sigma_c = 0$ et $\sigma_p = 3$. On suppose que l'algorithme glouton choisi en premier l'arête ac. Par conséquent, la solution approchée renvoyée par l'algorithme glouton a un poids de un, tandis qu'une solution optimale a un poids de cinq.*

On a construit une application φ respectant les inéquations (3.1) et (3.2). Ainsi,

$$\omega(S_{opt}) \leq \omega(\varphi(S_{opt})) \leq 5 \times \omega(S_{app}). \tag{3.3}$$

Le facteur montré ici est serré, comme le montre l'exemple donné par la figure 3.12.

□

En associant ce résultat à celui du théorème 6, on obtient directement le résultat suivant.

Corollaire 4

ÉCHAFAUDAGE MAXIMUM est APX-complet pour les graphes de clusters connectés.

FIGURE 3.13 – *Une collection contenant un chemin alterné et un cycle alterné. Les arêtes de couplage sont dessinées en gras. Les nombres sous les sommets correspondent à leurs positions respectives dans l'ordre. Les sommets appartenant aux quatre premières positions appartiennent au chemin alterné et les sommets appartenant aux quatre dernières positions appartiennent au cycle alterné.*

3.2.3 Fonction de faisabilité dans le cas général

Nous avons déjà dit qu'il n'était pas possible de trouver une fonction de faisabilité s'exécutant en temps polynomial dans le cas général car le problème ÉCHAFAUDAGE est NP-complet. Cependant, il peut être intéressant de créer une fonction non polynomiale : si on exécute l'algorithme glouton dans le cas général sur des instances de petites ou moyennes tailles, cela peut indiquer à quel point l'algorithme glouton peut s'approcher de la solution optimale.

Nous montrons dans cette partie comment créer une telle fonction de faisabilité en faisant une réduction vers le problème SATISFAISABILITÉ: on va construire une formule booléenne permettant de modéliser le problème ÉCHAFAUDAGE. On pourra ensuite répondre à la question de faisabilité en utilisant un solveur permettant de trouver une assignation satisfaisant la formule ainsi créée.

Soit (G^*, M^*, ω) un graphe d'échafaudage dans lequel on souhaite savoir si le problème d'échafaudage est faisable. Soit S une solution initiante dans (G^*, M^*, ω). L'idée générale de la formulation en variable booléenne est d'ordonner les sommets de $V(G^*)$ de façon à ce que tous les sommets appartenant à un même élément alterné soient consécutifs dans l'ordre. Soient $v_1, \ldots, v_k$ les sommets d'un élément alterné e et soient $p_1, \ldots, p_k$ leurs positions respectives dans l'ordre telles que $\forall p_i, p_j, |p_i - p_j| \leq k$. Soient v_i et v_j deux sommets voisins dans e. Si e est un chemin alterné, alors $|p_i - p_j| = 1$. Si e est un cycle alterné, alors soit $|p_i - p_j| = 1$ ou $|p_i - p_j| = k$. Dit autrement, si deux sommets sont voisins dans un élément alterné, alors leurs positions sont consécutives, à l'exception des sommets possédant la première et dernière position d'un cycle alterné. Voir la figure 3.13 pour un exemple.

Dans ce qui suit, étant donné un graphe d'échafaudage (G^*, M^*, ω), nous allons décrire une formule $\mathcal{F}$ booléenne permettant de transformer le problème ÉCHAFAUDAGE en problème de SATISFAISABILITÉ. Pour simplifier la compréhension, nous décrirons la formule $\mathcal{F}$ en logique propositionnelle. Il est toujours possible de transformer une formulation en logique propositionnelle pour qu'elle soit sous forme normale conjonctive, comme indiqué dans [72]. Nous allons décrire un ensemble de règles qui, associées par le connecteur logique « $\wedge$ » (et) permettent de trouver une collection de σ_p chemins alternés et σ_c cycles alternés dans le graphe d'échafaudage.

Nous allons faire deux formulations différentes : une première permettant de trouver une collection ne contenant que des chemins alternés et une seconde permettant de trouver une collection pouvant contenir à la fois des chemins alternés et des cycles

alternés.

3.2.3.1 Version sans cycles alternés

Pour la première formulation, nous cherchons une collection contenant exactement σ_p chemins alternés. Nous allons obliger tous les sommets consécutifs dans l'ordre à être voisins, à l'exception d'exactement $\sigma_p - 1$ qui correspondent aux derniers sommets de chaque élément dans l'ordre.

Soit $\mathcal{V} = \{x_1, \ldots, x_k\}$ un ensemble de variables booléennes. Dans ce qui suit, nous aurons parfois besoin d'imposer qu'une seule variable de $\mathcal{V}$ soit assignée à VRAI. Pour cela, nous introduisons la règle d'unicité sur l'ensemble $\mathcal{V}$, notée $unique(\mathcal{V})$. Cette règle peut être formulée de la façon suivante :

$$unique(\mathcal{V}) : \left(\bigvee_{x \in \mathcal{V}} x \right) \wedge \left(\bigwedge_{x_i, x_j \in \mathcal{V}} x_i \Rightarrow \neg x_j \right).$$

Pour chaque sommet, nous associons une position dans l'ordre. On dénote par $P = \{1, \ldots, |V(G^*)|\}$ l'ensemble des positions possibles. Pour chaque sommet v et pour toute position i, nous introduisons une variable booléenne p_i^v signifiant que le sommet v appartient à la position i. Comme deux sommets ne peuvent pas posséder la même position dans l'ordre, nous appliquons une règle d'unicité pour chaque position, c'est-à-dire :

$$\forall i \in P, unique(\{p_i^v | v \in V(G^*)\}). \tag{3.4}$$

On souhaite que chaque sommet soit associé à une position, pour cela on introduit la règle suivante :

$$\forall v \in V(G^*), \bigvee_{i \in P} p_i^v. \tag{3.5}$$

Comme le nombre de positions et de sommets sont égaux, l'association des deux règles précédentes nous assure qu'un sommet v ne pourra pas être assigné à deux positions. Soit $E = \{e_1, \ldots, e_{\sigma_p}\}$ l'ensemble des chemins alternés recherchés. Pour chaque élément e_k de cet ensemble et pour chaque position i, on introduit une variable booléenne $fin_{e_k}^i$ qui indique que le sommet placé à la position i est le dernier sommet de ce chemin alterné dans l'ordre. Comme chaque chemin alterné possède un unique dernier élément, on créé la règle d'unicité suivante :

$$\forall e_k \in E, unique(\{fin_{e_k}^i \mid i \in P\}). \tag{3.6}$$

On empêche deux chemins alternés d'avoir la même position de la façon suivante :

$$\forall i \in P, \forall e_k, e_{k'} \in E, fin_{e_k}^i \Rightarrow \neg fin_{e_k'}^i. \tag{3.7}$$

Par simplicité, on introduit également une variable booléenne FIN^i pour chaque position i indiquant que le sommet placé à la position i est un dernier sommet d'un chemin alterné. Cela se traduit comme suit :

$$\forall i \in P, \forall e_k \in E, fin_{e_k}^i \Rightarrow FIN^i. \tag{3.8}$$

On souhaite que le dernier sommet dans l'ordre soit un dernier sommet d'un chemin alterné, arbitrairement on choisira le chemin alterné e_{σ_p}. On assigne la valeur VRAI à la variable $fin^{|V(G^*)|}_{e_{\sigma_p}}$, cela peut se faire en introduisant une clause ne contenant que ce littéral.

Les deux extrémités d'une arête de couplage ont des positions consécutives dans l'ordre :

$$\forall v \in V(G^*), \forall i \in P, p^v_i \Rightarrow (p^{M^*(v)}_{i+1} \vee p^{M^*(v)}_{i-1}). \tag{3.9}$$

Si un sommet v est le dernier sommet dans l'ordre d'un chemin alterné, alors son sommet couplé $M^*(v)$ appartient à la position précédente de v dans l'ordre :

$$\forall v \in V(G^*), \forall i \in P, (p^v_i \wedge FIN^i) \Rightarrow p^{M^*(v)}_{i-1}. \tag{3.10}$$

Enfin, pour chaque sommet v, si v n'est pas un dernier sommet d'un chemin alterné, alors exactement un de ses voisins parmi $N^*(v)$ est consécutif dans l'ordre :

$$\forall v \in V(G^*), \forall i \in P, p^v_i \Rightarrow FIN^i \vee \left(\bigvee_{u \in N^*(v)} p^u_{i+1} \vee p^u_{i-1} \right). \tag{3.11}$$

Enfin, pour chaque arête non couplante uv, nous créons une variable booléenne λ_{uv} indiquant que uv appartient à la solution. Si l'arête uv appartient à la solution, alors u et v ont des positions consécutives dans l'ordre :

$$\forall uv \in E(G^*) \setminus M^*, \forall i \in P, \forall e_k k \in E, \lambda_{uv} \Rightarrow (p^v_i \Rightarrow p^u_{i+1} \vee p^u_{i-1}). \tag{3.12}$$

3.2.3.2 Version avec des cycles alternés

Pour la formulation permettant de trouver également des cycles alternés, on va réutiliser les règles précédentes, en prenant cette fois-ci l'ensemble des éléments alternés dans $E = \{e_1, \ldots, e_{\sigma_c+\sigma_p}\}$. Dans cette nouvelle formulation, il faut pour chaque cycle alterné obliger le dernier sommet de ce cycle dans l'ordre à être adjacent avec le premier sommet de ce cycle dans l'ordre. On a besoin pour cela d'introduire, pour chaque élément e_k et chaque position i une nouvelle variable booléenne $\mathit{début}^i_{e_k}$ indiquant que le sommet placé à la position i est le premier sommet de l'élément alterné e_k. Comme cette variable est très similaire à la variable $fin^i_{e_k}$, on reprend les règles (3.6) et (3.7) pour cette variable, c'est-à-dire :

$$\forall e_k \in E, unique(\{\mathit{début}^i_{e_k} \mid i \in P\}) \tag{3.13}$$

$$\forall i \in P, \forall e_k, e_{k'} \in E, \mathit{début}^i_{e_k} \Rightarrow \neg \mathit{début}^i_{e_{k'}}. \tag{3.14}$$

De même, on introduit une variable $D\acute{E}BUT^i$ indiquant que le sommet placé à la position i est un premier sommet d'un élément alterné.

$$\forall i \in P, \forall e_k \in E, \mathit{début}^i_{e_k} \Rightarrow D\acute{E}BUT^i. \tag{3.15}$$

Comme un sommet ne peut pas être à la fois un premier sommet et un dernier sommet d'un élément alterné, on a :

$$\forall i \in P, \forall e_k \in E, D\acute{E}BUT^i_{e_k} \Rightarrow \neg FIN^i. \tag{3.16}$$

Si un sommet v est le premier sommet dans l'ordre d'un élément alterné, alors son sommet couplé $M^*(v)$ appartient à la position suivante de v dans l'ordre :

$$\forall v \in V(G^*), \forall i \in P, (p_i^v \wedge FIN^i) \Rightarrow p_{i+1}^{M^*(v)}. \tag{3.17}$$

Pour les cycles alternés, nous devons également encoder le fait que le premier et le dernier sommet sont voisins dans la solution. Soit E^c l'ensemble des cycles alternés recherchés, on a :

$$\forall v \in V(G^*), \forall i, j \in P, e_k \in E^c, (p_i^v \wedge début_{e_k}^i) \Rightarrow \left(\bigvee_{u \in N^*(v)} p_i^u \wedge fin_{e_k}^j \right). \tag{3.18}$$

Si une arête non couplante uv appartient à la solution, alors les positions de u et v sont soit consécutives dans l'ordre, soient u et v sont les premiers et derniers éléments d'un même élément alterné. On remplace donc la règle Équation 3.12 par la règle suivant :

$$\begin{aligned} \forall uv \in E(G^*) \setminus M^*, \forall i \in P, \lambda_{uv} \quad &\Rightarrow \quad (p_i^v \Rightarrow p_{i+1}^u \vee p_{i-1}^u) \\ &\bigvee_{e_k \in E^c, j \in P} (début_{e_k}^i \wedge fin_{e_k}^j \wedge (p_i^v \vee p_j^v) \wedge (p_i^u \vee p_j^u)). \end{aligned} \tag{3.19}$$

Enfin, pour chaque élément alterné e_k, il faut imposer que pour toute position i située entre le premier et le dernier sommet de e_k, le sommet situé à la position i ne soit ni un premier sommet, ni un dernier sommet d'un autre élément alterné. Pour cela, pour chaque élément alterné e_k, on va imposer que le sommet précédent le premier sommet de e_k dans l'ordre soit le dernier sommet de l'élément alterné e_{k-1}. De même, on va imposer que le sommet suivant le dernier sommet de e_k dans l'ordre soit le premier sommet de l'élément alterné e_{k+1}. Cela se modélise de la façon suivant :

$$\forall e_k in E, \forall i \in P, début_{e_k}^i \Rightarrow fin_{e_{k-1}}^{i-1} \wedge fin_{e_k}^i \Rightarrow début_{e_{k+1}}^{i+1}. \tag{3.20}$$

3.2.3.3 Utilisation dans l'algorithme glouton

On peut utiliser la formulation en variables booléenne décrite précédemment afin de créer une fonction de faisabilité pour l'algorithme 6. Pour cela, lorsqu'une arête n'appartenant pas au couplage uv est choisie ou testée par l'algorithme, nous assignons la variable booléenne λ_{uv} à VRAI. Ainsi, pour chaque arête uv non couplante de la solution initiante S, on ajoutera une clause contenant le seul littéral λ_{uv}. Malheureusement, il s'avère que dans ce cas l'algorithme 6 ne possède pas un facteur d'approximation borné par une constante, comme le montre la figure 3.14.

3.3 Expériences

3.3.1 Instances sélectionnées

Nous indiquons à présent dans cette section les performances de différentes versions de l'algorithme glouton sur des instances réelles. Des statistiques sur ces ins-

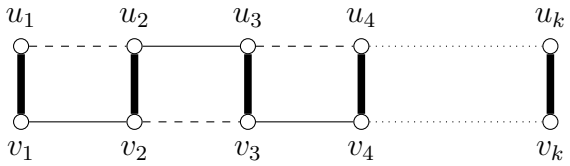


FIGURE 3.14 – *L'algorithme 6 renvoie une solution approchée dont le facteur d'approximation n'est pas borné par une constante. Soit (G^*, M^*, ω) un graphe d'échafaudage composé d'une grille de taille $2 \times k$ où les arêtes de couplage, dessinées en gras, correspondent aux arêtes entre les deux lignes. Soient $(u_1, \ldots, u_k)$ et $(v_1, \ldots, v_k)$ les sommets de la première et dernière ligne, respectivement. On recherche dans ce graphe d'échafaudage une collection comprenant un chemin alterné et aucun cycle alterné. Si l'algorithme glouton choisit en premier l'arête u_1u_2, alors la seule solution faisable est $S_{app} = \{u_\ell u_\ell \mid \ell \mod 2 = 1\} \cup \{v_\ell v_{\ell+1} \mid \ell \mod 2 = 0\}$. S' est composée des arêtes dessinées en pointillés dans la figure. Supposons qu'une solution optimale est composée par $S_{opt} = E(G^*) \setminus (M^* \cup S_{app})$. S_{opt} est composée des arêtes dessinées en traits pleins dans la figure. Si toutes les arêtes composant S_{opt} et l'arête u_1u_2 ont un poids de un et que toutes les arêtes de $S_{app} \setminus \{u_1u_2\}$ ont un poids nul, alors nous avons, $(k-1) \cdot \omega(S_{app}) = \omega(S_{opt})$ qui est un ratio non borné par une constante.*

tances sont présentées dans le tableau 3.5. On peut remarquer que les instances sont toutes relativement peu denses, y compris pour les instances de grande taille : leur degré moyen et médian est semblable. Pour toutes ces instances, on ne recherche que des chemins alternés pour résoudre ÉCHAFAUDAGE MAXIMUM. Cela est dû à la façon dont ont été générées les instances. En effet, certains organismes étudiés peuvent contenir des chromosomes circulaires mais comme l'information de séquençage disponible sur celui-ci est uniquement parcellaire, des parties d'un chromosome circulaire ne sont pas couvertes, ce qui transforme la séquence circulaire en plusieurs séquences linéaires. C'est également le cas pour les chromosomes linéaires. Pour déterminer le nombre adéquat de chemins alternés à chercher, la formulation linéaire en nombres entiers est utilisée. Celle-ci, calcule une collection de chemins alternés et cycles alternés, sans contrainte sur leurs nombres, qui maximise le poids de la solution.

3.3.2 Complétion des instances

Nous avons implémenté et comparé trois versions de l'algorithme glouton : la version sur les graphes complets, présentée dans [20, 27], la version sur les graphes de clusters connectés, présentée dans la sous-section 3.2.2, et la version pour les graphes de blocs. Un *graphe de blocs* est un graphe constitué de cliques séparées par des points d'articulations, il peut être vu comme un graphe de clusters connectés pour lequel on aurait contracté tous les isthmes en un unique sommet. Formellement, un graphe de blocs est un graphe G pouvant être décomposé en un ensemble de cliques non forcément disjointes $C = \{c_1, \ldots, c_n\}$ tel que pour n'importe quel sous-ensemble $C' \subseteq C$, l'intersection de toutes les cliques de C' est soit égal au graphe vide, soit égal à un unique sommet (qui est un point d'articulation dans le graphe). Comme un graphe de blocs a une structure proche de celle d'un graphe de clusters connectés, on peut facilement adapter l'algorithme de faisabilité pour les graphes de clusters connectés aux graphes de blocs.

TABLE 3.5 – *Statistiques sur les instances pour* ÉCHAFAUDAGE. *Pour ces instances, le nombre de cycles alternés recherchés est égal à zéro et le nombre de chemins alternés recherchés est donné dans la ligne σ_p.*

	anopheles	anthrax	ebola	gloebacter
Nombre de sommets	84 090	8 110	34	9 034
Nombre d'arêtes	113 497	11 013	43	12 402
Degré maximum	50	7	5	12
Degré moyen	2.7	2.7	2.5	2.7
Degré médian	3	3	2	3
Composantes connexes	444	3	1	7
$\boldsymbol{\sigma_p}$	8 050	371	4	506

	lactobacillus	monarch	pandora	pseudomonas
Nombre de sommets	3 796	28	4 902	10 496
Nombre d'arêtes	52 333	33	6 722	14 334
Degré maximum	12	4	7	9
Degré moyen	2.7	2.3	2.7	2.7
Degré médian	3	2	3	3
Composantes connexes	2	1	2	13
$\boldsymbol{\sigma_p}$	185	4	291	543

	rice	sacchr12	sacchr3
Nombre de sommets	168	1 778	592
Nombre d'arêtes	223	2 411	823
Degré maximum	6	10	7
Degré moyen	2.7	2.7	2.8
Degré médian	3	3	3
Composantes connexes	1	4	1
$\boldsymbol{\sigma_p}$	10	34	101

Lorsque nous utilisons l'algorithme glouton sur les instances réelles, comme expliqué dans la sous-section 3.2.1, nous devons le compléter avec des arêtes de poids nul. L'avantage de la version utilisant les graphes de blocs par rapport à celle utilisant les graphes de clusters connectés est qu'elle nécessite de compléter l'instance originale avec moins d'arêtes, comme l'explique la figure 3.15. Des statistiques sur la complétion des instances utilisées sont présentées dans la tableau 3.6. Cela va nous aider à vérifier l'hypothèse suivante : est-ce que le fait d'ajouter moins d'arêtes à une instance réelle permet d'obtenir une solution de ÉCHAFAUDAGE MAXIMUM avec l'algorithme glouton ayant un poids plus important ? Nous avons déjà vu que d'un point de vue théorique, cela n'était pas vrai car l'algorithme glouton sur les graphes complets est un algorithme 3-approché (théorème 9) tandis que l'algorithme glouton sur les graphes de clusters connectés est un algorithme 5-approché (théorème 10). Cependant, on peut espérer que sur des instances réelles, ce ne soit pas forcément le cas. Ainsi, si l'hypothèse est vraie, les solutions renvoyées par la version sur les graphes de clusters connectés devraient être meilleures que celles renvoyées par la version sur les graphes complets. De même, les solutions renvoyées par la version sur les graphes de blocs devraient être meilleures que celles renvoyées par la version sur les graphes de clusters connectés.

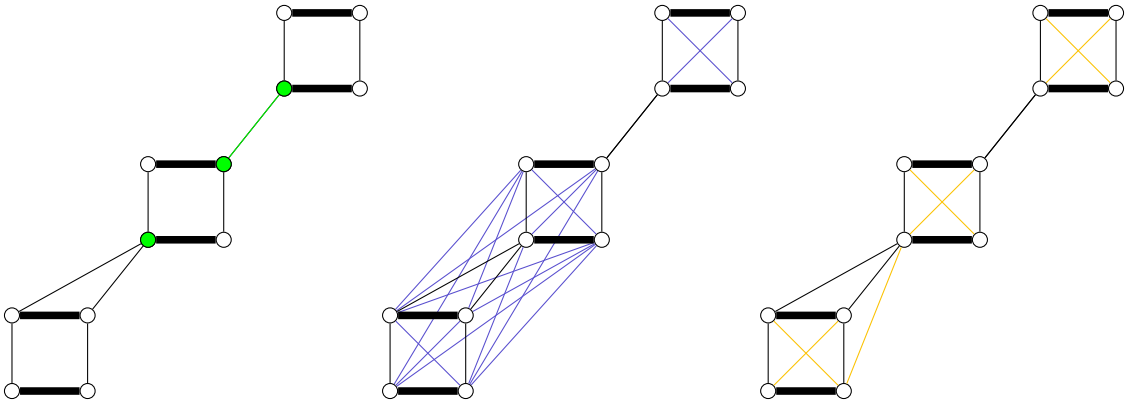


FIGURE 3.15 – *Un graphe d'échafaudage complété en graphe de clusters connectés et en graphe de blocs. Le graphe de gauche est le graphe original, il possède un isthme et trois points d'articulation, dessinés en verts. Lorsqu'on complète ce graphe en graphe de clusters connectés, on obtient le graphe dessiné au centre, avec les arêtes ajoutées dessinées en bleu. Ce graphe contient deux cliques et nécessite d'ajouter vingt arêtes au graphe de départ. En revanche, lorsqu'on complète ce graphe en graphe de blocs, on obtient le graphe de droite qui est constitué de quatre cliques (l'isthme compte comme une clique). Ce graphe ne nécessite d'ajouter que sept arêtes au graphe de départ.*

3.3.3 Optimisation

Pour les graphes de clusters connectés et les graphes de blocs, on peut noter deux optimisations importantes qui permettent grandement d'accélérer le calcul. La première consiste à conserver les entrées de table calculées par la fonction de faisabilité d'un appel à l'autre : en effet, une fois que toutes les entrées ont été calculées, il n'est nécessaire que de modifier les entrées présentes dans la branche contenant l'arête non couplante à évaluer. Conserver les entrées de table d'un appel à l'autre permet donc de limiter le nombre d'entrées à modifier. La deuxième optimisation consiste à ne pas parcourir tous les éléments alternés d'une clique mais seulement ceux qui sont reliés à un enfant de la clique. Pour ceux qui ne sont pas reliés à un enfant, on peut remplir en temps constant les entrées de table correspondant à l'union de tous ces éléments alternés juste en maintenant le nombre de cycles alternés, le nombre de chemins alternés de longueur au moins deux et le nombre de chemins alternés ne contenant qu'une seule arête. Ainsi, cela permet de rendre plus rapide le calcul des entrées de table pour les cliques et d'autant plus que la plupart des éléments alternés des cliques ne sont pas reliés à des enfants.

3.3.4 Résultats

Les résultats des différentes versions sont détaillés dans le tableau 3.7. L'algorithme glouton, quelle que soit sa version, donne des solutions ayant un poids proche du poids optimal : le facteur d'approximation est compris entre 1 et 1.0168. Il calcule même une solution optimale pour l'instance `ebola`.

Concernant notre hypothèse, un premier enseignement que l'on peut retirer de ces résultats est que bien que le facteur théorique soit plus grand pour les graphes

TABLE 3.6 – *Statistiques sur les complétions des différentes versions de l'algorithme glouton. Le taux de complétion correspond au pourcentage d'arêtes ajoutées par rapport à la version complète. Les grandes cliques sont les cliques possédant plus que deux sommets.*

		anopheles	anthrax	ebola	gloebacter
Complet	Arêtes ajoutées	$3 \cdot 10^9$	$3.3 \cdot 10^7$	561	$4 \cdot 10^7$
	Taux complétion	100%	100%	100%	100%
Cluster	Arêtes ajoutées	$8 \cdot 10^8$	$3.1 \cdot 10^7$	455	$3.8 \cdot 10^7$
	Taux complétion	27%	94%	81%	95%
	Cliques	3 599	123	2	177
	Grandes cliques	58	4	1	2
	Taille clique max.	77 156	7 870	32	8 706
Bloc	Arêtes ajoutées	$6 \cdot 10^8$	$3 \cdot 10^7$	396	$3.6 \cdot 10^7$
	Taux complétion	20%	91%	70%	95%
	Cliques	9 028	355	5	494
	Grandes cliques	642	2	1	16
	Taille clique max.	74 095	7 740	30	8 510

		lactobacillus	monarch	pandora	pseudomonas
Complet	Arêtes ajoutées	$7.2 \cdot 10^6$	378	$1.2 \cdot 10^7$	$5.5 \cdot 10^7$
	Taux complétion	100%	100%	100%	100%
Cluster	Arêtes ajoutées	$6.8 \cdot 10^6$	171	$1.1 \cdot 10^7$	$5.2 \cdot 10^7$
	Taux complétion	94%	45%	91%	95%
	Cliques	64	8	84	237
	Grandes cliques	1	1	1	10
	Taille clique max.	3 680	20	4 754	10 030
Bloc	Arêtes ajoutées	$6.4 \cdot 10^6$	147	$1 \cdot 10^7$	$4.8 \cdot 10^7$
	Taux complétion	89%	39%	83%	87%
	Cliques	190	10	252	626
	Grandes cliques	8	1	9	32
	Taille clique max.	3 595	19	4 636	9 813

		rice	sacchr12	sacchr3
Complet	Arêtes ajoutées	14 028	$1.6 \cdot 10^6$	$1.7 \cdot 10^7$
	Taux complétion	100%	100%	100%
Cluster	Arêtes ajoutées	10 704	$1.4 \cdot 10^6$	$1.6 \cdot 10^5$
	Taux complétion	76%	88%	94%
	Cliques	12	45	8
	Grandes cliques	2	1	1
	Taille clique max.	148	1 694	578
Bloc	Arêtes ajoutées	9 687	$1.3 \cdot 10^6$	$1.6 \cdot 10^5$
	Taux complétion	69%	81%	94%
	Cliques	12	114	24
	Grandes cliques	3	5	1
	Taille clique max.	141	1 660	569

de clusters connectés, cette version produit des solutions au moins aussi bonnes que celles retournées par la version utilisant les graphes complets. Les trois algorithmes gloutons trouvent la même solution pour sept instances : `ebola`, `gloeobacter`,

`lactobacillus`, `monarch`, `pandora`, `pseudomonas` et `rice`. Les versions utilisant des graphes de clusters connectés et les graphes de blocs retournent de meilleures solutions pour les quatre autres instances : `anopheles`, `anthrax`, `sacchr3` et `sacchr12`. La version utilisant les graphes de blocs améliore la solution de la version utilisant les graphes de clusters connectés pour une instance `anopheles`. Cela tend à vérifier notre hypothèse. Cependant, il est à noter que les différences de facteur entre les différents algorithmes gloutons sont très faibles : par exemple pour `anthrax`, la valeur de l'amélioration est égale à $4 \cdot 10^{-6}$. On peut expliquer cette faible amélioration par le fait que les graphes complétés restent très denses et contiennent une clique qui concentre au moins 90% des sommets du graphe.

Pour les temps de calcul, on trouve des résultats cohérents avec les complexités théoriques : les solutions des graphes de clusters connectés sont plus lents à calculer que les solutions des graphes complets. Une bonne nouvelle vient du fait que l'on reste dans des temps de calcul raisonnables, même pour les grandes instances comme `anopheles` qui nécessite moins de deux minutes de calculs.

3.3.5 Formulation SAT

Pour finir cette section dédiée aux tests, disons un mot sur l'implémentation de la fonction de faisabilité utilisant une formulation SAT. Nous avons implémenté la formulation présentée dans la sous-sous-section 3.2.3.1 n'incluant pas les cycles alternés. Pour trouver une assignation satisfaisant la formule booléenne produite, nous avons utilisé deux solveurs différents : `cryptominisat` [81] et `glucose` (basé sur le solveur `MiniSat`) [10]. Malheureusement, cette implémentation n'est pas utilisable sur les instances réelles car cela produit des formules booléennes trop grandes. À titre d'exemple, la formule produite pour l'instance `sacch3` comprend $411\ 375$ variables booléennes et $229\ 829\ 007$ clauses. L'espace mémoire nécessaire est trop important pour que les résultats soient obtenus. Il faudra effectuer un certain nombre de pré-traitements et d'optimisations pour potentiellement pouvoir utiliser cette formulation.

3.4 Conclusion et perspectives sur l'échafaudage

Dans ce chapitre, nous avons formulé des résultats à la fois de complexité et d'approximation sur le problème de l'échafaudage. Pour conclure cette partie sur l'échafaudage, nous abordons quelques perspectives pour étendre ces travaux.

3.4.1 Complexité

Dans la première section, nous avons montré des résultats de complexité. Ces résultats permettent d'indiquer quels calculs de solutions sont inaccessibles en temps polynomial. Déterminer la frontière entre les classes de graphes pour lesquelles on

TABLE 3.7 – *Résultats des différentes versions de l'algorithme glouton. La valeur donnée par la colonne « Facteur » correspond au quotient du poids optimal par le poids approché. Les temps sont donnés en secondes. Les cellules vertes indiquent là où le poids est meilleur pour les nouvelles versions du graphe glouton. La cellule bleue indique là où la version des graphes de blocs est meilleure que la version des graphes de clusters connectés.*

	Complet			Cluster		
Donnée	Score	Temps	Facteur	Score	Temps	Facteur
anopheles	1 707 529	2.90	1.0171	1 707 759	99.91	1.017
anthrax	226 709	0.26	1.00598	226 712	0.60	1.00596
ebola	776	0.00	1	776	0.00	1
gloeobacter	218 602	0.29	1.00881	218 602	0.90	1.00881
lactobacillus	95 497	0.12	1.00854	95 497	0.22	1.00854
monarch	506	0.00	1.002	506	0.00	1.002
pandora	119 599	0.16	1.0093	119 599	0.31	1.0093
pseudomonas	279 607	0.32	1.0049	279 607	1.18	1.0049
rice	4 293	0.00	1.00629	4.293	0.01	1.00629
sacchr3	14 52	0.02	1.00682	14 531	0.03	1.00633
sacchr12	46 041	0.05	1.00769	46 050	0.07	1.00749

	Bloc			ILP	
Donnée	Score	Temps	Facteur	Score	Temps
anopheles	1 707 762	160.77	1.0168	1 736 748	3600 >
anthrax	226 712	0.96	1.00596	228 064	26.22
ebola	776	0.00	1	776	0.01
gloeobacter	218 602	1.38	1.00881	220 527	14.86
lactobacillus	95 497	0.27	1.00854	96 313	2.48
monarch	506	0.00	1.002	507	0.01
pandora	119 599	0.48	1.0093	120 710	3.85
pseudomonas	279 607	1.81	1.0049	280 978	19.72
rice	4 293	0.01	1.00629	4 320	0.02
sacchr3	14 531	0.03	1.00633	14 623	0.15
sacchr12	46 050	0.09	1.00749	46 395	1.18

peut calculer une solution en temps polynomial et celles ou ce n'est pas le cas est important, notamment en vue de construire une fonction de faisabilité pour l'algorithme glouton. En effet, si l'on souhaite garder l'exécution de cet algorithme dans un temps polynomial, il faut nécessaire l'exécuter sur une classe de graphe où une fonction de faisabilité est polynomiale. Il est ainsi inutile de chercher à construire une fonction de faisabilité polynomiale pour une classe de graphes où ÉCHAFAUDAGE a été montré NP-complet. Nous avons donc montré que ÉCHAFAUDAGE était NP-complet quand $|M^*| = g^* \cdot \sigma_c + \sigma_p$ car dans ce cas chacun des cycles alterné correspond à une maille du graphe d'échafaudage et chaque chemin alterné est constitué d'une unique arête de couplage.

Une perspective de recherche vient d'un questionnement provenant du résultat du théorème 8 : ici, on a un problème d'optimisation pour lequel trouver une solution, même non optimale, est difficile. Et même si l'on se restreint aux classes de graphes pour lesquelles trouver n'importe quelle solution est trivial (comme pour les graphes

complets), trouver une solution optimale est compliqué. On a montré que ÉCHAFAUDAGE MAXIMUM est poly-APX-difficile pour un certain ensemble d'instances : cela veut dire que si l'on complique un peu la recherche de solution, le problème passe de APX-complet à poly-APX-difficile. On peut se demander si ce comportement n'est pas le même pour d'autres problèmes.

Formellement, on peut formuler ce questionnement de la façon suivante. Soit Π un problème d'optimisation consistant à trouver une solution optimisant un critère $score : \Sigma_\Pi \mapsto \mathbb{R}^+$ et tel que calculer n'importe quelle solution pour les instances dans Σ_Π est NP-complet. Dans ce cas, comme les mêmes arguments que ceux donnés pour la preuve du théorème 7 s'appliquent, Π n'est pas approximable en temps polynomial, sauf si P = NP. Supposons qu'il existe un sous-ensemble d'instances $\Sigma'_\Pi \subset \Sigma_\Pi$ pour lequel calculer une solution peut être fait en temps polynomial mais pour lequel trouver une solution optimale est NP-difficile. La question serait : existe-t-il un ensemble d'instances $\Sigma''_\Pi \subset \Sigma_\Pi$ pour lequel Π serait poly-APX-difficile ?

3.4.2 Algorithme glouton

Nous avons également décrit une fonction de faisabilité permettant d'exécuter l'algorithme glouton dans les graphes de clusters connectés. Les graphes de cette classe sont plus proches des instances réelles que les graphes complets. En effet, on a besoin d'ajouter moins d'arêtes de poids nul pour compléter les instances réelles en graphes de clusters connectés que pour les compléter en graphes complets. Nous avons implémenté et testé cette fonction de faisabilité sur les graphes de clusters connectés ainsi qu'une fonction de faisabilité sur une classe de graphes proche : les graphes de blocs. L'idée qu'il y avait derrière ces tests était de vérifier l'hypothèse suivante : est-ce que utiliser une fonction de faisabilité permettant de compléter le graphe avec moins d'arêtes de poids nul permet d'obtenir une meilleure solution sur des instances réelles avec l'algorithme glouton ?

Nous avons vu que cette hypothèse semblait plutôt vérifiée, mais que les résultats n'étaient pas significativement améliorés dans les graphes de clusters connectés et les graphes de blocs par rapport aux graphes complets. On peut penser que cela est principalement dû au fait que les graphes de clusters connectés sont des graphes denses et donc que le nombre d'arêtes de poids nul à ajouter reste important. Une prolongation naturelle de ce travail est de trouver une fonction de faisabilité qui s'approche davantage des instances réelles. S'il semble compliqué de trouver une classe de graphes caractérisant les instances réelles, une approche pourrait être de trouver des algorithmes calculant des entrées de table de même format que celui des entrées de table de l'algorithme sur les graphes de cluster connectés. Ainsi, on pourrait inclure ces algorithmes dans l'algorithme dynamique ascendant, comme le montre la figure 3.16.

Une autre approche pourrait être d'utiliser l'aléatoire : au lieu de parcourir les arêtes par ordre décroissant de poids, on pourrait les parcourir de façon aléatoire, en donnant toutefois une meilleure probabilité d'être choisies pour les arêtes ayant un poids élevé En déroulant plusieurs fois l'algorithme randomisé, on pourrait atteindre un meilleur facteur d'approximation. Dans la littérature, le facteur d'approximation

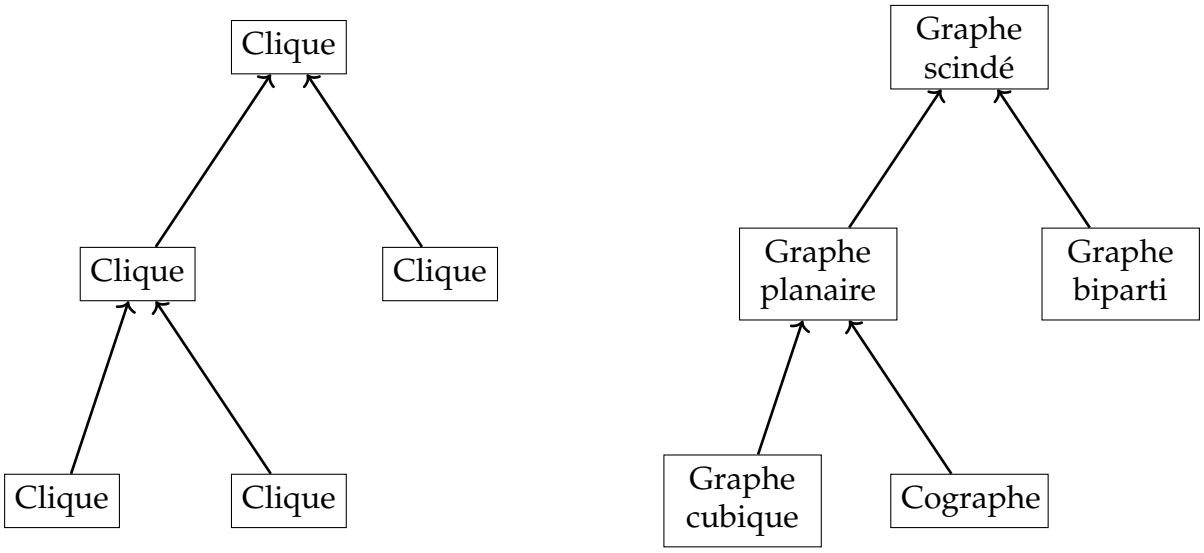


FIGURE 3.16 – *Exemple d'adaptation de l'algorithme de faisabilité sur les graphes de clusters connectés, au lieu de considérer uniquement des cliques dans l'arbre, on pourrait considérer d'autres classes de graphes, cela permettrait d'avoir à ajouter moins d'arêtes pour compléter le graphe.*

pour les graphes complets a été amélioré à $\frac{9-8\epsilon}{5-5\epsilon}$ pour toute constante ϵ entre 0 et 1 en utilisant une approche probabiliste [22].

IV

Linéarisation

Dans ce chapitre, nous présentons des résultats de complexité, d'approximation et des méthodes exactes pour le problème de COUPE SEMI-BRUTALE, défini dans la section 2.4. On rappelle les deux fonctions de score, définies dans la sous-section 2.4.4, que l'on va utiliser pour évaluer une solution.

Score de coupe. Minimiser le nombre de coupes : $score(X) = |X|$.

Score de poids. Minimiser le poids total des arêtes retirées :
$score(X) = \sum\{m'(uv) \cdot \omega(uv) \mid uv \in E(G_s^*) \setminus M^* \wedge \{u, v\}\} \cap X = \varnothing\}$.

Le tableau 2.3 et le tableau 2.4 dans le chapitre 2 récapitulent les résultats de complexité et d'approximation qui seront montrés dans ce chapitre.

Dans la section 4.1, nous commencerons par présenter quelques cas pouvant être résolus de façon polynomiale. Ensuite, nous explorerons dans la section 4.2 la complétude de ce problème dans NP. Des résultats d'inapproximabilité et d'approximation seront présentés dans la section 4.3 et la section 4.4, respectivement. La section 4.5 montrera comment le problème peut être résolu à l'aide de méthodes exactes. Enfin, nous finirons ce chapitre par la section 4.6 qui exposera des statistiques issues de tests sur des instances réelles.

4.1 Cas polynomiaux

Intéressons-nous tout d'abord aux cas « faciles », c'est-à-dire aux classes de graphes pour lesquels COUPE SEMI-BRUTALE peut-être résolu en temps. Nous explorerons à la fois des classes de graphes peu denses et des classes de graphes denses. Dans les graphes peu denses explorés, COUPE SEMI-BRUTALE peut être résolu en temps polynomial pour les deux fonctions de score car le nombre de choix à effectuer est faible, ce qui permet de ne pas faire exploser le nombre de combinaisons. Pour les graphes denses, le problème est facile uniquement pour le score de coupe. Cela provient du

fait que l'on doit pratiquement couper une extrémité par arête couplante dans ce type de graphe et donc que le nombre de choix à faire est également restreint pour cette fonction de score.

4.1.1 Graphes peu denses

Parmi les graphes peu denses, nous allons étudier deux classes de graphes pour lesquelles le problème COUPE SEMI-BRUTALE peut être résolu en temps polynomial : les arbres et les graphes de degré maximum au plus égal à deux.

4.1.1.1 Arbres

Dans un premier temps, intéressons-nous aux graphes de solution (G_s^*, M^*, ω) tels que le graphe G_s^* est un arbre. Dans [88], il a été décrit un algorithme ascendant permettant de calculer une solution optimale pour le score de poids. Nous compléterons ce résultat à la fois en présentant la preuve formelle de l'exactitude de cet algorithme et une adaptation de celui-ci, afin qu'il puisse calculer une solution optimale pour le score de coupe.

On suppose que l'arbre G_s^* est enraciné en une extrémité d'une arête ambiguë. Notons que par l'application de la règle 5, si une arête n'est pas ambiguë, alors elle est incidente à une feuille. Soit v un sommet, soit T' le sous-arbre enraciné en v, on dénote par T_v le sous-graphe induit par les sommets $V(T') \cup \{M^*(v)\}$ et par $enf(v)$ la liste des enfants de v dans l'arbre. Dans un premier temps considérons le score de poids. Pour ce sommet, on va calculer deux entrées de table : $c(v)$ et $\bar{c}(v)$. L'entrée de table $c(v)$ représente le score minimum d'une solution dans T_v dans laquelle v est de degré un et l'entrée de table $\bar{c}(v)$ représente le score minimum d'une solution dans T_v dans laquelle $M^*(v)$ est de degré un. Pour faciliter l'écriture de l'algorithme dynamique, nous considérons que pour toute arête de couplage e, on a $\omega(e) = 0$ (*i.e.* on donne un poids nul à toutes les arêtes couplantes). Si v ou $M^*(v)$ est une feuille, on assigne les valeurs $c(v) = \bar{c}(v) = 0$. Sinon, on assigne :

$$c(v) = \sum_{u \in enf(v)} min(\bar{c}(u), c(u)) + \omega(uv)$$

$$\bar{c}(v) = \sum_{u \in enf(v)} \begin{cases} c(u) & \text{si } u = M^*(v); \\ min(\bar{c}(u), c(u) + \omega(uv)) & \text{sinon.} \end{cases}$$

Lemme 10

Pour chaque sommet v, les valeurs des entrées de table $c(v)$ et $\bar{c}(v)$ représentent le score de poids minimum dans T_v quand v ou $M^*(v)$ est de degré un, respectivement.

Preuve.

Faisons une preuve par induction sur la hauteur de l'arbre. Soit v un sommet de l'arbre. On dénote par $h(v)$ la hauteur de l'arbre T_v. Si $h(v) = 1$, v ou $M^*(v)$ est une feuille et donc une solution de COUPE SEMI-BRUTALE ne comprend aucune coupe, son score est de zéro et donc la propriété du lemme est vérifiée. Supposons à présent que $h(v) > 1$ et que pour tout sommet v', les entrées de table $c(v')$ et $\overline{c}(v')$ vérifient la propriété du lemme. Montrons que :

(1) toute solution X de COUPE SEMI-BRUTALE dans T_v a $score(X) \geq c(v)$ si v est de degré un après avoir appliqué X, ou $score(X) \geq \overline{c}(v)$ si $M^*(v)$ est de degré un après avoir appliqué X ;

(2) il existe une solution dans T_v ayant pour score de poids $min(c(x), \overline{c}(v))$.

(1) Soit X une solution de COUPE SEMI-BRUTALE dans T_v. Dénotons par $X_u = V(T_u) \cap X$ la restriction de X aux sommets de T_u, pour tout enfant u de v. X_u est une solution de COUPE SEMI-BRUTALE dans T_u. La hauteur de T_u est plus petite que $h(v)$. Par hypothèse d'induction, on a $score(X_u) \geq c(u)$ si toutes les arêtes non couplantes incidentes à u sont enlevées ou $score(X_u) \geq \overline{c}(u)$ sinon.

— Supposons que nous sommes dans le cas où toutes les arêtes n'appartenant pas au couplage incidentes à v sont enlevées par X. Le score de poids de X correspond à la somme des poids de ces arêtes plus la somme des scores de poids de n'importe quelle solution des enfants :

$$score(X) = \sum_{u \in\ enf(v)} score(X_u) + \omega(uv)$$

En utilisant l'équation précédente et l'hypothèse d'induction, nous avons $score(X) \geq c(x)$.

— Supposons à présent que nous sommes dans le cas où toutes les arêtes non couplantes incidentes à $M^*(u)$ sont enlevées par X. Pour tout enfant u, le score de poids de X correspond au score de poids de X_y plus la valeur de $\omega(uv)$ si u appartient à X_u :

$$score(X) = \sum_{u \in enf(v)} \begin{cases} score(X_u) + \omega(uv) & \text{si } u \in X_u; \\ score(X_y) & \text{sinon.} \end{cases}$$

On distingue $M^*(v)$ parmi les enfants de v.

— Si $M^*(v)$ est un enfant de v, alors toutes les arêtes incidentes sont enlevées par $X_{M^*(v)}$ dans $T_{M^*(v)}$. Dans ce cas, $score(X_{M^*(v)}) \geq c(M^*(v))$ par hypothèse d'induction.

— Pour tous les autres enfants u de v, soit $M^*(u)$ est de degré un dans X_u, menant à une suppression de poids égale à $\overline{c}(u)$, soit u est de degré un dans X_u menant à une suppression de poids égale à $c(u) + \omega(uv)$.

En utilisant les équations précédentes et l'hypothèse d'induction, on obtient $score(X) \geq \overline{c}(v)$.

(2) À présent, nous montrons qu'il est possible de construire deux solutions X_v et $\overline{X}_v$ de COUPE SEMI-BRUTALE dans T_v ayant pour score de poids $c(v)$ et $\overline{c}(v)$, respectivement. Soit u un enfant de v. On dénote par X_u une solution de COUPE SEMI-BRUTALE dans T_y avec $score(X_u) = c(u)$ où toutes les arêtes incidentes à u et n'appartenant pas au couplage sont enlevées. De même, on dénote par $\overline{X}_u$ une solution de COUPE SEMI-BRUTALE dans T_y avec $score(\overline{X_u}) = \overline{c}(u)$ où toutes les arêtes incidentes à $M^*(u)$ et n'appartenant pas au couplage sont enlevées. Les deux solutions X_u et $\overline{X}_u$ existent par hypothèse d'induction.

— On définit l'ensemble X_v par

$$X_v = \{v\} \cup \bigcup_{u \in enf(v)} \begin{cases} X_u & \text{si } c(u) < \overline{c}(u); \\ \overline{X}_u & \text{sinon.} \end{cases}$$

Comme les ensembles X_u et $\overline{X}_u$ sont des solutions de COUPE SEMI-BRUTALE dans T_u, ils rendent propres toutes les arêtes ambiguës incidentes aux descendants de u. Enlever les arêtes n'appartenant pas au couplage incidentes à v rend propre l'arête ambiguë $vM^*(v)$ dans T_v. Ainsi l'ensemble X_v est une solution de COUPE SEMI-BRUTALE dans T_v et cette solution a un score de poids égal à $c(v)$.

— On définit l'ensemble $\overline{X}_v$ par

$$\overline{X}_v = \bigcup_{u \in enf(v)} \begin{cases} X_u & \text{si } c(u) + \omega(uv) \leq \overline{c}(u) \text{ ou } u = M^*(v); \\ \overline{X}_u & \text{sinon.} \end{cases}$$

Comme les ensembles X_u et $\overline{X}_u$ sont des solutions de COUPE SEMI-BRUTALE dans T_u, ils rendent propres toutes les arêtes ambiguës incidentes aux descendants de u. Si $M^*(v)$ est un descendant de v, alors une solution enlevant toutes les arêtes n'appartenant pas au couplage incidentes à $M^*(v)$ dans $T_{M^*(v)}$ rend propre l'arête $M^*(v)v$. Pour tout autre enfant u de v, soit u appartient à $\overline{X}_v$ soit toutes les arêtes incidentes à $M^*(v)$ sont enlevées par $\overline{X}_v$. $\overline{X}_v$ rend propre $M^*(u)u$ et donc $\overline{X}_u$ est une solution de COUPE SEMI-BRUTALE dans T_u ayant un score de poids égal à $\overline{c}(v)$.

□

On peut à présent modifier l'algorithme présenté pour trouver une solution possédant un score de poids optimal afin que celui-ci trouve une solution possédant un score de coupe optimal. Pour cela, pour chaque sommet v, nous ajoutons une troisième entrée de table $n(v)$. Cette entrée va prendre en valeur le score de coupe minimum d'une solution dans T_v dans laquelle tous les voisins de v à l'exception de $M^*(v)$ sont dans la solution. L'entrée de table $c(v)$ va alors prendre en valeur le score de coupe minimum d'une solution dans T_v contenant v. Si v ou $M^*(v)$ est une feuille alors on assigne les valeurs $c(v) = \infty$ et $\overline{c}(v) = n(v) = 0$. La valeur infinie est assignée dans ce cas car le

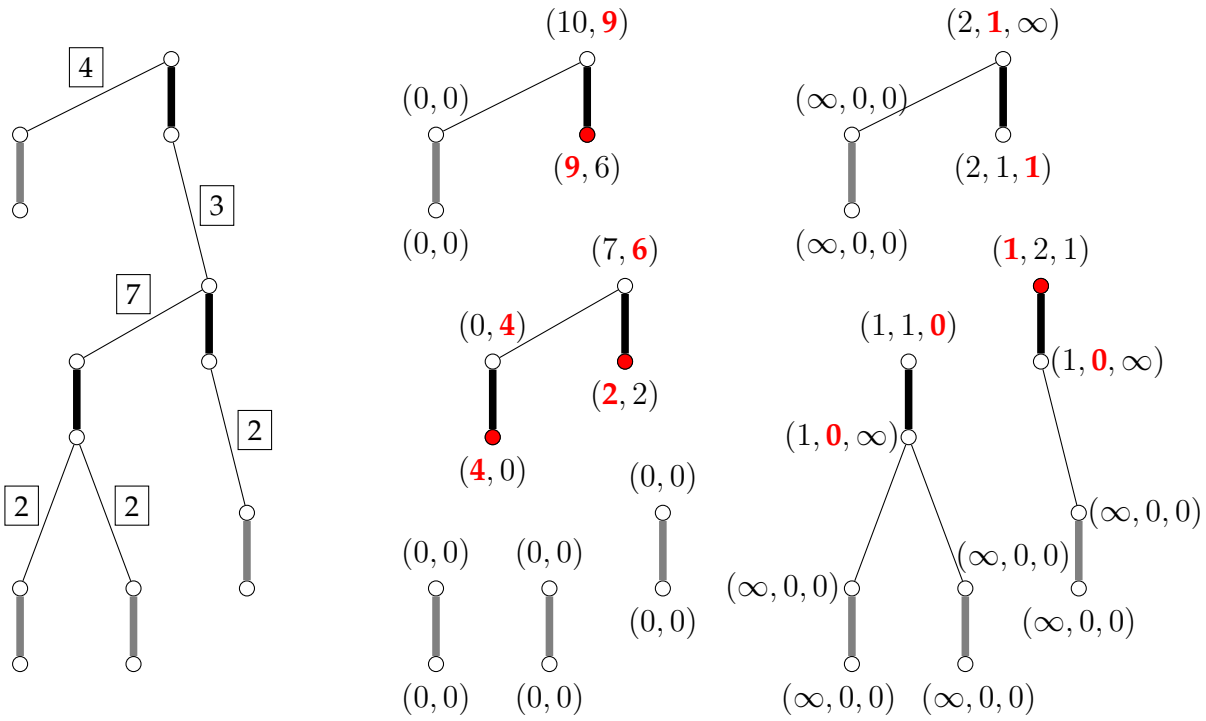


FIGURE 4.1 – *Exemple de l'application de l'algorithme dynamique sur les arbres. Les arêtes de couplage sont en gras, les arêtes ambiguës sont dessinées en noir tandis que les arêtes non ambiguës sont dessinées en gris.* ***Gauche :*** *Graphe de solution de départ, les poids des arêtes sont représentés par les nombres encadrés sur les arêtes.* ***Milieu :*** *Résolution de* COUPE SEMI-BRUTALE *pour le score de poids. À côté de chaque sommet les entrées de table sont représentées par un vecteur où le premier élément contient la valeur de c et le second élément la valeur de $\overline{c}$. Les nombres en rouge représentent les valeurs choisies lors du retour en arrière. Les sommets en rouge représentent les sommets coupés dans une solution optimale.* ***Droite :*** *Résolution de* COUPE SEMI-BRUTALE *pour le score de coupe. À côté de chaque sommet les entrées de table sont représentées par un vecteur où le premier élément contient la valeur de c, le second élément la valeur de $\overline{c}$ et le dernier élément la valeur de n. Les nombres en rouge représentent les valeurs choisies lors du retour en arrière. Le sommet constitue une solution optimale.*

sommet v ne peut pas appartenir à une solution par définition. Sinon, on assigne :

$$c(v) = \sum_{u \in enf(v)} min(c(u), \overline{c}(u), n(u)) + 1,$$

$$\overline{c}(v) = \sum_{u \in enf(v)} \begin{cases} min(c(u), n(u)) & \text{si } u = M^*(v); \\ min(c(u), \overline{c}(u)) & \text{sinon;} \end{cases}$$

$$n(v) = \sum_{u \in enf(v)} \begin{cases} \overline{c}(u) & \text{si } u = M^*(v); \\ c(u) & \text{sinon.} \end{cases}$$

La valeur ∞ va être assignée à l'entrée $n(v)$ si v est adjacente à un sommet u tel que l'arête de couplage $M^*(u)u$ n'est pas ambiguë. On peut facilement adapter la preuve utilisée pour le lemme 10 pour prouver le lemme suivant, nous laisserons au lecteur le soin d'observer la véracité de celui-ci.

Lemme 11

Pour chaque sommet v, les valeurs des entrées de table $c(v)$, $\overline{c}(v)$ et $n(v)$ représentent le score de coupe minimum dans T_v quand v appartient à la solution, $M^*(v)$ est de degré un et tous les voisins sauf $M^*(v)$ appartiennent à la solution, respectivement.

Les deux lemmes précédents mènent au résultat suivant.

Théorème 11

COUPE SEMI-BRUTALE peut être résolu en temps et en espace linéaire dans les arbres pour les deux fonctions de score.

Preuve.

Par le lemme 10 (resp. lemme 11), on peut calculer les valeurs des entrées de table $c(r)$ et $(\overline{c}(r))$ (resp. $c(r)$, $\overline{c}(r)$ et $n(r)$) correspondantes pour la racine r. En prenant la valeur minimum parmi ces entrées, il suffit de revenir en arrière pour choisir les coupes ayant mené au calcul de cette valeur (*i.e.* v est dans la solution si et seulement si $c(v)$ a été choisie lors du calcul ascendant). □

4.1.1.2 Chemins et cycles

Traitons à présent le cas des graphes de solution ayant un degré maximum au plus deux : c'est-à-dire les graphes de solution qui correspondent à une union disjointe de chemins et de cycles. Comme l'on peut traiter séparément chaque composante connexe d'un graphe de solution sans incidence sur les autres, on supposera que le graphe de solution est soit constitué d'un unique chemin ou d'un unique cycle.

S'il est constitué d'un unique chemin, alors le résultat du théorème 11 permet d'obtenir un algorithme en temps linéaire sur le graphe. S'il est constitué d'un unique cycle C, alors, on choisit deux arêtes non couplantes e_1 et e_2 incidentes à une même arête de couplage. Comme au moins une de ces arêtes est enlevée dans n'importe quelle solution, il suffit de calculer les deux solutions optimales dans les sous-graphes $C \setminus \{e_1\}$ et $C \setminus \{e_2\}$ et de choisir la meilleure en prenant en compte le poids de l'arête enlevée. Cela nous mène assez trivialement au résultat suivant.

Théorème 12

COUPE SEMI-BRUTALE peut être résolu en temps linéaire pour les deux fonctions de score dans les graphes de solution (G_s^*, M^*, ω) où $\Delta(G_s^*) \leq 2$.

Nous verrons par la suite dans la sous-section 4.3.1 que COUPE SEMI-BRUTALE est NP-complet pour le cas où $\Delta \geq 3$.

4.1.2 Graphes denses

Dans certaines classes de graphes denses, on peut montrer que le problème COUPE SEMI-BRUTALE peut également être résolu en temps polynomial pour le score de coupe. Pour le score de poids, nous verrons plus tard que ce problème est NP-difficile pour une très grande partie des classes de graphes denses. Pour les preuves qui vont suivre, il est peut être important de noter qu'il est toujours possible de résoudre COUPE SEMI-BRUTALE avec $|M^*|$ coupes en coupant arbitrairement un sommet par arête de couplage (dans le cas où toutes les arêtes de couplage sont ambiguës). Dans les graphes denses, ce problème devient solvable en temps polynomial car la plupart des arêtes de couplage doivent contenir une coupe. On va s'intéresser à trois classes de graphes denses : les graphes complets, les graphes bipartis complets et les cographes.

Théorème 13

COUPE SEMI-BRUTALE peut être résolu en temps linéaire pour le score de coupe dans les graphes complets.

Preuve.

Soit (K_n, M^*, ω) un graphe de solution tel que K_n est un graphe complet à n sommets. Si $n = 2$, le couplage ne contient qu'une arête. Cette arête est déjà propre et il n'y a rien à faire. Supposons que $n > 2$. S'il existe une solution X pour laquelle une arête uv ne contient pas de coupe, alors soit tous les autres voisins de u sont coupés ou tous les autres voisins de v sont coupés. Cela implique que $|X| = |V(G_s^*)| - 2 > |M^*|$. Ainsi X n'est pas optimale et donc $|M^*|$ coupes sont nécessaires. Une solution contenant une coupe par arête de couplage est optimale pour le score de coupe. □

Théorème 14

COUPE SEMI-BRUTALE peut être résolu en temps linéaire pour le score de coupe dans les graphes bipartis complets.

Preuve.

Notons que dans les graphes de solution bipartis, les deux parties de la bipartition sont de même taille et le couplage forme une bijection entre de ces deux parties. Soit $(K_{n,n}, M^*, \omega)$ un graphe de solution où $K_{n,n}$ est un graphe biparti complet (avec $n = |M^*|$). Si $n = 1$, alors le graphe est constitué d'une seule arête de couplage qui est déjà propre. Supposons que $n \geq 2$. Coupons tous les sommets d'un côté de la bipartition à l'exception d'un seul sommet que l'on dénotera u. Comme tous les voisins de $M^*(u)$ à l'exception de u ont été coupés, l'arête $M^*(u)u$ est propre. Comme toutes les autres arêtes du graphe contiennent une coupe, on a montré que $n - 1$ coupes sont suffisantes pour rendre propres toutes les arêtes de couplage du graphe. Pour montrer que $n-1$ coupes sont également nécessaires, supposons qu'il existe une solution X contenant $n-2$ coupes.

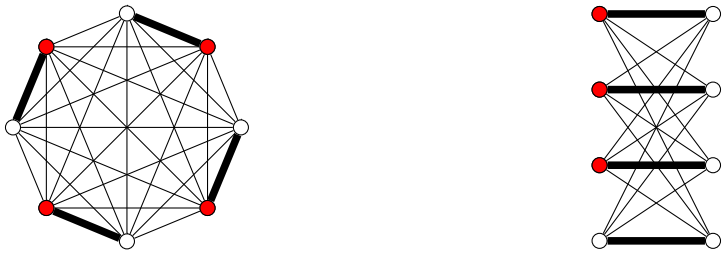


FIGURE 4.2 – *Exemples de linéarisation d'un graphe complet et d'un graphe biparti complet. Les sommets rouges correspondent aux sommets coupés.*

Comme il y a n arêtes de couplage dans le graphe, il existe deux arêtes de couplage uv et xy qui n'ont pas de sommet dans X. Comme $K_{n,n}$ est un graphe biparti complet, le graphe induit par les sommets $\{u, v, x, y\}$ forme un cycle et donc les arêtes de couplage uv et xy ne sont pas propres. □

Des exemples de solutions pour les deux classes de graphes précédentes sont exposés par la figure 4.2.

Théorème 15

COUPE SEMI-BRUTALE peut être résolu en temps linéaire pour le score de coupe dans les cographes.

Preuve.

Soit (G_s^*, M^*, ω) un graphe de solution où G_s^* est un cographe. Si $|M^*| = 1$, alors l'unique arête constituant le graphe est déjà propre. Si $|M^*| = 2$, alors G_s^* est soit un chemin, soit un triangle avec un sommet pendant (*i.e.* un sommet du triangle est relié à un sommet de degré un), soit un cycle de taille quatre avec éventuellement une corde. Dans le premier cas, il n'y pas d'arêtes ambiguës, dans le deuxième cas, couper une extrémité de la seule arête ambiguë est une solution optimale et enfin, dans le dernier cas, couper un sommet auquel la corde est incidente est une solution optimale.
Supposons à présent que $|M^*| > 2$. Soit (V_1, V_2) la partition des sommets de G_s^* en deux cliques. Dans un premier temps, supposons qu'il existe une solution X avec $|M^*| - 2$ coupes. Il existe donc deux arêtes de couplage uv et xy n'ayant pas de sommets dans X. Si une des deux parties V_1 ou V_2 contiennent au moins trois sommets parmi $\{u, v, x, y\}$, disons u,v et x, alors l'arête uv n'est pas propre car ses deux extrémités sont adjacentes à x. Ainsi chacune des deux parties contient exactement deux sommets parmi $\{u, v, x, y\}$. Si une des parties contient ux et l'autre partie contient vy, alors les sommets $\{u, v, x, y\}$ induisent un cycle et aucune des arêtes uv et xy n'est propre. Ainsi, sans perte de généralité, $\{u, v\} \subseteq V_1$ et $\{x, y\} \subseteq V_2$. Dans ce cas, tous les sommets de $V(G_s^*) \setminus \{u, v, x, y\}$ doivent être coupés, impliquant une solution contenant $2(|M^*| - 2)$ coupes, ce qui représente une contradiction. Ainsi, aucune solution X de taille $|M^*| - 2$ ne peut exister.
Comme trouver une solution de taille $|M^*|$ est trivial, il reste à traiter les cas où il est possible de trouver une solution de taille $|M^*| - 1$. C'est-à-dire, les cas où il

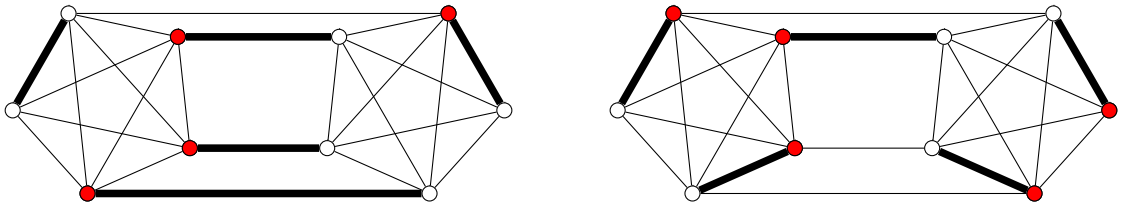


FIGURE 4.3 – *Deux exemples de cographes avec leurs solutions* COUPE SEMI-BRUTALE. *Les sommets rouges sont coupés dans une solution optimale et les arêtes de couplage sont dessinées en gras. Le cographe de gauche ne nécessite que $|M^*| - 1$ coupes dans une solution optimale tandis que le cographe de droite nécessite $|M^*|$ coupes dans une solution optimale.*

existe une arête de couplage uv telle que l'on peut couper tous les sommets $N^*(u)$ pour économiser une coupe sur l'arête uv. Dans ce but, montrons qu'on peut résoudre COUPE SEMI-BRUTALE dans G_s^* avec $|M^*| - 1$ coupes si et seulement si il existe une arête de couplage uv ayant au moins une extrémité, disons u, telle qu'il n'existe pas d'arête de couplage xy telle que $\{x, y\} \subset N^*(u)$.

- $\Rightarrow$ S'il existe une solution X pour G_s^* avec $|X| = |M^*| - 1$, alors il y a exactement une arête de couplage uv n'ayant pas de sommets dans X et chacune des autres arêtes de couplage contient exactement un sommet coupé. Sans perte de généralité, supposons que tous les sommets de $N^*(u) \subseteq X$. Comme toutes les arêtes couplantes à l'exception de uv contiennent une unique coupe, il ne peut donc pas exister une arête de couplage xy avec $\{x, y\} \subset X$ et donc $\{x, y\} \not\subset N(u)$.
- $\Leftarrow$ Soit Q un ensemble de sommets contenant un sommet arbitraire dans chacune des arêtes de couplage xy telles que $\{x, y\} \cap N(u) = \varnothing$ (*i.e.* Q contient exactement un sommet par arête couplante qui n'est pas adjacente à uv). Prenons l'ensemble $X = Q \cup N^*(u)$. On a $|Q| = |M^*| - |N^*(u)| - 1$ et donc $|X| = |Q| + |N^*(u)| = |M^*| - 1$. Toutes les arêtes de couplage à l'exception de uv contiennent un sommet dans X. Comme $N^*(u) \subseteq X$, l'arête uv est propre et donc X est une solution.

On peut vérifier en temps linéaire si une telle arête couplante uv existe et donc le théorème est vérifié. □

Pour aider à la compréhension de la preuve précédente, on peut observer la figure 4.3 qui donne deux exemples particuliers de solutions optimales dans des cographes.

4.2 Difficulté de calcul

Après avoir vu des cas « faciles » dans la section précédente, nous montrons que dans l'ensemble, le problème COUPE SEMI-BRUTALE est difficile à résoudre. Nous ferons ici aussi la distinction entre les classes de graphes peu denses et les classes de graphes denses. Pour les graphes peu denses, nous allons montrer que ce problème

est NP-difficile et PLS-complet à l'aide de réductions à partir de certains problèmes de SATISFAISABILITÉ. D'autres résultats sur la paramétrisation de la complexité viendront compléter ceux donnés par les réductions. Enfin, pour les graphes denses, nous montrerons que pour le score de poids, COUPE SEMI-BRUTALE reste difficile. Nous montrerons également que le score de coupe peut s'avérer compliqué à calculer dans certains graphes denses en montrant qu'il est NP-difficile de trouver une solution optimale avec cette fonction de score dans les graphes scindés.

4.2.1 Graphes peu denses

On a vu dans la sous-section 4.1.1 que dans les arbres, les cycles et les chemins le problème pouvait être résolu en temps polynomial. Nous montrons à présent que si l'on se place dans des graphes avec légèrement plus de densité, c'est-à-dire les graphes ayant un degré maximum au plus trois, le problème devient difficile à résoudre. Nous montrons l'appartenance de COUPE SEMI-BRUTALE aux classes NP-difficile et PLS-complet.

4.2.1.1 Difficulté dans la classe NP

Dans un premier temps, montrons que le problème COUPE SEMI-BRUTALE est NP-difficile dans les graphes planaires, sous-cubiques et bipartis, pour les deux fonctions de score. Pour cela, nous allons créer une réduction à partir de 3-SATISFAISABILITÉ. Nous allons utiliser la construction suivante.

Construction 5

Étant donné une instance φ de 3-SATISFAISABILITÉ comprenant n variables $x_1, \dots, x_n$ et m clauses $C_1, \dots, C_m$, nous produisons le graphe de solution (G_s^*, M^*, ω) suivant, associé à la bipartition (V_1, V_2).

- Pour chaque variable x_i, construire un cycle c_i de longueur $4|\psi|$ sur l'ensemble sommets $\bigcup_{j \leq |\psi_i|} \{u_j^i, \overline{u}_j^i, v_j^i, \overline{v}_j^i\}$, tel que pour tout $j \leq |\psi_i|$:
  - $u_j^i \overline{u}_j^i, v_j^i \overline{v}_j^i \in M^*$;
  - les sommets u_j^i et v_j^i appartiennent à V_1 et les sommets $\overline{u}_j^i$ et $\overline{v}_j^i$ appartiennent à V_2.
- Pour chaque clause C_ℓ, construire un cycle q_ℓ de longueur six sur l'ensemble de sommets $\bigcup_{j \leq 3} \{r_j^\ell, b_j^\ell\}$ tel que pour tout $j \leq 3, r_j^\ell b_j^\ell \in M^*$ et r_j^ℓ appartient à V_2 et b_j^ℓ appartient à V_1.
- Pour toute chaque clause C_ℓ et chaque $j \leq 3$, soit x_i la $j^{ème}$ variable apparaissant dans C_ℓ et soit t l'entier tel que C_ℓ est la $t^{ème}$ clause dans laquelle x_i apparaît. Alors,
  - créer deux arêtes de couplage $a_t^i \overline{a}_t^i$ et $c_t^i \overline{c}_t^i$ où les sommets a_t^i et c_t^i appartiennent à V_1 et les sommets $\overline{a}_t^i$ et $\overline{c}_t^i$ appartiennent à V_2 ;

  - si x_i est un littéral positif de C_ℓ, créer les arêtes $r^\ell_j u^i_t, b^\ell_j \overline{a}^i_t$ et $\overline{u}^i_t c^i_t$;
  - si x_i est un littéral négatif de C_ℓ, créer les arêtes $b^\ell_j \overline{u}^i_t, r^\ell_j a^\ell_j$ et $u^i_t \overline{c}^i_j$.
- Toutes les arêtes n'appartenant pas au couplage ont un poids un, excepté les arêtes $\overline{u}^i_t c^i_t$ et $u^i_t \overline{c}^i_t$ qui ont un poids nul.

On appelle un cycle c_i associé à la variable x_i un *gadget variable* (un exemple est donné par la figure 4.4) et on appelle un cycle q_ℓ associé à la clause C_ℓ un *gadget clause* (un exemple est donné par la figure 4.5). Notons que dans le graphe produit par cette construction, toutes les arêtes de couplage, à l'exception des arêtes $a^\ell_i \overline{a}^\ell_i$ et $c^i_t \overline{c}^i_t$, sont ambiguës.

Le graphe résultant de la construction est de degré maximum trois. Les deux ensembles V_1 et V_2 induisent des stables, ce graphe est donc également biparti. Enfin, contracter chacun des gadgets en un unique sommet permet d'obtenir le graphe G_φ. Comme les arêtes n'appartenant pas au couplage reliant les différents gadgets peuvent être placés dans n'importe quel ordre sur les cycles constituant les gadgets, si la formule donnée en entrée est planaire, alors le graphe de solution produit par la construction est planaire. Dans ce cas, on a bien un graphe biparti, planaire et de degré maximum trois.

Afin de montrer que le problème COUPE SEMI-BRUTALE est NP-difficile dans de tels graphes, il suffit de montrer que la construction 5 constitue une réduction. Pour montrer cela, nous appliquerons cette construction sur des instances appartenant à un sous-ensemble restreint de 3-SATISFAISABILITÉ pour lesquelles la formule booléenne donnée en entrée est également planaire et monotone, ce qui reste NP-complet [32]. Avant cela, formulons quelques propriétés partagées par tout graphe issu de la construction 5.

Lemme 12

Soit φ une instance de 3-SATISFAISABILITÉ MONOTONE PLANAIRE et soit (G^*_s, M^*, ω) son graphe résultant de la construction 5. Soit $X \subseteq V(G^*_s)$ un ensemble de coupes rendant propres toutes les arêtes ambiguës de (G^*_s, M^*, ω). On définit un entier s tel $s = 1$ sous le score de coupe et $s = 2$ sous le score de poids. On suppose que lorsqu'on applique les coupes de X, on commence par couper les sommets des cycles c_i et que ensuite, on coupe les sommets des cycles q_ℓ. Il existe un ensemble X' de coupes qui rend propres toutes les arêtes ambiguës avec $score(X') \leq score(X)$ et tel que pour toute variable x_i et toute clause C_ℓ on a :

(a) $score(X' \cap V(c_i)) \geq s|\psi_i|$ et $score(X' \cap V(q_\ell)) \geq 2$;

(b) si $score(X' \cap V(c_i)) = s|\psi_i|$, alors $X' \cap V(c_i) = \bigcup_{j \leq |\psi_i|} \{u^i_j\}$ ou $X' \cap V(c_i) = \bigcup_{j \leq |\psi_i|} \{\overline{u}^i_j\}$;

(c) $score(X' \cap V(q_\ell)) = 2$ si et seulement si X' contient un sommet adjacent à q_l .

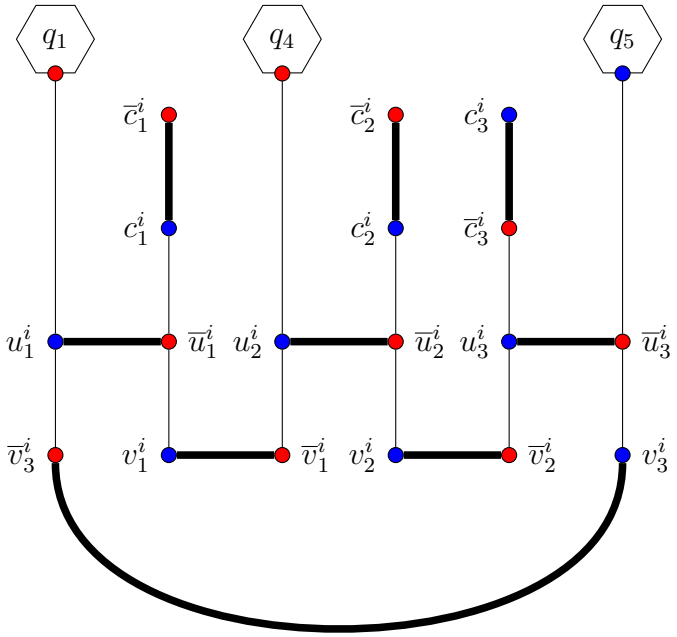


FIGURE 4.4 – *Cycle c_i associé à la variable x_i. Les arêtes de couplage sont dessinées en gras, les sommets colorés en bleus appartiennent à V_1 et les sommets colorés en rouge appartiennent à V_2. Le gadget variable c_i est relié aux gadgets clauses q_1, q_4 et q_5. Dans la formule donnée en entrée, la variable x_i apparaît positivement dans les clauses C_1 et C_4 et négativement dans la clause C_5.*

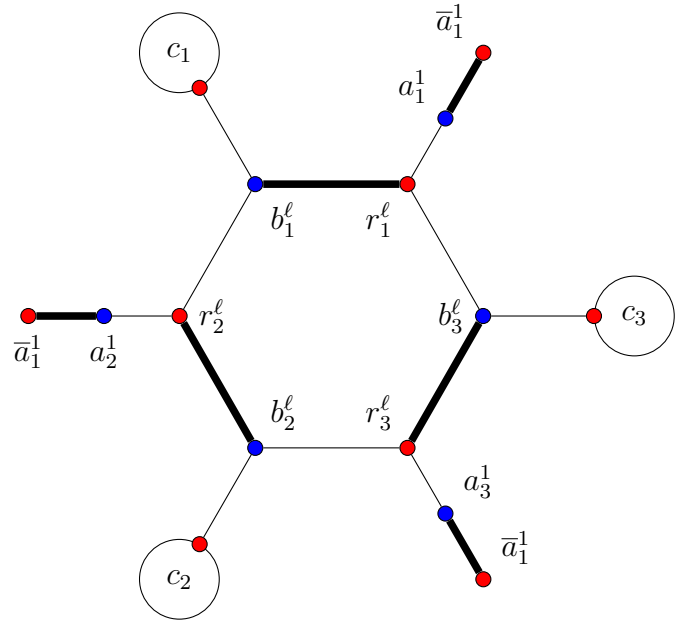


FIGURE 4.5 – *Gadget q_ℓ correspondant à la clause $C_\ell = (x_1 \vee x_2 \vee x_3)$. Les arêtes de couplage sont dessinées en gras. Les sommets colorés en bleu appartiennent à V_1 et les sommets colorés en rouges appartiennent à V_2.*

Preuve.

(a) Dans le cycle c_i, pour chaque $j \leq |\psi_i|$, nous devons enlever deux arêtes pour rendre propre l'arête de couplage $u_j^i \overline{u}_j^i$, ce qui peut être fait unique-

ment en coupant un sommet parmi $\{\overline{v}^i_{j-1}, u^i_j, \overline{u}^i_j, v^i_j\}$. Ainsi, nous devons enlever au moins $2|\psi_i|$ arêtes en utilisant au moins $|\psi_i|$ coupes, et donc $score(X' \cap V(c_i)) \geq s|\psi_i|$. Dans le cycle q_ℓ, nous devons enlever au moins deux arêtes du cycle, ce qui peut être fait en coupant au moins deux sommets. Ainsi, on a $score(X' \cap V(q_\ell)) \geq 2$.

(b) Notons que couper tous les sommets de $\bigcup_{j \leq |\psi_i|} \{u^i_j\}$ ou $\bigcup_{j \leq |\psi_i|} \{\overline{u}^i_j\}$ suffit à rendre propre toutes les arêtes ambiguës de c_i et dans ce cas nous avons $score(X' \cap V(c_i)) = s|\psi_i|$. Si X contient un sommet $\overline{u}^i_j$ mais ne contient pas le sommet $\overline{u}^i_{j+1}$ pour un certain j, alors une coupe supplémentaire est nécessaire pour rendre propre l'arête $v^i_j\overline{v}^i_j$ (et de façon analogue pour u^i_j), ce qui augmente le score de un. Ainsi, si $score(X \cap V(c_i)) = s|\psi_i|$, on peut supposer que X contient soit $\bigcup_{j \leq |\psi_i|} \{u^i_j\}$, soit $\bigcup_{j \leq |\psi_i|} \{\overline{u}^i_j\}$. Si X contient une coupe sur un sommet v^i_j ou $\overline{v}^i_j$, alors comme l'arête $v^i_j\overline{v}^i_j$ est déjà rendue propre par une coupe dans $\{u^i_j, \overline{u}^i_j\}$, on peut enlever ce sommet v^i_j ou $\overline{v}^i_j$ dans l'ensemble X'.

(c) Nous devons enlever, deux arêtes de poids non nul dans le cycle pour rendre propres les trois arêtes ambiguës de q_ℓ. Supposons que tous les littéraux de C_ℓ apparaissent positivement dans la clause. Supposons également, par symétrie, que $\{b^\ell_1, b^\ell_2\} \subset X'$. Si l'arête non couplante n'appartenant pas au cycle incident à r^ℓ_3 est enlevée, alors toutes les arêtes ambiguës de q_ℓ sont propres. Si ce n'est pas le cas, alors on doit enlever une arête de poids non nul avec une coupe supplémentaire dans q_ℓ.

□

Nous pouvons à présent montrer que COUPE SEMI-BRUTALE est NP-difficile. Pour cela, nous utiliserons la deuxième item du lemme précédent pour simuler le comportement d'une variable : dans une solution optimale, on souhaite que seuls les sommets u^i_j soient coupés si l'assignation de la variable est VRAI ou que seuls les sommets $\overline{u}^i_j$ soient coupés si l'assignation de la variable est fausse. Les arêtes non couplantes entre les gadgets clauses « satisfaites » par l'assignation de ce gadget variable sont alors enlevées par la solution. Le troisième item est quant à lui utilisé pour simuler le comportement d'une clause : si la clause est satisfaite, alors cela veut dire qu'au moins une arête non couplante incidente au gadget de cette clause est enlevée et donc que l'on peut rendre propres les arêtes de son cycle avec un score de deux. Dans le cas contraire, il faut utiliser un score de trois pour rendre propres les trois arêtes du cycle.

Théorème 16

COUPE SEMI-BRUTALE est NP-difficile pour les deux fonctions de score, même dans les graphes planaires, bipartis et sous-cubiques.

Preuve.

Rappelons tout d'abord que 3-SATISFAISABILITÉ MONOTONE PLANAIRE est NP-complet [32]. Dans ce cas, comme indiqué plus haut, le graphe produit par la construction 5 est planaire. Soit φ une instance de 3-SATISFAISABILITÉ MO-

NOTONE PLANAIRE et soit (G_s^*, M^*, ω) son graphe de solution produit par la construction 5. Ce graphe de solution peut être produit en temps polynomial. Montrons que φ est satisfaisable si et seulement si son graphe de solution (G_s^*, M^*, ω) produit par la construction 5 admet une solution pour COUPE SEMI-BRUTALE ayant pour score $(3s+2)m$ (où $s = 1$ pour le score de coupe et $s = 2$ pour le score de poids).

- $\Rightarrow$ Soit β une assignation satisfaisante des variables de φ. Construisons une solution X dans (G_s^*, M^*, ω). Pour chaque variable x_i et pour tout $j \leq |\psi_i|$, si son assignation est égale à VRAI, alors nous coupons tous les sommets u_j^i, sinon nous coupons tous les sommets $\overline{u}_j^i$. Comme β est satisfaisante, alors au moins une arête adjacente à chaque cycle q_ℓ est enlevée. Par le lemme 12(c), on peut rendre propres toutes les arêtes de chaque cycle q_ℓ avec un score de deux dans ce cycle. Comme nous avons coupé tous les sommets $\bigcup_{j \leq |\psi_j|} u_j^i$ ou tous les sommets $\bigcup_{j \leq |\psi_j|} \overline{u}_j^i$ dans chaque cycle c_i, toutes les arêtes ambiguës du graphe deviennent propres et nous obtenons une solution ayant pour score $2m + \sum_{i \leq n} s|\psi_i| = (2+3s)m$.
- $\Leftarrow$ Soit $X \subseteq V(G_s^*)$ un ensemble de coupes qui rend propre toutes les arêtes ambiguës de (G_s^*, M^*, ω) et tel que $score(X) = (3s+2)m$. Par le lemme 12(a), pour chaque variable x_i, X a un score de $s|\psi_i|$ dans le cycle c_i et pour chaque clause C_ℓ, X a un score de deux dans le cycle q_ℓ. De plus, par le lemme 12(b), pour chaque cycle c_i, on a $X \cap V(c_i) = \bigcup_{j \leq |\psi_i|} \{u_j^i\}$ ou $X \cap V(c_i) = \bigcup_{j \leq |\psi_i|} \{\overline{u}_j^i\}$. On assigne la valeur VRAI à la variable x_i dans le premier cas et la valeur FAUX dans le second. Pour montrer que l'assignation satisfait φ, supposons qu'il existe une clause C_ℓ qui n'est pas satisfaite. Alors, aucune des arêtes incidentes au cycle q_ℓ n'est enlevée, ce qui d'après le lemme 12(c), contredit le fait que le score dans q_ℓ soit égal à deux.

Ainsi, on a montré que la construction 5 constitue une réduction polynomiale et donc COUPE SEMI-BRUTALE est NP-difficile, même dans les graphes bipartis, planaires et sous-cubiques. □

Un autre résultat concernant la résolution sous-exponentielle du problème COUPE SEMI-BRUTALE peut être dérivé de la construction précédente. Dans [64], il a été montré que si l'hypothèse ETH est vraie, alors 3-SATISFAISABILITÉ ne pouvait être résolu avec un algorithme de complexité sous-exponentielle $\mathcal{O}(2^{o(m)})$ où m correspond au nombre de clauses. Dans le cas planair, il n'existe pas d'algorithme de complexité $\mathcal{O}(2^{\sqrt{m}})$, où n est le nombre de variable [63].

On peut utiliser la construction précédente afin de produire un graphe de solution à partir d'une formule 3-SATISFAISABILITÉ générale. Cela enlève la propriété planaire du graphe. Afin de garder les propriétés du lemme 12 pour le score de poids, il faut que pour toute clause c_ℓ, chacun des gadgets des variables de c_ℓ soient adjacents à un sommet b_j^ℓ ou que chacun des gadgets des variables de c_ℓ soient adjacents à un sommet r_j^ℓ. En faisant cela, le graphe de solution n'est plus biparti.

Le nombre d'arêtes et de sommets dans un graphe de solution produit par la construc-

tion 5 est linéaire par rapport au nombre de clauses de la formule de départ : pour chaque clause, on ajoute exactement seize sommets et vingt-sept arêtes dans le graphe de solution. Ainsi, si l'hypothèse ETH est vérifiée, cela implique directement le résultat suivant.

Corollaire 5

COUPE SEMI-BRUTALE ne peut être résolu en temps :

- $\mathcal{O}(2^{o(|V(G_s^*)|)})$ ou $\mathcal{O}(2^{o(|E(G_s^*)|)})$ même dans les graphes de solutions bipartis et sous-cubiques pour le score de coupe ;
- $\mathcal{O}(2^{o(|V(G_s^*)|)})$ ou $\mathcal{O}(2^{o(|E(G_s^*)|)})$ même dans les graphes de solutions sous-cubiques pour le score de poids ;
- $\mathcal{O}(2^{o(|\sqrt{V(G_s^*)}|)})$ ou $\mathcal{O}(2^{o(|\sqrt{E(G_s^*)}|)})$ même dans les graphes de solutions planaires, bipartis et sous-cubiques pour le score de coupe ;
- $\mathcal{O}(2^{o(|\sqrt{V(G_s^*)}|)})$ ou $\mathcal{O}(2^{o(|\sqrt{E(G_s^*)}|)})$ même dans les graphes de solutions planaires et sous-cubiques pour le score de coupe ;

sauf si ETH est fausse.

4.2.1.2 PLS-complétude

Intéressons nous à présent à la difficulté de trouver un minimum local pour COUPE SEMI-BRUTALE pour le score de poids. Dans cette partie, on ne considéra que les solutions normalisées, c'est-à-dire les ensembles de coupes où chaque arête ambiguë contient exactement une coupe. Pour exprimer COUPE SEMI-BRUTALE sous la forme d'un problème de recherche locale, il faut d'abord définir une fonction de voisinage pour les solutions normalisées. Pour cela, lorsque l'on cherche à créer une solution voisine à partir d'une solution initiale, les coupes de cette solution initiale peuvent être vues comme des jetons que l'on peut faire glisser le long d'une arête de couplage. Formellement, nous nous intéressons à l'opération suivante.

Définition 30

Soit (G_s^*, M^*, ω) un graphe de solution et X une solution de COUPE SEMI-BRUTALE dans (G_s^*, M^*, ω). Soit uv une arête ambiguë de (G_s^*, M^*, ω) avec $u \in X$. L'opération de *glissement de voisinage* appliquée à uv consiste à produire une nouvelle solution X' en suivant ces étapes :

1. $X' \leftarrow (X \cup \{v\}) \setminus \{u\}$;
2. pour chaque voisin $n_u \in N^*(u)$, $X' \leftarrow (X' \cup \{M^*(n_u)\}) \setminus \{n_u\}$;
3. pour chaque voisin $n_v \in N^*(v)$, $X' \leftarrow (X' \cup \{n_v\}) \setminus \{M^*(n_v)\}$.

Montrons que trouver un minimum local pour le problème COUPE SEMI-BRUTALE avec cette fonction de voisinage est dans PLS.

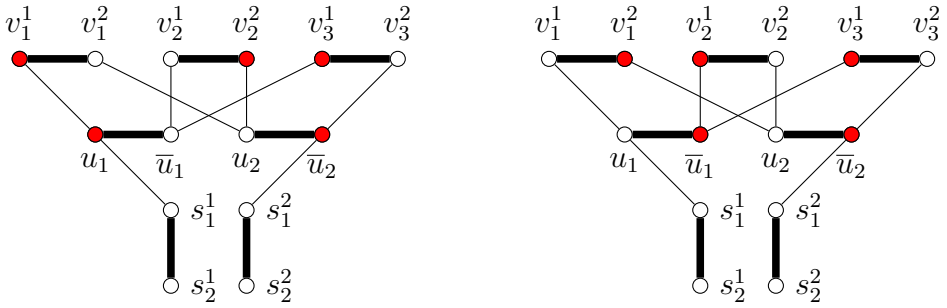


FIGURE 4.6 – *Graphe de solution produit par la construction 6 à partir de la formule booléenne* $\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ *(toute les clauses sont de poids un). Les arêtes de couplage sont dessinées en gras et toutes les arêtes n'appartenant pas au couplage ont un poids de un. À gauche, une solution* X *de score de poids six est montrée par les sommets rouges. Dans* X*, l'arête clause* $v_1^1 v_1^2$ *est satisfaite, l'arête clause* $v_2^1 v_2^2$ *est insatisfaite et l'arête clause* $v_3^1 v_3^2$ *n'est ni satisfaite, ni insatisfaite. L'assignation correspondante à* X *est* $x_1 =$ VRAI *et* $x_2 =$ FAUX*. À droite, une solution voisine* X' *de* X *est dessinée avec les sommets rouges.* X' *a un score de poids de cinq.* X' *est obtenue en faisant un glissement de voisinage sur l'arête* $u_1 \overline{u}_1$*.*

Lemme 13

COUPE SEMI-BRUTALE/Glissement de voisinage appartient à PLS.

Preuve.

Montrons qu'il existe trois fonctions calculables en temps polynomial A_L, B_L et C_L telles que définies dans la définition 19. On peut créer une solution normalisée initiale en coupant arbitrairement exactement une extrémité pour chaque arête ambiguë, ce qui permet de créer la fonction A_L. Soit X une solution de COUPE SEMI-BRUTALE. Le score de poids de X se calcule en temps linéaire, en parcourant les arêtes du graphe de solution et en faisant la somme des poids de celles incidentes à un sommet de X. On a donc bien une fonction en temps polynomial B_L. Enfin, une opération de glissement de voisinage peut se faire en temps linéaire, car il suffit de changer un nombre de coupes inférieur à $2\Delta(G_s^*)$. Comme il y a exactement autant de glissements de voisinage possibles que d'arêtes de couplage, on peut donc construire le voisinage de X en temps polynomial. La fonction B_L est en temps polynomial, on peut donc trouver la solution voisine de S avec le score de poids minimum dans ce voisinage. Ainsi, on a bien une fonction en temps polynomial C_L. □

Montrons maintenant que ce problème de recherche local est PLS-complet en faisant une PLS-réduction à partir du problème MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ en considérant la fonction de voisinage Flip qui est définie dans le premier chapitre par la définition 18. Pour cela, nous considérons la construction suivante.

Construction 6

À partir de (φ, ω') une instance du problème MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ, construire le graphe de solution (G_s^*, M^*, ω) suivant.

— Pour chaque variable x_i, construire une arête de couplage $u_i\overline{u}_i$.

— Pour chaque clause C_ℓ, construire une arête de couplage $v_\ell^1 v_\ell^2$.

— Pour chaque clause C_ℓ, soit x_k la $t^{ième}$ variable de la clause. Si x_k apparaît positivement dans C_ℓ, créer l'arête $v_\ell^t u_k$ avec $\omega(v_\ell^t u_k) = \omega'(C_\ell)$. Sinon, créer l'arête $v_\ell^t \overline{u}_k$ avec $\omega(v_\ell^t \overline{u}_k) = \omega'(C_\ell)$.

— Finalement, pour chaque variable x_i, construire une arête de couplage $s_1^i s_2^i$. Si $w(u_i) < w(\overline{u}_i)$, créer l'arête $s_1^i u_1$ avec $\omega(s_1^i u_1) = \omega(\overline{u}_i) - \omega(u_i)$. Si $\omega(u_i) > \omega(\overline{u}_i)$, créer l'arête $s_1^i \overline{u}_1$ avec $\omega(s_1^i \overline{u}_1) = \omega(u_i) - \omega(\overline{u}_i)$.

Dans cette construction, une arête de couplage $u_i\overline{u}_i$ est appelée *arête variable* et une arête de couplage $v_\ell^1 v_\ell^2$ est appelée *arête clause*. Un exemple de graphe issu de la construction 6 est donné par la figure 4.6. Pour passer d'une solution normalisée de la construction à une assignation des variables dans la formule initiale, on utilise la définition suivante.

Définition 31

Soit φ une instance du problème MAXIMUM 2-SATISFAISABILITÉ et soit (G_s^*, M^*, ω) son graphe de solution produit par la construction 6. Soit X une solution de (G_s^*, M^*, ω) contenant exactement une coupe par arête variable. Une assignation des variables β pour φ *correspond* à X si, pour toutes les arêtes de couplage $u_i\overline{u}_i$, on a $u_i \in X \Rightarrow \beta(x_i) = \text{VRAI}$ et $\overline{u}_i \in X \Rightarrow \beta(x_i) = \text{FAUX}$.

Notons que pour toute solution normalisée, il existe une assignation correspondante. Notons également que comme la définition précédente ne prend en compte que les coupes dans les arêtes variables (et pas les coupes dans les arêtes clauses), une assignation des variables peut correspondre à plusieurs solutions normalisées. Soit X une solution normalisée dans un graphe produit par la construction. Pour garder un vocabulaire similaire à celui des problèmes de satisfaisabilité, nous disons qu'une arête clause $v_\ell^1 v_\ell^2$ est *satisfaite* si la coupe contenue dans $v_\ell^1 v_\ell^2$ est adjacente à une coupe dans une arête variable. Si aucune des deux extrémités de l'arête n'est incidente à une coupe, on dit que $v_\ell^1 v_\ell^2$ est *insatisfaite*. Ces deux termes sont choisis ainsi car si une arête clause $v_\ell^1 v_\ell^2$ est satisfaite, alors la clause C_ℓ est satisfaite dans l'assignation correspondante. De même, si une arête clause $v_\ell^1 v_\ell^2$ est insatisfaite, alors la clause C_ℓ est insatisfaite dans l'assignation correspondante.

Il est possible qu'une arête clause ne soit ni satisfaite, ni insatisfaite, comme montré dans la figure 4.6. Après une opération de glissement de voisinage d'une arête variable $u_i\overline{u}_i$, toutes les arêtes clauses incidente à $u_i\overline{u}_i$ sont soit satisfaites, soit insatisfaites. Ainsi, pour remédier au fait qu'une arête clause ne soit ni satisfaite, ni insatisfaite, on peut effectuer un glissement de voisinage sur une des arêtes variables incidente à cette arête clause (et éventuellement une deuxième si l'on souhaite garder la même assignation correspondante).

Lemme 14

Soit (φ, ω') une instance du problème MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ et soit (G_s^*, M^*, ω) son graphe de solution produit par la construction 6. Soient X une solution normalisée pour (G_s^*, M^*, ω) et β l'assignation correspondante à X. On a les deux propriétés suivantes :

(a) si X est un minimum local pour l'opération de glissement de voisinage, alors toutes les arêtes clauses sont soit satisfaites, soit insatisfaites dans X ;

(b) si toutes les arêtes clauses sont satisfaites ou insatisfaites dans X, alors $score(X) = 2\omega'(\varphi) - \omega'(\beta)$.

Preuve.

(a) Supposons qu'il existe une arête clause $v_\ell^1 v_\ell^2$ qui n'est ni satisfaite, ni insatisfaite et supposons que le sommet v_ℓ^1 est coupé. Notons e_1 et e_2 les arêtes non couplantes et incidentes à v_ℓ^1 et v_ℓ^2, respectivement. Il existe une coupe sur le sommet adjacent à v_ℓ^2 dans une arête variable et donc les deux arêtes e_1 et e_2 sont enlevées par X. En effectuant une opération de glissement de voisinage sur l'arête $v_\ell^1 v_\ell^2$, le sommet v_ℓ^1 est remplacé par v_ℓ^2 dans X et donc l'arête e_1 n'est plus enlevée dans cette solution. Ainsi, il existe une solution voisine possédant un meilleur score de poids et donc X n'est pas un minimum local.

(b) Soit $u_i\overline{u}_i$ une arête variable. Par construction, on a $\omega(u_i) = \omega(\overline{u}_i) = \sum_{C_\ell \in \psi_i} \omega'(C_\ell)$. Ainsi, la somme de toutes les arêtes enlevées par une coupe dans une arête variable est égale à $\sum_i \sum_{C_\ell \in \psi_i} \omega'(C_\ell) = \omega'(\varphi)$. Soit $v_\ell^1 v_\ell^2$ une arête clause. Si $v_\ell^1 v_\ell^2$ est satisfaite, alors sa coupe n'augmente pas le score de poids de la solution car l'arête n'appartenant pas au couplage incidente à cette coupe est déjà incidente à une coupe dans une arête variable. Si au contraire, $v_\ell^1 v_\ell^2$ est insatisfaite, alors sa coupe enlève une arête non couplante supplémentaire e et comme $\omega(e) = \omega'(C_\ell)$ par construction, cela augmente le score de poids de X de $\omega'(C_\ell)$. Comme la somme des poids des clauses insatisfaites dans β est égale à $\omega'(\varphi) - \omega'(\beta)$, on obtient bien un score de poids pour X égal à $2\omega(\varphi) - \omega'(\beta)$.

□

Le lemme précédent nous permet de créer une relation entre le score de poids d'une solution normalisée X où toutes les arêtes clauses sont soit satisfaites, soit insatisfaites et le poids de son assignation correspondante. On peut ainsi établir de façon presque immédiate que si X est un minimum local, alors son assignation correspondante est un maximum local.

Théorème 17

COUPE SEMI-BRUTALE/Glissement de voisinage est PLS-complet pour le score de poids.

Preuve.

Le lemme 13 nous montre que COUPE SEMI-BRUTALE/Glissement de voisinage est dans PLS. Pour montrer que ce problème de recherche locale est PLS-complet, nous construisons une PLS-réduction à partir du problème de recherche locale MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ/Flip. Soit (φ, ω') une instance de MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ et soit (G_s^*, M^*, ω) son graphe de solution produit par la construction 6. Ce graphe de solution peut être calculé en temps polynomial. Soit X une solution normalisée dans (G_s^*, M^*, ω) et β l'assignation correspondante à X. Montrons que si X est un minimum local pour le score de poids alors β est un maximum local.
Par le lemme 14(b), on a $score(X) = 2\omega'(\varphi) - \omega'(\beta)$. Supposons par l'absurde que β n'est pas un maximum local, cela signifie qu'il existe une variable x_i in β telle que changer sa valeur produit une assignation β' telle que $\omega'(\beta') > \omega(\beta)$. Soit X' l'ensemble de coupes produit à partir de X en faisant une opération de glissement de voisinage sur l'arête variable $u_i\overline{u}_i$. L'assignation correspondante à X' est β'. Par le lemme 14(a), toutes les arêtes clauses sont soit satisfaites, soit insatisfaites par X. Comme après une opération de glissement de voisinage, les arêtes clauses modifiées sont nécessairement soit satisfaites, soit insatisfaites, c'est le cas de toutes les arêtes clauses après une telle opération. Ainsi par le lemme 14(b), $score(X') = \omega'(\varphi) - \omega'(\beta') < score(X)$, ce qui contredit le fait que X est un minimum local.
Ainsi, on a montré que la construction 6 constitue une PLS-réduction et donc COUPE SEMI-BRUTALE/Glissement de voisinage est PLS-complet pour le score de poids. □

4.2.1.3 Complexité paramétrée

On peut utiliser la construction utilisée dans la preuve précédente afin de montrer que si l'on paramétrise COUPE SEMI-BRUTALE par le poids de la solution, alors ce problème est $W[1]$-difficile pour le score de poids. On regardera toutefois une version un peu modifiée de la version de décision. Pour un graphe de solution (G_s^*, M^*, ω), on dénote par $\omega(G_s^*)$ la somme des poids des arêtes non couplantes du graphe de solution, c'est-à-dire $\omega(G_s^*) = \sum_{e \in E(G_s^*) \setminus M^*} \omega(e)$. Le problème paramétré étudié est le suivant.

COUPE SEMI-BRUTALE k-POIDS PARAMÉTRÉE

Entrée : un graphe de solution (G_s^*, M^*, ω) et un paramètre k.
Sortie : un ensemble de coupes X permettant de rendre propre toutes les arêtes de M^* et telle que le score de poids X est au plus égal à $\omega(GS) - k$.

Comme MAXIMUM 2-SATISFAISABILITÉ PONDÉRÉ est $W[1]$-complet, la propriété lemme 14(b) implique directement le corollaire suivant.

Corollaire 6

COUPE SEMI-BRUTALE k-POIDS PARAMÉTRÉE est $W[1]$-difficile.

Ici la formulation du problème paramétré est particulière car le score de poids recherché dépend du poids total de l'instance. Il pourrait être intéressant par la suite d'étudier une approche où cette valeur est complètement indépendante de l'instance.

4.2.2 Graphes denses

Intéressons nous à présent à montrer les difficultés de calculs pour les classes de graphes denses. Tout d'abord, nous pouvons étendre la preuve du théorème 16 pour le score de poids à tous les supergraphes des graphes planaires, bipartis et sous-cubiques en utilisant l'argument suivant. Si on ajoute des arêtes de poids nul à un graphe de solution, alors cela ne change pas le score de poids d'une solution optimale. Ainsi, si l'on trouve une solution optimale dans un tel supergraphe, alors cela implique que cette solution est également optimale dans le graphe de départ. Si on ne peut pas trouver une solution optimale en temps polynomial dans le graphe de départ, alors c'est également le cas dans le supergraphe. Cette observation mène au résultat suivant.

Corollaire 7

Soit $\mathcal{G}$ la classe de graphes telle que pour tout graphe planaire, biparti et sous-cubique G, $\mathcal{G}$ contient les supergraphes de G. COUPE SEMI-BRUTALE est NP-difficile pour le score de poids dans $\mathcal{G}$.

Pour le score de coupe, nous avons vu dans la partie 4.1.2 que COUPE SEMI-BRUTALE pouvait être résolu en temps polynomial dans beaucoup de classes de graphes denses. Nous montrons ici un exemple d'une classe de graphes denses, la classe des graphes scindés, pour laquelle COUPE SEMI-BRUTALE est tout de même NP-difficile. Pour cela, nous effectuons une réduction à partir de COUVERTURE PAR SOMMETS en utilisant la construction suivante.

Construction 7

À partir d'un graphe simple connexe G, nous produisons le graphe de solution (G_s^*, M^*, ω) suivant :

- pour chaque sommet v de G, construire une arête de couplage v_1v_2 ;
- pour chaque arête uv de G, créer deux arêtes v_1u_2 et u_1v_2 ;
- pour chaque paire de sommets (u, v), créer une arête u_1v_1.

Le sous-graphe induit par l'ensemble de sommets $\{v_1 \mid v \in V(G)\}$ est une clique et

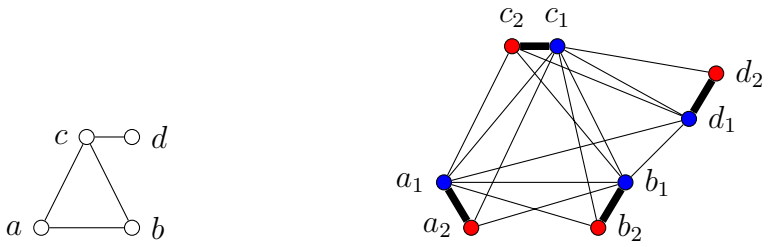


FIGURE 4.7 – *La construction 7 produit le graphe de solution de droite à partir du graphe à gauche. Les arêtes de couplage sont dessinées en gras, les sommets rouges forment un stable et les sommets bleus forment une clique.*

le sous-graphe induit par l'ensemble de sommets $\{v_2 \mid v \in V(G)\}$ est un stable. Ainsi, le graphe de solution produit par la construction est bien un graphe scindé. Notons également que toutes les arêtes de couplage de ce graphe sont ambiguës. Un exemple de graphe produit par la construction est représenté par la figure 4.7. Montrons que cette construction est une réduction polynomiale.

Théorème 18

COUPE SEMI-BRUTALE est NP-difficile dans les graphes scindés pour le score de coupe.

Preuve.

Soit G un graphe simple connexe et (G_s^*, M^*, ω) son graphe produit par la construction 7. Le graphe de solution peut être produit en temps polynomial. Montrons que G admet une couverture par sommets de taille k si et seulement si k coupes peuvent rendre propres toutes les arêtes de couplage de (G_s^*, M^*, ω).

- $\Rightarrow$ Soit V' une couverture par sommets de G de taille k. Pour chaque sommet $v \in V'$, coupons le sommet v_1 dans (G_s^*, M^*, ω). Supposons qu'il reste une arête de couplage v_1v_2 qui n'est pas propre. Cela signifie qu'il existe une arête non couplante v_2u_1 qui n'a pas été enlevée par une coupe. Aucun des deux sommets u et v de G n'appartient à V' et donc l'arête uv n'est pas incidente à un sommet de V', contredisant le fait que V' est une couverture. Ainsi, une telle arête v_1v_2 ne peut pas exister et donc on a construit une solution dans (G_s^*, M^*, ω) contenant k coupes.
- $\Leftarrow$ Soit X une solution de COUPE SEMI-BRUTALE dans (G_s^*, M^*, ω) contenant k coupes. Pour chaque $v \in V(G)$, s'il existe un sommet $v_2 \in X$, alors on modifie la solution en remplaçant v_2 par v_1 (*i.e.* $X := (X \setminus \{v_2\}) \cup \{v_1\}$). Comme le voisinage de v_2 est inclus dans le voisinage de v_1, X reste une solution et possède le même score de coupe. On peut donc supposer que $v_2 \notin X$ pour tout sommet $v \in V(G)$. Pour chaque sommet $v_1 \in X$, ajouter le sommet v dans la couverture par sommets de G. S'il existe une arête $uv \in E(G)$ qui n'est pas couverte, alors cela signifie que l'ensemble $\{u_1, u_2, v_1, v_2\} \cap X = \varnothing$. Comme les sommets u_1, u_2, v_1 et v_2 forment un cycle, les arêtes de couplage u_1u_2 et v_1v_2 ne sont pas propres. Ainsi une telle arête uv ne peut pas exister et donc on a construit une couverture de G de taille k.

Ainsi on a montré que la construction 7 constitue une réduction polynomiale et donc COUPE SEMI-BRUTALE est NP-difficile dans les graphes scindés pour le score de coupe. □

4.3 Non-approximation

On a montré dans la section précédente qu'il était difficile de calculer des solutions optimales dans un certain nombre de classes de graphes. On va s'intéresser dans cette section à la difficulté de calculer une bonne solution approchée en temps polynomial. Nous allons à la fois montrer l'appartenance à la classe de complexité APX-complet et donner des valeurs de bornes inférieures d'approximation pour COUPE SEMI-BRUTALE, dans l'hypothèse où P $\neq$ NP (et éventuellement dans le cas où l'hypothèse UGC est vérifiée). Pour cela, nous allons la plupart du temps réutiliser les constructions qui ont été définies dans la section 4.2 pour créer des réductions préservant les facteurs d'approximation.

4.3.1 Réduction à partir de MAXIMUM 3-SATISFAISABILITÉ

Dans un premier temps, montrons que la construction 5 peut être modifiée pour créer une L-réduction à partir du problème MAXIMUM 3-SATISFAISABILITÉ(4), dans lequel le nombre d'occurrences pour chaque variable est borné par quatre, pour le score de coupe. L'idée pour cela est d'obliger chaque clause insatisfaite à augmenter de exactement un le nombre de coupes nécessaires dans la construction. Toutefois, il faut apporter au préalable une légère modification à celle-ci. En effet, s'il est tout à fait possible d'utiliser une coupe supplémentaire pour chaque clause insatisfaite, l'inverse, n'est pas vrai : si une variable x_i apparaît deux fois positivement et deux fois négativement, cinq coupes dans c_i peuvent dans certains cas être suffisantes pour enlever toutes les arêtes reliant le cycle c_i aux cycles q_ℓ. Ainsi, en utilisant une seule coupe supplémentaire, on a pu simuler la « satisfaction » de deux clauses supplémentaires de la formule (voir figure 4.8 pour un exemple). La modification apportée consiste juste à intervertir l'ordre de certaines clauses dans la liste ψ_i pour certaines variables x_i. La construction utilisée est alors la suivante.

Construction 8

Soit φ une instance de MAXIMUM 3-SATISFAISABILITÉ(4). Avant d'appliquer la construction 5 à φ, nous appliquons les changements suivants : pour toute variable x_i apparaissant deux fois positivement et deux fois négativement dans φ, on ordonne la liste ψ_i en alternant les clauses dans lesquelles x_i apparaît positivement avec celles où x_i apparaît négativement. Toutes les autres variables demeurent inchangées.

Le graphe de solution résultant (G_s^*, M^*, ω) reste biparti et sous-cubique, mais n'est

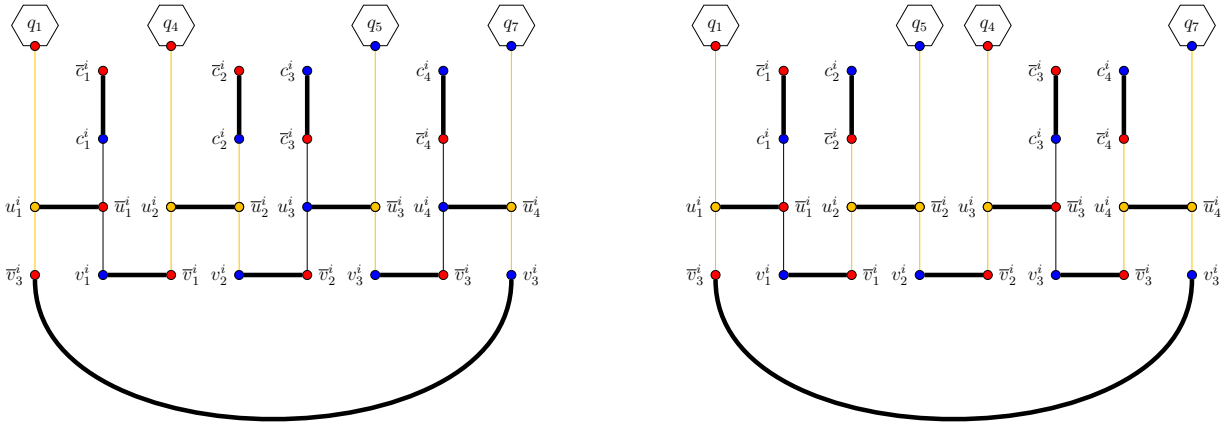


FIGURE 4.8 – *Exemple de cycle c_i produit par la construction 5, à droite, et par la construction 8, à gauche, pour une variable x_i apparaissant deux fois positivement et deux fois négativement. Les arêtes de couplage sont dessinées en gras, les sommets jaunes sont coupés et les arêtes jaunes sont les arêtes non couplantes retirées par les coupes. On peut observer qu'il est nécessaire d'utiliser six coupes pour supprimer toutes les arêtes reliant c_i aux cycles c_ℓ dans la construction 8 tandis que seules cinq coupes sont nécessaires dans la construction 5.*

plus forcément planaire. Notons que dans le cas du score de coupe, nous ne prenons pas en considération le poids des arêtes et donc tous les gadgets clauses sont symétriques. Ainsi les propriétés (a) et (c) du lemme 12 restent vraies. Nous ajoutons la propriété suivante.

Lemme 15

Soit φ une instance de MAXIMUM 3-SATISFAISABILITÉ(4) et (G_s^*, M^*, ω) son graphe de solution résultant de la construction 8. Soit $X \subseteq V(G_s^*)$, un ensemble de coupes minimum rendant propres toutes les arêtes de couplage de (G_s^*, M^*, ω). Soit x_i une variable de φ. Il existe un ensemble de coupes X' de même taille qui rend également propres toutes les arêtes de (G_s^*, M^*, ω) et tel que $X' \cap V(c_i) = \{u_j^i \mid j \leq |\psi_i|\}$ ou $X' \cap V(c_i) = \{\overline{u}_j^i \mid j \leq |\psi_i|\}$

Preuve.

Rappelons que comme l'ensemble X rend propre toutes les arêtes dans M^*, par le lemme 12(a), nous avons $score(X \cap V(c_i)) \geq |\psi_i|$. Dans un premier temps, supposons, par symétrie, que la variable x_i apparaît majoritairement positivement dans φ. Si x_i apparaît quatre fois positivement, alors remplacer les sommets de l'ensemble $X \cap V(c_i)$ par l'ensemble $\{u_j^i \mid j \leq |\psi_i|\}$ dans X mène à une solution X', comme souhaité. Supposons que x_i apparaît trois fois positivement. Soit C_ℓ la clause dans laquelle x_i apparaît négativement et dénotons par $b_{j'}^\ell$ le sommet adjacent à c_i dans q_ℓ. Si $score(X \cap V(c_i)) > |\psi_i|$, alors remplacer les sommets de l'ensemble $X \cap V(c_i)$ par l'ensemble $\{u_j^i \mid j \leq\} \cup \{b_{j'}^\ell\}$ dans X mène à une solution X' comme souhaité. Finalement, si $score(X \cap V(c_i)) = |\psi_i|$, alors X possède déjà la propriété demandée, sinon au moins une arête de couplage de c_i ne serait pas propre.

Supposons à présent que x_i apparaît deux fois positivement et deux fois négativement dans φ. Si un sommet v^i_j (resp. $\overline{v}^i_j$) est coupé, on peut le remplacer dans X par le sommet $\overline{u}^i_j$ (resp. $u^i_{j+1 \mod 4}$). Ainsi, on peut supposer que seuls les sommets u^i_j ou $\overline{u}^i_j$ appartiennent à X. Supposons, sans perte de généralité, que les sommets u^i_1 et u^i_3 sont adjacents à des gadgets clauses et que ces deux sommets appartiennent à X. Par construction, le sommet $\overline{u}^i_2$ est adjacent à un gadget clause q_ℓ, dénotons par $b^\ell_{j'}$ le sommet voisin de $\overline{u}^i_2$ dans q_ℓ . Si $\overline{u}^i_2$ est coupé, alors cela veut dire qu'il existe un sommet coupé parmi $\{\overline{u}^i_1, u^i_2\}$ pour rendre propre l'arête de couplage $v^i_1\overline{v}^i_1$. Comme la coupe du sommet $\overline{u}^i_2$ n'est utile que pour enlever l'arête $\overline{u}^i_2 b^\ell_{j'}$, on peut remplacer les sommets coupés dans $X \cap \{\overline{u}^i_1, u^i_2, \overline{u}^i_2\}$ par les sommets $\{u^i_2, b^\ell_{j'}\}$, ce qui n'augmente pas la cardinalité de X. On peut effectuer un raisonnement similaire si $\overline{u}^i_4$ est coupé et remplacer les sommets dans $X \cap \{\overline{u}^i_3, u^i_4, \overline{u}^i_4\}$, par les sommets $\{u^i_4, b^\ell_{j''}\}$ où $b^\ell_{j''}$ est le voisin de $\overline{u}^i_4$ dans le gadget clause $q_{\ell'}$ adjacent à $\overline{u}^i_4$. Ainsi, on obtient bien une solution X' avec $score(X') \leq score(X)$ et telle que $X' \cap V(c_i) = \{u^i_j \mid j \leq |\psi_i|\}$. Par symétrie, si les deux sommets $\overline{u}^i_2$ et $\overline{u}^i_4$ sont coupés, on peut effectuer une transformation analogue pour construire une solution X' avec $score(X') \leq score(X)$ et telle que $X' \cap V(c_i) = \{\overline{u}^i_j \mid j \leq |\psi_i|\}$. Notons que si les quatre sommets $u^i_1, \overline{u}^i_2, u^i_3$ et $\overline{u}^i_4$ sont coupés, alors il est possible de choisir quelle transformation effectuer (*i.e.* soit couper uniquement des sommets de V_1, soit uniquement couper des sommets de V_2). □

Nous allons nous servir du résultat de [13] montrant qu'il est difficile d'approximer MAXIMUM 3-SATISFAISABILITÉ(4) avec un facteur meilleur que $\epsilon_4 \leq 1.00052$ afin de trouver un résultat d'inapproximation pour COUPE SEMI-BRUTALE pour le score de coupe.

Théorème 19

Il n'est pas possible d'approximer COUPE SEMI-BRUTALE en temps polynomial avec un facteur d'approximation meilleur que $1 + \frac{7(\epsilon_4 - 1)}{41 \cdot \epsilon_4}$ pour le score de coupe, même dans les graphes bipartis et sous-cubiques (sauf si P = NP).

Preuve.

Tout d'abord, notons que dans une solution optimale de MAXIMUM 3-SATISFAISABILITÉ(4), au moins $7/8$ des clauses de la formule sont satisfaites [40] ce qui se traduit par l'équation suivante :

$$OPT(\varphi) \geq 7m/8. \tag{4.1}$$

Pour montrer la présence d'un L-réduction, nous définissons f comme l'application associant toute instance φ de MAXIMUM 3-SATISFAISABILITÉ(4) à son graphe de solution $I = (G^*_s, M^*, \omega)$ produit par la construction 8. Soit X une solution de COUPE SEMI-BRUTALE dans I qui possède les propriétés lemme 12(a), lemme 12(c) et lemme 15. On définit g comme la fonction transformant une solution X en une assignation pour φ, de la même façon que dans la preuve du théorème 16 : la valeur VRAI est assignée à la variable x_i si tous les sommets $\{u^i_j \mid j \leq |\psi_i|\}$ sont coupés, sinon la valeur FAUX est assignée à x_i. $5m$ coupes

sont nécessaires pour rendre propres toutes les arêtes de couplage et, par le lemme 15, chaque coupe en plus va apparaître dans un gadget de clause. Ainsi, nous pouvons trouver une solution dans I contenant une coupe supplémentaire pour chaque clause non satisfaite. Cela mène à l'équation suivante :

$$OPT(I) \leq 5m + {}^{m}/_{8} \overset{(4.1)}{\leq} {}^{41}/_{7} \cdot OPT(\varphi). \tag{4.2}$$

Comme, le score de X augmente de un pour chaque clause insatisfaite de $g(X)$, la somme des valeurs de $score(X)$ et du nombre de clauses satisfaites par $g(X)$ est constante, on a donc :

$$6m = val(g(X)) + val(X) = OPT(I) + OPT(\varphi). \tag{4.3}$$

Ainsi, on a construit une L-réduction avec $\alpha_1 = {}^{41}/_{7}$ et $\alpha_2 = 1$. Si X a été calculée en temps polynomial, comme la fonction g est polynomiale, on a $\epsilon_4 \cdot val(g(X)) \leq OPT(\varphi)$. On peut donc conclure que

$$\begin{aligned} val(X) &\overset{(4.3)}{=} OPT(I) + OPT(\varphi) - val(g(X)) \\ &\geq OPT(I) + (1 - {}^{1}/_{\epsilon_4}) \cdot OPT(I) \\ &\overset{(4.2)}{\geq} (1 + \frac{7(\epsilon_4 - 1)}{41 \cdot \epsilon_4}) \cdot OPT(I). \end{aligned}$$

□

Notons que, en perdant la propriété de bipartition, on peut également utiliser la construction 8 pour montrer qu'il est difficile d'approximer COUPE SEMI-BRUTALE en temps polynomial pour le score de poids avec un facteur d'approximation meilleur que ${}^{(7(\epsilon_4-1))}/_{(65\cdot\epsilon_4)}) \approx 1.000056$ [83]. Toutefois, nous montrerons dans la section qui suit comment obtenir une meilleure borne inférieure d'approximation inférieure pour le score de poids en effectuant une réduction à partir de MAXIMUM 2-SATISFAISABILITÉ.

4.3.2 Réduction à partir de MAXIMUM 2-SATISFAISABILITÉ

Nous présentons à présent une L-réduction de MAXIMUM 2-SATISFAISABILITÉ vers COUPE SEMI-BRUTALE, pour les deux fonctions de score. Nous réutiliserons pour cela la construction 6 décrite dans la sous-sous-section 4.2.1.2. Afin de pouvoir construire une L-réduction, nous avons besoin que chaque arête clause soit satisfaite ou insatisfaite pour avoir une correspondance entre le score de poids d'une solution X et le nombre de clauses satisfaites dans son assignation correspondante. Pour cela, si une solution possède une arête clause qui n'est ni satisfaite ni insatisfaite, nous pouvons la transformer pour qu'elle respecte la propriété suivante.

Lemme 16

Soit φ une instance de MAXIMUM 2-SATISFAISABILITÉ et soit (G_s^*, M^*, ω) son graphe de solution produit par la construction 6, en considérant que toutes les clauses de φ sont de poids un. Soit X une solution normalisée de (G_s^*, M^*, ω) et soit β l'assignation des variables correspondant à X. Soit m' le nombre de clauses non satisfaites dans β. Il existe une solution X' dans (G_s^*, M^*, ω) telle que $score(X') = 2m + m' \leq score(X)$, pour le score de poids, et β correspond à X'.

Preuve.

Supposons qu'il existe une arête clause $v_j^1 v_j^2$ qui n'est ni satisfaite, ni insatisfaite et supposons que la coupe de $v_j^1 v_j^2$ est effectuée sur le sommet v_j^1. Cela signifie que le sommet voisin de v_j^2 dans une arête variable est coupé. Prenons l'ensemble de coupes $X' = X \cup \{v_j^1\} \setminus \{v_j^2\}$. Comme toutes les arêtes de couplage ambiguës possèdent une coupe, X' est une solution normalisée. L'arête non couplante incidente à v_j^2 est déjà enlevée dans X par une coupe dans une arête variable et l'arête non couplante incidente à v_j^1 n'est plus enlevée dans X', nous avons donc $score(X') \leq score(X)$, pour le score de poids. Comme nous n'avons pas modifié les coupes dans les arêtes variables, l'assignation β correspond également à X'. Ainsi, nous pouvons supposer que X' ne contient aucune arête clause qui n'est ni satisfaite, ni insatisfaite.
Soit $u_i \overline{u}_i$ une arête variable. Par construction, $\omega(u_i) = \omega(\overline{u}_i) = |\psi_i|$. Ainsi, la somme des poids des arêtes enlevées par les coupes des arêtes variables est constante et est égale à $\sum_{i \leq n} |\psi_i| = 2m$. Supposons que les arêtes n'appartenant pas au couplage incidentes aux coupes dans les arêtes variables sont enlevées du graphe de solution. Soit $v_j^1 v_j^2$ une arête clause. Si $v_j^1 v_j^2$ est satisfaite, alors le sommet coupé dans $v_j^1 v_j^2$ n'augmente pas le poids de la solution car la seule arête n'appartenant pas au couplage incidente à celle-ci est déjà enlevée par une coupe dans l'arête variable. Si $v_j^1 v_j^2$ est insatisfaite, alors le sommet coupé dans $v_j^1 v_j^2$ enlève une arête n'appartenant pas au couplage et augmente ainsi le poids de la solution de un. Comme la somme des poids des arêtes enlevées par les coupes des arêtes clauses insatisfaites correspond au nombre de clauses insatisfaites par β, on a $score(X') = 2m + m'$ pour le score de poids. □

Nous dénotons par ϵ_2 le meilleur facteur d'inapproximation connu pour MAXIMUM 2-SATISFAISABILITÉ. Sous l'hypothèse P $\neq$ NP, nous avons $\epsilon_2 = {}^{22}/_{21}$ [40] et sous l'hypothèse UGC, nous avons $\epsilon_2 \approx 1.06$ [50].

Théorème 20

Il n'est pas possible d'approximer COUPE SEMI-BRUTALE en temps polynomial avec un facteur d'approximation meilleur que $1 + \frac{\epsilon_2 - 1}{3 \cdot \epsilon_2}$ pour le score de poids (sauf si $P =$ NP ou UGC est fausse).

Preuve.

Tout d'abord, nous pouvons adapter l'argument utilisé dans [40] pour donner une borne inférieure sur le nombre minimum de clauses satisfaites pour MAXIMUM 2-SATISFAISABILITÉ. Pour toute instance φ, une assignation aléatoire des variables peut satisfaire chaque clause avec une probabilité égale à $3/4$ et ainsi, il n'est pas difficile de trouver une assignation qui satisfait au moins $3m/4$ des clauses. Cela mène à l'équation suivante :

$$OPT(\varphi) \geq 3m/4. \tag{4.4}$$

De façon similaire à la preuve donnée pour le théorème 19, nous montrons que la construction 6 constitue une L-réduction. Soit f l'application associant à toute instance φ de MAXIMUM 2-SATISFAISABILITÉ le graphe de solution produit par la construction 6 à partir de φ. Soit X une solution normalisée dans $f(\varphi)$ correspondant à la propriété du lemme 16. Soit g la fonction qui construit l'assignation des variables de φ correspondante à X. Par le lemme 16, nous avons :

$$OPT(f(\varphi)) \leq 2m + m/4 \overset{(4.4)}{\leq} 3 \cdot OPT(\varphi). \tag{4.5}$$

Comme le score de X augmente de un pour chaque clause insatisfaite de $g(X)$, la somme de $score(X)$ et du nombre de clauses satisfaites par $g(X)$ est constante, on a donc :

$$3m = val(g(X)) + val(X) = OPT(f(\varphi)) + OPT(\varphi). \tag{4.6}$$

Ainsi, on a construit une L-réduction avec $\alpha_1 = 3$ et $\alpha_2 = 1$. Si X a été calculée en temps polynomial, comme la fonction g est polynomiale, on a $\epsilon_2 \cdot val(g(X)) < OPT(\varphi)$, on peut donc conclure que

$$\begin{aligned} val(X) &\overset{(4.6)}{=} OPT(I) + OPT(\varphi) - val(g(X)) \\ &\geq OPT(I) + (1 - 1/\epsilon_2) \cdot OPT(I) \\ &\overset{(4.5)}{\geq} (1 + \frac{\epsilon_2 - 1}{3 \cdot \epsilon_4}) \cdot OPT(I) \end{aligned}$$

□

Si on restreint le nombre d'occurrences des variables à k dans une instance de MAXIMUM 2-SATISFAISABILITÉ, cela permet de borner le degré maximum du graphe de solution produit par la construction 6 par k (en supposant qu'il n'y ait pas de variables apparaissant exclusivement positivement ou exclusivement négativement). Berman et Karpinski [12] ont montré que MAXIMUM 2-SATISFAISABILITÉ(3) ne pouvait pas être approché en temps polynomial avec un facteur meilleur que $\epsilon_2' \leq 2012/2011$, sauf si P = NP. En réutilisant les arguments de la preuve précédente, on peut montrer le résultat suivant.

Théorème 21

Il n'est pas possible d'approximer COUPE SEMI-BRUTALE en temps polynomial avec un facteur d'approximation meilleur que $1 + \frac{\epsilon'_2 - 1}{3 \cdot \epsilon'_2}$ dans les graphes sous-cubiques pour le score de poids (sauf si P = NP).

De la même façon on peut utiliser le résultat d'inapproximabilité de MAXIMUM 2-SATISFAISABILITÉ(3) pour améliorer la borne inférieure d'approximation du théorème 19 pour les graphes sous-cubiques pour le score de coupe. Toutefois, nous ne pouvons pas utiliser une solution normalisée car cela ne permet pas d'optimiser le score de coupe. Pour construire une assignation des variables correspondante, comme donnée par la définition 31, nous introduisons le lemme suivant.

Lemme 17

Soit φ une instance de MAXIMUM 2-SATISFAISABILITÉ(2) et soit (G^*_s, M^*, ω) son graphe de solution produit par la construction 6. Soit X une solution de (G^*_s, M^*, ω). Il existe une solution X' avec $score(X') \leq score(X)$ pour le score de coupe et telle que chaque arête variable contient exactement une coupe.

Preuve.

Soit $u_i\overline{u}_i$ une arête variable. Supposons par symétrie que x_i apparaît deux fois positivement et une fois négativement. Supposons que $u_i\overline{u}_i$ ne contienne pas de coupe. Si le sommet voisin v^t_j de $\overline{u}_i$ est coupé, alors nous pouvons échanger $v^{t'}_{j'}$ avec $\overline{u}_i$ dans X. Sinon, les deux voisins v^t_j et $v^{t'}_{j'}$ de u_i sont coupés et nous pouvons prendre l'ensemble $X' = X \cup \{u_i\} \setminus \{v^t_j, v^{t'}_{j'}\}$ qui est une solution possédant un meilleur score de coupe. On peut donc supposer que toutes les arêtes variables possèdent au moins une coupe. Supposons à présent que $u_i\overline{u}_i$ possèdent deux coupes. La coupe en $\overline{u}_i$ ne permet d'enlever que l'arête $\overline{u}_i v^t_j$, car l'arête de couplage $s^i_1 s^i_2$ n'est pas ambiguë. L'ensemble $X' = X \cup \{v^t_j\} \setminus \{\overline{u}_i\}$ est une solution de même score de coupe. Ainsi, on peut supposer que chaque arête variable contient une unique coupe. □

Théorème 22

Il n'est pas possible d'approximer COUPE SEMI-BRUTALE en temps polynomial avec un facteur d'approximation meilleur que $1 + \frac{9(\epsilon'_2 - 1)}{11 \cdot \epsilon'_2}$ dans les graphes sous-cubiques pour le score de coupe (sauf si $P = \neq$ NP).

Preuve.

Montrons que la construction 6 constitue également une L-réduction pour le score de coupe. Soit f une application associant à toute instance φ de MAXIMUM 2-SATISFAISABILITÉ(3) son graphe de solution produit par la construction 6. Soit X une solution de $f(\varphi)$ respectant la propriété du lemme 17. Soit g la fonction qui construit l'assignation des variables de φ correspondante à X. Soit C_ℓ une

clause de φ. Si C_ℓ est satisfaite dans $g(X)$ par l'assignation de la variable x_i, nous pouvons supposer que l'arête clause $v_\ell^1 v_\ell^2$ ne contient pas de coupe car il existe déjà un sommet coupé dans l'arête variable $u_i\overline{u}_i$ qui rend propre l'arête clause $v_\ell^1 v_\ell^2$. Sinon, comme aucune arête incidente à une extrémité de l'arête $v_\ell^1 v_\ell^2$ n'est retirée par une coupe dans une arête variable, cela signifie que l'arête clause $v_\ell^1 v_\ell^2$ contient une coupe. Ainsi X contient une coupe pour arête variable et une coupe pour chaque arête clause $v_\ell^1 v_\ell^2$, où C_ℓ n'est pas satisfaite dans $g(X)$. Comme le nombre de variables dans φ est égal à $2m/3$, cela donne les deux équations suivantes :

$$OPT(f(\varphi)) \overset{(4.4)}{\leq} 2m/3 + m/4 \leq 11/9 \cdot OPT(\varphi), \tag{4.7}$$

et

$$5m/3 = val(g(X)) + val(X) = OPT(f(\varphi)) + OPT(\varphi). \tag{4.8}$$

On a donc construit une L-réduction avec $\alpha_1 = 11/9$ et $\alpha_2 = 1$ et l'on obtient

$$val(X) \geq \left(1 + \frac{9(\epsilon_2' - 1)}{11 \cdot \epsilon_2'}\right) \cdot OPT(I).$$

□

4.3.3 Réduction à partir de COUVERTURE PAR SOMMETS

Nous finissons cette section en montrant un résultat d'inapproximabilité pour le score de coupe dans les graphes scindés et dans certains graphes avec un degré maximum borné. Pour cela nous effectuons deux S-réduction à partir du problème COUVERTURE PAR SOMMETS. Nous dénotons par ρ le meilleur facteur d'inapproximation connu pour COUVERTURE PAR SOMMETS. Sous l'hypothèse $P \neq$ NP, nous avons $\rho = 1.3606$ [34] et sous l'hypothèse UGC, nous avons $\rho = 2 - \epsilon$, pour tout $\epsilon > 0$ [51]. Comme la preuve donnée pour le théorème 18 constitue une S-réduction, le résultat suivant est immédiat.

Théorème 23

Il n'est pas possible d'approximer COUPE SEMI-BRUTALE en temps polynomial avec un facteur d'approximation meilleur que ρ pour le score de coupe dans les graphes scindés (sauf si $P =$ NP ou que UGC est fausse).

Une autre construction à partir de COUVERTURE PAR SOMMETS permet de créer une S-réduction permettant de donner une borne inférieure d'approximation dans des graphes avec des degrés maximum bornés. Cette construction a été présentée originalement dans [88].

Construction 9

À partir d'un graphe G, nous produisons le graphe de solution (G_s^*, M^*, ω) suivant :

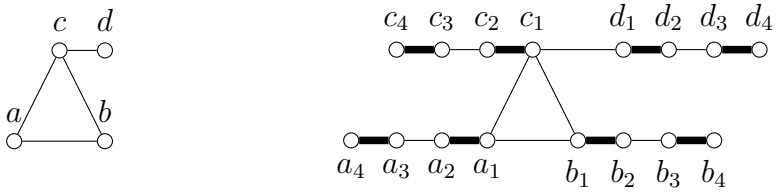


FIGURE 4.9 – *La construction 9 produit le graphe de solution de droite à partir du graphe à gauche. Les arêtes de couplage sont dessinées en gras.*

— pour chaque sommet $v \in V(G)$, construire le chemin (v_1, v_2, v_3, v_4) avec $v_1v_2, v_3v_4 \in M^*$;

— pour chaque arête $uv \in E(G)$, créer l'arête u_1v_1.

Dans le graphe de solution produit par la construction précédente, les arêtes ambiguës sont les arêtes v_2v_1, pour tout sommet v de l'instance originelle. Le degré maximum du graphe de solution produit est égal à $\Delta(G)+1$. Un exemple de graphe produit par la construction est donné par la figure 4.9.

Berman et Karpinski [12] ont montré que, dans l'hypothèse où P $\neq$ NP, pour les graphes de degré maximum trois, quatre, ou cinq, le problème COUVERTURE PAR SOMMETS ne pouvait pas être approximé avec un facteur meilleur que $^{145}/_{144}$, $^{79}/_{78}$ ou $^{74}/_{73}$, respectivement. Notons ρ_Δ le meilleur facteur d'inapproximation connu pour COUVERTURE PAR SOMMETS dans les graphes de degré borné Δ. On peut utiliser la construction 9 pour montrer le résultat d'inapproximation suivant.

Théorème 24 – [88]

Il n'est pas possible d'approximer COUPE SEMI-BRUTALE en temps polynomial avec un facteur d'approximation meilleur que ρ_Δ dans les graphes de solution de degré maximum Δ pour le score de coupe.

Preuve.

Soit G un graphe et soit (G_s^*, M^*, ω) son graphe de solution produit par la construction 9. Montrons qu'il existe une couverture par sommet de taille k dans G si et seulement si, on peut rendre propres toutes les arêtes de couplage de (G_s^*, M^*, ω) en utilisant k coupes.

$\Rightarrow$ Soit V' une couverture par sommet de taille k de G. Pour tout sommet $v \in V'$, coupons le sommet v_1. Toutes les arêtes n'appartenant pas au couplage incidentes aux sommets u_1 dans le graphe de solution sont coupées, et donc toutes les arêtes de couplage u_1u_2 sont propres. Ainsi, on a construit une solution pour COUPE SEMI-BRUTALE dans (G_s^*, M^*, ω) en utilisant k coupes.

$\Leftarrow$ Soit X une solution de COUPE SEMI-BRUTALE de taille k dans (G_s^*, M^*, ω). Prenons l'ensemble $V' = \{v \mid \{u_1, u_2\} \cap X \neq \varnothing\}$. On a $|V'| \leq |X|$. Si V' n'est pas une couverture par sommet, alors cela signifie qu'il existe une $uv \in$

$E(G)$ telle que $\{u, v\} \cap V' = \varnothing$. Alors, aucun des sommets parmi $\{u_1, v_1, v_2\}$ n'est coupé dans X. Comme u_3 ne peut pas être coupé car il n'appartient pas à une arête ambiguë, cela signifie que l'arête de couplage v_1v_2 n'est pas propre et donc que X n'est pas une solution. Ainsi, on a construit une couverture par sommets de taille k pour G.

La construction 9 est une réduction stricte et cela permet de transférer les résultats d'inapproximabilité de COUVERTURE PAR SOMMETS vers COUPE SEMI-BRUTALE pour le score de coupe.

□

4.4 Approximation

Dans la section précédente, nous avons présenté des résultats négatifs concernant l'approximation, c'est-à-dire des valeurs pour lesquelles COUPE SEMI-BRUTALE ne peut pas être approximé en temps polynomial. Nous présentons à présent dans cette section, des résultats positifs : c'est-à-dire des algorithmes polynomiaux permettant de calculer une solution avec un facteur d'approximation constant. Nous allons décrire deux algorithmes approchés : un pour chaque fonction de score. L'existence de ces algorithmes, combinée aux bornes d'inapproximation va permettre de conclure à l'appartenance de COUPE SEMI-BRUTALE à la classe APX-complet. Notons que l'algorithme présenté pour le score de poids a vocation à pouvoir être utilisé sur des instances réelles. Nous évaluerons ses performances sur ce type d'instance dans la section 4.6.

4.4.1 Score de coupe

Commençons par donner un algorithme d'approximation pour le score de coupe. Cet algorithme d'approximation s'inspire de l'algorithme d'approximation classique utilisé pour approximer le problème COUVERTURE PAR SOMMETS qui est décrit dans le premier chapitre par l'algorithme 3.

Cet algorithme d'approximation peut se décrire, de façon informelle, comme suit. Dans cet algorithme, on souhaite interdire la présence d'une arête non couverte par un sommet après l'exécution de l'algorithme. Pour cela, tant qu'il reste une arête non couverte, on ajoute les deux sommets auxquels cette arête est incidente à la couverture. Comme au moins un de ces deux sommets appartient à une couverture optimale, cela donne un facteur d'approximation de deux quand toutes les arêtes ont été couvertes. Pour COUPE SEMI-BRUTALE, on souhaite interdire les arêtes ambiguës : il reste des arêtes ambiguës dans le graphe de solution si et seulement si il existe un chemin de longueur trois tel que l'arête centrale appartienne au couplage. On élimine, l'arête ambiguë en coupant les quatre sommets de ce chemin. Comme au moins un de ces sommets est coupé dans une solution optimale, cela nous donne un facteur d'approximation de quatre.

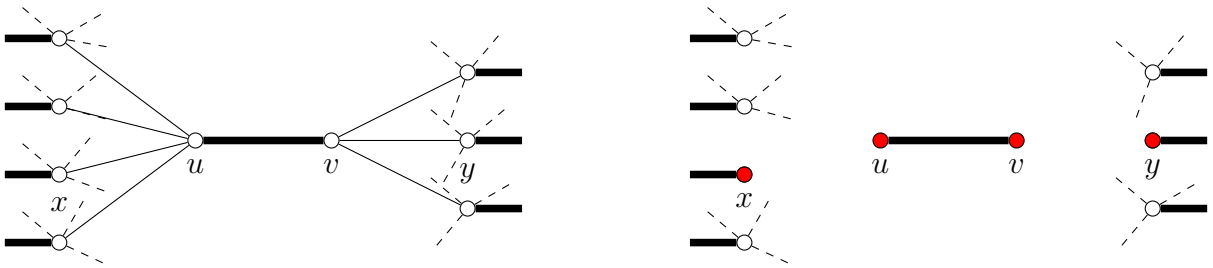


FIGURE 4.10 – *Exemple de chemin interdit (x, u, v, y). Les arêtes de couplage sont dessinées en gras et les arêtes en traits pleins sont les arêtes incidentes à u ou v. À droite, on a éliminé le chemin interdit en coupant les quatre sommets x, u, v et y.*

Dans ce qui suit, un chemin (x, u, v, y) est dit *interdit* si l'arête uv est une arête de couplage, voir la figure 4.10 pour un exemple.

Lemme 18

Soit un ensemble maximal de chemins interdits sommets-disjoints dans un graphe de solution (G_s^*, M^*, ω). Soit X un ensemble de coupes rendant propres toutes les arêtes de couplage dans (G_s^*, M^*, ω). Alors,

(a) couper toutes extrémités d'arêtes ambiguës dans $V(Q)$ rend propres toutes les arêtes de couplage de (G_s^*, M^*, ω);

(b) $X \cap p \neq \varnothing$ pour tout chemin p de Q.

Preuve.

(a) Supposons par contradiction, que après avoir coupé tous les sommets de $V(Q)$, il existe une arête de couplage uv qui n'est pas propre. Par définition, il existe deux arêtes non couplantes xu et vy qui n'ont pas été enlevées par une coupe. Sans perte de généralité, si x est une extrémité d'une arête ambiguë, alors x n'appartient pas à $V(Q)$ sinon l'arête xu (resp. yv) aurait été enlevée par la coupe du sommet x. Si x est une extrémité d'une arête non ambiguë, alors par la propriété 5, x ne peut être contenue dans un chemin interdit ne contenant pas u et donc x n'appartient pas à $V(Q)$. Ainsi, les deux sommets x et y n'appartiennent pas à $V(Q)$. Alors, le chemin (x, u, v, y) est interdit et cela contredit le fait que Q soit maximal.

(b) Soit $(x, u, v, y) \in Q$ un chemin interdit de (G_s^*, M^*, ω). Supposons par contradiction que $\{x, u, v, y\} \cap X = \varnothing$. Cela signifie que les deux arêtes xu et vy ne sont pas enlevées par une coupe dans X et donc l'arête de couplage uv n'est pas propre, ce qui contredit le fait que X est une solution.

□

En utilisant le résultat du lemme précédent, il est possible de créer une 4-approximation pour le score de coupe en temps linéaire.

Algorithme 15 : 4-approximation pour le score de coupe

Entrée : Un graphe de solution (G_s^*, M^*, ω).
Sortie : Un ensemble de coupes $X \subseteq V(G_s^*)$ rendant propres toutes les arêtes de couplage de (G_s^*, M^*, ω).

1 $Q \leftarrow$ ensemble maximal de chemins interdits sommets-disjoints;
2 **retourner** $V(Q) \cap \{u \mid u \texttt{ est une extrémité d'une arête ambiguë}\}$;

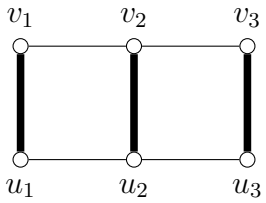


FIGURE 4.11 – *Le facteur d'approximation de l'algorithme 15 est serré. Les arêtes de couplage sont dessinées en gras. L'algorithme d'approximation donne une solution* $\{v_1, v_2, u_2, u_3\}$ *alors qu'une solution optimale est* $\{v_2\}$.

Théorème 25

L'algorithme 15 donne une solution en temps linéaire pour COUPE SEMI-BRUTALE avec un facteur d'approximation de quatre pour le score de coupe. Ce facteur d'approximation est serré.

Preuve.

Soit (G_s^*, M^*, ω) un graphe de solution. Un ensemble maximal Q de chemins interdits peut être calculé en temps linéaire en parcourant toutes les arêtes de couplage uv de (G_s^*, M^*, ω) : si u possède un voisin x n'appartenant pas à $V(Q)$ et si v possède un voisin y n'appartenant pas à $V(Q)$, alors le chemin (x, u, v, y) est interdit et on l'ajoute à Q. Considérons l'ensemble X' retourné par l'algorithme. Par le lemme 18(a), X' est une solution pour COUPE SEMI-BRUTALE. Pour montrer que cette solution est une 4-approximation, considérons une solution optimale X pour (G_s^*, M^*, ω). Par le lemme 18(b), chaque chemin interdit de Q contient au moins un sommet de X. Comme les chemins interdits de Q sont sommets-disjoints et contiennent quatre sommets, on peut conclure que $V(Q)$ contient au plus quatre fois plus de sommets que X. Le facteur d'approximation est serré, comme le montre la figure 4.11 □

Notons que pour obtenir un facteur d'approximation constant, nous avons coupé beaucoup de sommets dans la solution approchée. Cela fait que cet algorithme n'a pas vocation à être performant car il aura tendance à couper presque tous les sommets du graphe de solution. Cependant, ce résultat est intéressant, comme nous le verrons à la fin de cette section, pour montrer l'appartenance de COUPE SEMI-BRUTALE à la classe APX.

4.4.2 Score de poids

Présentons à présent un algorithme approché pour le score de poids. Soit (G_s^*, M^*, ω) un graphe de solution et X un ensemble de coupe. Pour un sommet u, on dénote par $\omega_X(u)$ la somme des poids des arêtes non couplantes incidentes à u n'ayant pas été enlevées par X. Formellement, soit $W_X(u) = \{uv \mid v \notin X\}$, on a $\omega_X(u) = \sum_{e \in W_X(u)} \omega(e)$.

Le principe de l'algorithme d'approximation pour le score de poids est le suivant : tant qu'il reste une arête de couplage ambiguë, on coupe l'extrémité d'une arête ambiguë u possédant la plus petite valeur $\omega_X(u)$. Cet algorithme est décrit formellement par l'algorithme 16.

Algorithme 16 : 2-approximation pour le score de poids.

Entrée : Un graphe de solution (G_s^*, M^*, ω).
Sortie : Un ensemble de coupes $X \subseteq V(G_s^*)$ rendant propres toutes les arêtes de couplage de (G_s^*, M^*, ω).

```
1 X ← ∅ ;
2 A ← liste des extrémités d'arêtes ambiguës de G*_s;
3 Tant que A ≠ ∅ faire
4 |   u ← argmin_{x∈A}(ω_X(x));
5 |   A ← A \ {u, M*(u)};
6 |   X ← X ∪ {u};
7 retourner X;
```

FIGURE 4.12 – *Le facteur d'approximation de algorithme 16 est serré. Les arêtes de couplage sont dessinées en gras, les arêtes dessinées en traits pleins sont enlevées par une solution optimale et les arêtes dessinées en pointillés sont enlevées par la solution retournée par l'algorithme.*

Théorème 26

L'algorithme 16 donne en temps $\mathcal{O}(|V(G_s^*)|+|E(G_s^*))\times log(|V(G_s^*)))$ une solution pour COUPE SEMI-BRUTALE avec un facteur d'approximation de deux pour le score de poids. Ce facteur d'approximation est serré.

Preuve.

Soit (G_s^*, M^*, ω) un graphe de solution. Tout d'abord, à chaque fois qu'une extrémité u est retirée de l'ensemble A, l'arête ambiguë à laquelle elle appartient contient soit une coupe en u, soit une coupe en $M^*(u)$ et est donc propre. Ainsi, l'algorithme se termine quand A est vide, cela signifie que toutes les arêtes ambiguës contiennent une coupe et donc que X est une solution. Soient E_{opt} l'ensemble des arêtes enlevées par une solution optimale et E_{app} l'ensemble des arêtes enlevées par la solution retournée par l'algorithme 16 pour (G_s^*, M^*, ω). Dénotons u_i le sommet ajouté à la solution à l'étape i de la boucle et par X_i la solution partielle à l'étape i. Soit E_i l'ensemble des arêtes retirées par la coupe en u_i, c'est-à-dire les arêtes n'appartenant pas au couplage qui sont incidentes à u et qui ne sont incidentes à aucun sommet de X_{i-1}. Si toutes les arêtes de E_i sont incidentes à un sommet de X_{opt}, alors on définit Q_i comme l'ensemble égal à E_i, sinon on définit Q_i comme l'ensemble des arêtes n'appartenant pas au couplage incidentes à $M^*(u_i)$. Dans les deux cas, les arêtes de Q_i sont enlevées par une coupe de X_{opt}. On a $\sum\limits_{e\in E_i}\omega(e) \leq \sum\limits_{e\in Q_i}\omega(e)$ et donc $\sum\limits_{e\in E_{app}}\omega(e) \leq \sum_i \sum\limits_{e\in Q_i}\omega(e)$. Pour toute arête uu' enlevée dans une solution optimale, uu' peut appartenir à l'ensemble Q_i si l'algorithme d'approximation coupe u ou $M^*(u)$ à l'étape i et uu' peut aussi appartenir à l'ensemble $Q_{i'}$ si l'algorithme d'approximation coupe u' ou $M^*(u')$ à l'étape i'. Ainsi chaque arête retirée par une solution optimale peut appartenir à au plus deux ensembles Q_i. De plus comme $\bigcup_i Q_i = E_{opt}$, on peut conclure que

$$\sum_i \sum_{e\in Q_i}\omega(e) \leq 2\sum_{e\in E_{opt}}\omega(e),$$

ce qui correspond au facteur d'approximation désiré. Ce facteur est serré, comme le montre la figure 4.12.
Concernant le temps d'exécution, la construction de la liste des arêtes ambiguës peut être faite en $\mathcal{O}(|M^*|)$, le tri de cette liste peut être effectué en $\mathcal{O}(|V(G_s^*)| \times log|V(G_s^*)|)$. Le maintien de cette liste triée lors des coupes peut se faire en $\mathcal{O}(|E(G_s^*)| \times log|V(G_s^*)|)$. On obtient bien une complexité en temps de $\mathcal{O}(|V(G_s^*)| + |E(G_s^*)) \times log(|V(G_s^*)))$.

□

4.4.3 Complétude dans la classe APX

Pour chaque fonction de score, on a montré qu'une solution optimale pouvait être approchée avec un facteur d'approximation constant. Comme les résultats donnés dans la section 4.3 montrent qu'on ne peut pas faire mieux qu'un ratio constant en temps polynomial, du moins avec l'hypothèse P $\neq$ NP, cela permet de conclure quant à la complétude du problème dans APX.

Corollaire 8

COUPE SEMI-BRUTALE est APX-complet pour les deux fonctions de score.

4.5 Méthodes exactes

Dans cette section, nous nous intéresserons à deux méthodes exactes pour résoudre COUPE SEMI-BRUTALE pour les deux fonctions de score. La première utilise une formulation linéaire en nombres entiers et la deuxième est un algorithme dynamique utilisant une décomposition arborescente du graphe de solution. Ces deux méthodes n'ont pas une complexité polynomiale mais étant donné que COUPE SEMI-BRUTALE est NP-difficile, il ne sera pas possible de faire tellement mieux. Toutefois, on peut espérer que sur des instances de petites ou moyennes tailles, elles puissent calculer une solution optimale en temps raisonnable. Nous aurons une première indication dans la section 4.6. Une autre utilité que l'on peut trouver aux méthodes exactes est qu'elles permettent de donner la valeur optimale pour l'évaluation des méthodes approchées, du moins sur des instances de taille pas trop importante.

4.5.1 Programmation linéaire en nombres entiers

Dans un premier temps, montrons comment on peut modéliser le problème COUPE SEMI-BRUTALE en utilisant une formulation linéaire en nombres entiers.

Variables. Soit (G_s^*, M^*, ω) un graphe de solution. Tout d'abord, nous allons décrire les variables utilisées dans la modélisation. Les variables utilisées sont des variables *binaires*, c'est-à-dire qu'elles peuvent prendre soit la valeur 1, soit la valeur 0 (elles peuvent être vues comme des variables booléennes où la valeur 1 correspond à la VRAI et la valeur 0 correspond à la FAUX). Pour chaque arête e_k n'appartenant pas au couplage, nous définissons une variable binaire x_k. La variable x_k prend la valeur 1 si et seulement si une de ses deux extrémités appartient à la solution, autrement dit, si e_k est enlevée du graphe de solution par la solution. Pour chaque sommet u_i, nous définissons deux variables binaires c_i et n_i. La variable c_1 prend la valeur 1 si et seulement si le sommet u_i appartient à la solution, et la variable n_i prend la valeur 1 si et seulement si tous les voisins de u_i, à l'exception de $M^*(u_i)$ sont dans la solution.

Contraintes. Définissons à présent les contraintes de la formulation.

(1) Pour toute arête ambiguë u_iu_j, on souhaite imposer que une des deux extrémités soit de degré un après l'application de la solution. Pour cela, on utilise la contrainte $n_i + n_j + c_i + c_j \geq 1$.

(2) Si une extrémité u_i appartient à une arête non ambiguë, alors il n'est pas possible de couper u_i. Dans ce cas, nous ajoutons la contrainte $u_i = 0$.

(3) Pour toute extrémité u_i d'arête ambiguë, on souhaite imposer que tous les voisins de u_i, à l'exception de $M^*(u_i)$ soient coupés si $n_i = 1$. Pour cela, on ajoute la contrainte $\sum_{u_j \in N(u_i)} c_j \geq n_i \cdot |N(u_i)|$.

(4) Pour chaque extrémité u_i d'une arête e_k non couplante, on souhaite que e_k soit enlevée du graphe si u_i appartient à la solution. Nous ajoutons la contrainte $x_k \geq c_i$.

Fonction objectif. Enfin, pour chaque fonction de score, nous définissons une fonction objectif à optimiser. Pour le score de coupe, on souhaite simplement minimiser le nombre de sommets dans la solution, autrement dit minimiser le nombre de variables c_i prenant la valeur 1. Ainsi, la fonction objectif pour le score de coupe est $min \sum_i c_i$. Pour le score de poids, on souhaite minimiser la somme des poids des arêtes pour lesquelles x_k prend la valeur 1. La fonction objectif pour le score de poids est donc $min \sum_k x_k \cdot \omega(e_k)$.

4.5.2 Programmation dynamique sur une décomposition arborescente

Une manière de résoudre COUPE SEMI-BRUTALE est d'utiliser une belle décomposition arborescente du graphe de solution. Soit (G_s^*, M^*, ω) un graphe de solution. Dans notre algorithme, pour tout sac B de la décomposition arborescente, on souhaite que pour tout sommet $u \in B$ on ait $M^*(u) \in B$.

Définition 32

Étant donné un graphe de solution (G_s^*, M^*, ω), une décomposition arborescente $(T, \mathcal{X})$ de G_s^* est appelée *M^*-préservante* si T est enraciné en un sac B_r telle que $B_r = \varnothing$ et chaque sac B_i de $\mathcal{X}$ fait partie d'un de ces quatre types.

- **Sac feuille.** B_i n'a pas d'enfant et $B_i = \varnothing$.
- **Sac joignant.** B_i a deux enfants B_j et B_k et $B_i = B_j = B_k$.
- **Sac uv-introduisant.** B_i a un enfant B_j, $uv \in M^*$ et $B_j = B_i \setminus \{u, v\}$.
- **Sac uv-oubliant.** B_i a un enfant B_j, $uv \in M^*$ et $B_i = B_j \setminus \{u, v\}$.

On dénote par tw^* le nombre maximum d'arêtes de couplage dans un sac.

Un exemple de décomposition arborescente M^*-préservante est donné par la fi-

gure 4.13. Cette définition est une adaptation d'une *bonne décomposition arborescente*, définie par Kloks [52]. La différence est qu'une bonne décomposition arborescente introduit ou oublie des sommets, alors qu'une décomposition arborescente M^*-préservante introduit ou oublie des arêtes de couplage. On peut construire une bonne décomposition arborescente à partir d'une décomposition arborescente en temps de calcul linéaire. Pour calculer une décomposition arborescente M^*-préservante, il suffit de contracter chaque arête de couplage en un unique sommet, puis de calculer une bonne décomposition arborescente et enfin de remplacer chaque sommet de cette décomposition en une arête de couplage. Enfin, notons que pour chaque arête de couplage uv, une décomposition arborescente M^*-préservante contient exactement un sac uv-oubliant. Notons également que le sac racine est un sac oubliant.

Soit (G_s^*, M^*, ω) un graphe de solution et $(T, \mathcal{X})$ une décomposition arborescente M^*-préservante de (G_s^*, M^*, ω). Par la suite, pour un sac $B_i \in \mathcal{X}$, on notera G_i le sous-graphe de G_s^* induit par les sommets introduits par B_i ou un descendant de B_i. Formellement, G_i est défini récursivement de la façon suivante : si B_i est un sac feuille, alors G_i est le graphe vide, sinon G_i est le sous-graphe de G_s^* induit par les sommets $B_i \cup \bigcup_{B_j \in enfants(B_i)} V(G_j)$.

4.5.2.1 Définitions préliminaires

Pour un ensemble de coupes X et un ensemble de sommets V', on introduit la notion de signature permettant de savoir quels sommets de V' sont de degré un après l'application de X dans un sous-graphe donné. Pour cela nous associons chaque sommet $u \in V'$ à un symbole "$\times$","N" ou "$\varnothing$" si, respectivement, u est coupé, tous les voisins de u sont coupés, ou si aucun des deux précédents cas n'est vrai. Formellement, la notion de signature est formulée par la définition suivante.

Définition 33

Soit (G_s^*, M^*, ω) un graphe de solution et H un sous-graphe de G_s^* Soit X un ensemble de coupes dans H et soit $V' \subseteq V(H)$ un ensemble de sommets. Une association $Y : V' \mapsto \{"\times","N","\varnothing"\}$ telle que :

(i) $Y(u) =$"$\times$"$\Leftrightarrow u \in X$;

(ii) $Y(u) =$"N"$\Rightarrow N_H(u) \subseteq X$;

est appelée *signature* de X dans V'. L'ensemble $\mathcal{T}(Y) = \{u \mid Y(u) =$"$\times$"$\}$ est la *trace* de Y.

Une solution X peut être associée à plusieurs signatures différentes, en fonction de l'ensemble de sommets V' considéré. De même, deux solutions différentes X et X' telles que $X \cap V' = X' \cap V'$ sont associées à une même signature lorsqu'on considère le sous-ensemble de sommets V'. Dans l'algorithme dynamique utilisé ici, nous allons parcourir les sacs d'une décomposition arborescente M^*-préservante en commençant par les sacs feuilles. L'idée est de générer toutes les solutions possibles à l'intérieur d'un sac B_i et de ne garder que la meilleure pour chaque signature dans B_i. De plus, pour réduire le nombre de solutions énumérées, nous ajouterons quelques restrictions

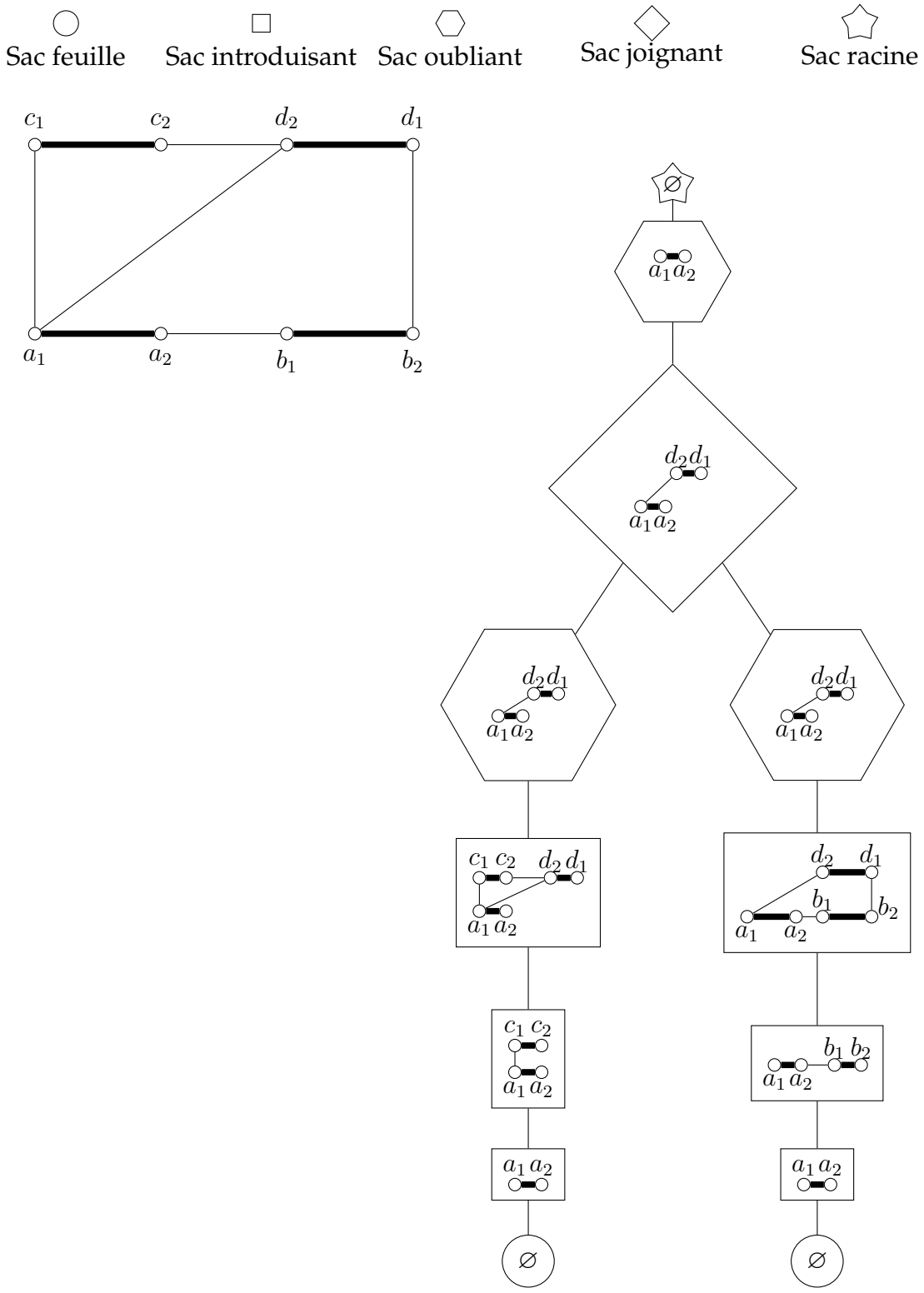


FIGURE 4.13 – *Graphe de solution et une décomposition arborescente M^*-préservante de celui-ci.*

sur les signatures. Pour un ensemble de sommets V', nous définissons un ensemble de signatures $\mathcal{Y}(V')$ de la façon suivante.

Définition 34

Soient (G_s^*, M^*, ω) un graphe de solution et $V' \subseteq V(G_s^*)$ un ensemble de sommets tel que pour tout sommet $u \in V'$ on a $M^*(u) \in V'$. L'ensemble $\mathcal{Y}(V')$ est l'ensemble de signatures Y telle que pour tout sommet $u \in V'$ les trois conditions suivantes sont vérifiées :

1. $uM^*(u)$ n'est pas ambiguë $\Leftrightarrow Y(u) = Y(M^*(u)) = \varnothing$;
2. si l'on considère le score de poids, alors :
   - $Y(u) \neq$"N",
   - $Y(u) =$"$\varnothing$"$\Leftrightarrow Y(M^*(u)) =$"$\times$";
3. si l'on considère le score de coupe et $uM^*(u)$ est ambiguë, alors :
   - $Y(u) =$"$\varnothing$"$\Rightarrow Y(M^*(u)) \neq$"$\varnothing$",
   - $Y(u) =$"N"$\Rightarrow Y(M^*(u)) =$"$\varnothing$".

La première condition interdit aux arêtes du graphe de solution non ambiguës d'avoir une extrémité coupée. La seconde condition permet d'obtenir une solution normalisée pour le score de poids en garantissant que toutes les arêtes ambiguës vont posséder une unique coupe. Ainsi, pour le score de poids, on a $|\mathcal{Y}(V')| = 2^{|M'|}$ pour un ensemble V' comprenant M' arêtes de couplage. Enfin la troisième condition permet que chaque arête ambiguë soit propre et élimine le cas où les deux voisinages des deux extrémités sont coupés qui n'est pas utile de retenir en mémoire. En effet, s'il est possible que ce cas survienne dans une solution optimale, alors cette solution sera également associée à une signature Y où $Y(u) =$"N" et $Y(M^*(u)) =$"$\varnothing$" (ou $Y(u) =$"$\varnothing$" et $Y(M^*(u)) =$"N"). Ainsi, pour le score de coupe, on a $|\mathcal{Y}(V')| = 5^{|M'|}$ pour un ensemble V' comprenant M' arêtes de couplage.

Pour construire les solutions dans l'algorithme dynamique, nous avons besoin d'associer plusieurs signatures, issues de sous-graphes différents. Pour cela, nous utiliserons l'opération suivante.

Définition 35

Soient $Y_i : V_i \mapsto \{$"$\times$","N","$\varnothing$"$\}$, pour $i \in \{1, 2\}$, deux signatures telles que $V_1 \cap V_2 = \varnothing$. L'union de Y_1 et Y_2 est l'application $Y_1 \cup Y_2 : V_1 \cup V_2 \mapsto \{$"$\times$","$N$","$\varnothing$"$\}$ telle que

$$(Y_1 \cup Y_2)(v) = \begin{cases} Y_1(v) & \text{si } v \in V_1 \\ Y_2(v) & \text{sinon.} \end{cases}$$

Comme nous ne souhaitons garder qu'une seule solution pour chaque signature, nous introduisons la notion suivante.

Définition 36

Soient $(T, \mathcal{X})$ une décomposition arborescente M^*-préservante d'un graphe de solution (G_s^*, M^*, ω) et B_i un sac de $\mathcal{T}$. Soient $X \subset V(G_i)$ un ensemble de coupes dans G_i et $Y \in \mathcal{Y}(B_i)$ une signature dans B_i. X est *éligible* pour (Y, B_i) si :

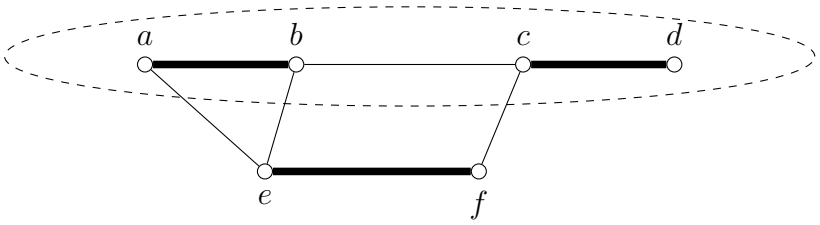


FIGURE 4.14 – *Un sous-graphe G_i avec $B_i = \{a, b, c, d\}$. Les arêtes de couplage sont dessinées en gras. Soient $Y_1 = \{(a,"\varnothing"), (b,"\varnothing"), (c,"\times"), (d,"\times")\}$, $Y_2 = \{(a,"\varnothing"), (b,"N"), (c,"\varnothing"), (d,"\varnothing")\}$ et $Y_3 = \{(a,"N"), (b,"\times"), (c,"\varnothing"), (d,"\varnothing")\}$. Aucun ensemble de coupes n'est éligible pour (Y_1, B_i) et (Y_2, B_i). L'ensemble $\{b, e\}$ est éligible pour (Y_3, B_i). La trace de Y_3 est $\mathcal{T}(Y) = \{b\}$.*

— Y est la signature de X dans B_i;

— toutes les arêtes de couplage dans G_i sont propres.

S'il n'existe aucun ensemble éligible pour (Y, B_i), on dit que la signature Y est *incompatible* avec G_i.

La figure 4.14 montre des exemples de signatures incompatibles et un exemple d'un ensemble éligible. On peut caractériser les signatures incompatibles grâce au lemme suivant.

Lemme 19

Soient $(T, \mathcal{X})$ une décomposition arborescente M^*-préservante d'un graphe de solution (G_s^*, M^*, ω) et B_i un sac de $\mathcal{T}$. Soit $Y \in \mathcal{Y}(B_i)$ une signature dans B_i. Y est incompatible avec G_i si et seulement si il existe deux sommets $u, v \in B_i$ tels que $uv \in E(G_i) \setminus M^*$, $Y(u) =$"N" et $Y(v) \neq$"$\times$".

Preuve.

$\Rightarrow$ Montrons que toute autre signature Y' possède un ensemble X éligible pour (Y', B_i). Soit A l'ensemble des extrémités d'arêtes ambiguës dans G_i, prenons $X = \{u \mid Y(u) =$"$\times$"$\} \cup (A \setminus B_i)$. Soient un sommet $u \in B_i$ et une arête $uv \in E(G_i) \setminus M^*$. Supposons que $Y(u) =$"N". Si $v \notin B_i$, alors $v \in X$, sinon comme $Y(v) =$"$\times$", $v \in X$. Ainsi les conditions de la définition 33 sont vérifiées et Y est la signature de X dans B_i. Montrons que X est une solution dans le sous-graphe (G_i, M^*, ω) Pour toute arête de couplage ambiguë uv, on a soit $N_{G_i}(u) \subseteq X, N_{G_i}(v) \subseteq X, u \in X$ ou $v \in X$ et donc X rend l'arête de couplage uv propre et donc X est une solution dans (G_i, M^*, ω). X est éligible pour (Y, B_i).

$\Leftarrow$ Comme $Y(u) =$"N", par la définition 33, on a $Y(v) =$"$\times$" et donc il n'est pas possible de construire un ensemble éligible pour (Y, B_i).

□

Enfin, les entrées de table de l'algorithme dynamique utiliseront la sémantique suivante.

Sémantique

Soit $Y : B_i \mapsto \{"\times","N","\varnothing"\}$ une signature. La table entrée $[Y]_i$ est une solution de score minimum éligible pour (Y, B_i). $[Y]_i = \perp$ si Y est incompatible. On définit $score(\perp) = \infty$ et $\perp \cup X = \perp$ pour tout ensemble de coupes X.

4.5.2.2 Algorithme dynamique

Soit (G_s^*, M^*, ω) un graphe de solution, nous calculons une décomposition arborescente M^*-préservante $(T, \mathcal{X})$ comme décrit précédemment. Nous parcourons les sacs de $(T, \mathcal{X})$ à partir des sacs feuilles et jusqu'au sac racine, noté B_r. Dans chaque sac B_i, nous calculons les entrées de table pour chaque signature $Y \in \mathcal{Y}(B_i)$. Pour obtenir une solution de score minimum dans (G_s^*, M^*, ω), il suffit de prendre la solution dans l'entrée de table $[\varnothing]_r$. Soit B_i un sac de la décomposition arborescente. On dénote par B_j et B_k les enfants de B_i, le cas échéant. Nous calculons les entrées de table $[Y]_i$ en fonction du type de sac de B_i.

— **Sac feuille.** Comme $B_i = \varnothing$, la seule entrée de table est $[\varnothing]_i$ et nous assignons $[\varnothing]_i = \varnothing$.

— **Sac joignant.** Pour toute signature $Y \in \mathcal{Y}(B_i)$, on assigne $[Y]_i = [Y]_j \cup [Y]_\ell$.

— **Sac uv-introduisant.** Nous effectuons les étapes suivantes.

1. Dans un premier temps, considérons que uv est isolée. On réutilise les entrées de table de B_j en complétant les signatures de $\mathcal{Y}(B_j)$ avec les sommets u et v pour que toutes les signatures de $\mathcal{Y}(B_i)$ soient instanciées :

$$\forall Y \in \mathcal{Y}(B_j), Y' \in \mathcal{Y}(\{u, v\}), [Y \cup Y']_i = [Y]_j \cup \mathcal{T}(Y').$$

2. Ensuite, nous introduisons les arêtes non couplantes incidentes à u ou v. Cela permet d'identifier les signatures incompatibles et d'assigner la valeur $\perp$ à leur entrée de table. Soit E' l'ensemble des arêtes n'appartenant pas au couplage et incidentes à u ou à v. Pour toute arête $xy \in E'$ et tout $Y \in \mathcal{Y}(B_i)$, on assigne $[Y]_i = \perp$ si une des deux extrémités de xy est assignée à "N" et l'autre extrémité n'est pas assignée à "$\times$", autrement dit si $\exists x' \in \{x, y\}, y' \in \{x, y\} \setminus \{x'\}$, tels que $Y(x') =$"N" et $Y(y') \neq$"$\times$". Le lemme 19 nous indique que ce sont les seuls cas où la signature Y est incompatible.

— **Sac uv-oubliant.** Pour toute signature $Y \in \mathcal{Y}(B_i)$, on assigne la meilleure solution $[Y']_j$ où Y' donne la même assignation que Y pour les sommets de B_i. Formellement, on assigne

$$[Y]_i = argmin(\{score([Y']_j) \mid Y' \in \mathcal{Y}(B_j), Y'' \in \mathcal{Y}(\{u, v\}), Y' = Y \cup Y''\})$$

Lemme 20

L'algorithme décrit est correct, c'est-à-dire, les valeurs calculées pour les entrées de table correspondent à la sémantique.

Preuve.

Faisons une preuve par induction sur la hauteur de B_i dans la décomposition arborescente. Si la hauteur de B_i est égale à zéro, alors B_i est un sac feuille et $B_i = \varnothing$. Dans ce cas, l'unique ensemble possible dans G_i est l'ensemble vide $\varnothing$. Comme l'unique solution éligible à $\{\varnothing, \varnothing\}$ est également un ensemble vide, l'assignation $[\varnothing]_i = \varnothing$ est correcte et est la seule possible. Supposons à présent que B_i n'est pas un sac feuille et supposons que les valeurs calculées des entrées de table sont correctes pour ses enfants.

— **Sac joignant.** Soient une signature $Y \in \mathcal{Y}(B_i)$ et un sommet $u \in B_i$. Comme $G_j \cap G_\ell = B_i$, une coupe dans l'ensemble $[Y]_j \setminus B_i$ ne peut pas enlever une arête incidente à un sommet de $G_\ell \setminus B_i$. De plus, il existe une solution éligible pour (Y, B_j) si et seulement si il existe une solution éligible pour (Y, B_ℓ). Si $[Y]_j$ rend propres toutes les arêtes ambiguës dans G_j et que $[Y]_\ell$ rend propres toutes les arêtes ambiguës dans G_ℓ, alors $[Y]_j \cup [Y]_\ell$ rend propres toutes les arêtes ambiguës dans $G_i \cup G_\ell$. L'ensemble $[Y]_i \cup [Y]_\ell$ est donc éligible pour (Y, B_i). Supposons qu'il existe un ensemble de coupes X' éligible pour (Y, B_i) tel $score(X') < score([Y]_j \cup [Y]_\ell)$ pour le sous-graphe G_i. Alors, soit $score(X') \cap V(G_j) < score([Y]_j)$ pour le sous-graphe G_j, soit $score(X') \cap V(G_\ell) < score([Y]_\ell)$ pour le sous-graphe G_ℓ. Dans chacun de ces cas, cela contredit l'hypothèse d'induction.

— **Sac uv-introduisant.** Prouvons que les deux étapes décrites sont correctes. Pour la première étape, nous considérons que l'arête couplante introduite est isolée. Les arêtes n'appartenant pas au couplage incidentes à u ou v sont introduites dans la deuxième étape.

1. Quand une arête couplante isolée uv est ajoutée au sous-graphe G_j, celle-ci est propre dans G_i. Si X est une solution pour le sous-graphe G_j, alors tout ensemble de coupes $X \subseteq X'$ est une solution pour le sous-graphe G_i. Pour toute signature $Y \in \mathcal{Y}(B_i)$, comme $[Y]_j$ est une solution de score minimum pour la signature Y dans B_j, pour chaque signature dans $Y' \in \mathcal{Y}(\{u, v\})$, l'ensemble de coupes $[Y]_j \cup \mathcal{T}(Y')$ est une solution de score minimum pour la signature $Y \cup Y'$ dans B_j. L'assignation $[Y \cup Y']_i = [Y]_j \cup \mathcal{T}(Y')$ est correcte si uv est isolée.
2. Introduisons à présent successivement chacune des arêtes n'appartenant pas au couplage incidente à u ou v. Soit E' l'ensemble de ces arêtes. Pour chaque arête $xx' \in E'$ et pour toute signature $Y \in \mathcal{Y}(B_i)$, si $Y(x) =$"N"$\wedge Y(x') \neq$"$\times$" ou si $Y(x') =$"N"$\wedge Y(x) \neq$"$\times$", alors par le lemme 19, la signature Y est incompatible et donc l'assignation $[Y]_i = \perp$ est correcte. Par réciprocité si après ces assignations, une entrée de table $[Y]_i \neq \perp$, alors il existe un ensemble de coupes compatible avec (Y, B_i). Montrons que dans ce cas, $[Y]_i$ contient une solution de score minimum pour la signature Y dans B_i. Supposons qu'il existe une solution X' ayant pour signature Y dans B_i et telle que $score(X') < score([Y]_i)$. Soient deux signatures $Y' \in \mathcal{Y}(B_j)$ et $Y'' \in \mathcal{Y}(\{u, v\})$ telles que $Y = Y' \cup Y''$. Comme $X' \cap \mathcal{T}(Y) = [Y]_i \cap \mathcal{T}(Y)$, cela signifie que $score(X' \setminus (\mathcal{T}(Y) \cap \{u, v\})) < score([Y']_j)$, ce qui contredit l'hypothèse d'induction.

3. **Sac uv-oubliant.** Comme $G_i = G_j$, chaque solution dans le sous-graphe G_j est également une solution dans le sous-graphe G_i. Pour chaque signature $Y \in \mathcal{Y}(B_i)$, toutes les solutions de l'ensemble $A = \{[Y']_j \mid Y' \in \mathcal{Y}(B_j), Y'' \in (\{u, v\}), Y' = Y \cup Y''\}$ ont Y comme signature Y dans B_i. L'entrée de table $[Y]_i$ enregistre une solution de A ayant le score minimum. Ainsi, l'assignation $[Y]_i = \{score([Y']_j) \mid Y' \in \mathcal{Y}(B_j), Y'' \in (\{u, v\}), Y' = Y \cup Y''\}$ est correcte.

□

Dans chacun des sacs B_i, nous devons itérer toutes les signatures dans $\mathcal{Y}(B_i)$. Comme remarqué plus haut, pour une unique arête de couplage le nombre de signatures possibles est égal à deux pour le score de poids et est égal à cinq pour le score de poids. Ainsi, pour un sac possédant t arêtes de couplage, il faudra effectuer 2^t itérations pour le score de poids et 5^t itérations pour le score de coupe. De plus, comme le nombre de sacs dépend du nombre d'arêtes dans le graphe de solution, nous obtenons une complexité en temps de $\mathcal{O}(2^{tw^*} \cdot |E(G_s^*)|)$ pour le score de poids et de $\mathcal{O}(5^{tw^*} \cdot |E(G_s^*)|)$ pour le score de coupe.

Corollaire 9

Soit (G_s^*, M^*, ω) un graphe de solution et $(T, \mathcal{X})$ une décomposition arborescente M^*-préservante de (G_s^*, M^*, ω). COUPE SEMI-BRUTALE peut être résolu en temps $\mathcal{O}(2^{tw^*} \cdot |E(G_s^*)|)$ pour le score de poids et en temps $\mathcal{O}(5^{tw^*} \cdot |E(G_s^*)|)$ pour le score de coupe.

4.6 Expériences

Nous indiquons à présent dans cette section les performances des algorithmes présentés dans les sections 4.4 et 4.5 sur des instances réelles. Nous avons implémenté l'algorithme 2-approché pour le score de poids ainsi que les deux algorithmes exacts utilisant la formulation linéaire en nombres entiers et la décomposition arborescente. Notons que nous ne présenterons pas les résultats de l'algorithme 4-approché pour le score de coupe car le nombre de coupes effectuées est trop large pour qu'il soit performant. En effet, l'existence de cet algorithme est surtout important pour la borne supérieure d'approximation mais cet algorithme n'a pas vocation à être performant sur les instances réelles car il aura tendance à couper quasiment tous les sommets.

4.6.1 Instances sélectionnées

L'obtention du jeu d'instance est détaillée dans la sous-sous-section 2.5.2.2. Le tableau 4.1 présente des statistiques sur ces instances une fois que toutes les règles de réduction présentées dans la sous-section 2.4.6 ont été appliquées. Une première limitation venant de ce jeu vient de la taille des instances : en effet comme chaque chemin

uniforme a été contracté en une seule arête, cela réduit grandement la taille de l'instance. Ainsi, à l'exception de `anopheles`, les graphes de solution ne vont contenir que quelques dizaines de sommets. Il sera donc difficile d'observer des différences significatives lors des tests, mais on pourra tout de même obtenir une première indication. Une autre indication que l'on peut retirer de ces statistiques est que la largeur arborescente semble petite même pour une instance assez grande comme `anopheles` : la valeur de cette arborescente est située entre un et quatre dans ce jeu de données. Cela peut s'expliquer par le fait que les graphes de solution réels sont peu denses.

TABLE 4.1 – *Instances utilisées pour tester* COUPE SEMI-BRUTALE.

Donnée	Arêtes ambiguës	Arêtes non ambiguës	Poids total	Degré moyen	Degré max.	Degré min.	Largeur arborescente
anopheles	1 523	7 695	14 937	2.4	6	2	3
anthrax	13	260	329	2.7	4	2	2
gloeobacter	44	432	694	2.8	6	2	2
lactobacillus	15	135	225	2.6	5	2	2
pandora	5	183	210	2.5	4	2	1
pseudomonas	47	413	650	2.6	5	2	2
rice	6	9	29	2.1	3	2	3
sacchr3	5	25	54	2.7	4	2	2
sacchr12	23	74	190	2.4	4	2	4

4.6.2 Algorithmes exacts

Les statistiques des résultats pour les algorithmes exacts sont résumés dans le tableau 4.2. Nous avons utilisé le logiciel `ILOG CPLEX` [3] pour résoudre le modèle linéaire en nombres entiers. Pour la programmation dynamique, nous avons utilisé la bibliothèque `HTD` [7] pour calculer une décomposition arborescente du graphe de solution. Notons également que pour réduire la largeur de la décomposition, nous avons enlevé les sommets des arêtes de couplage non ambiguës du graphe de solution; en rajoutant cette information sur les sommets voisins des sommets enlevés pour que l'algorithme calcule bien une solution minimum.

Dans un premier temps, nous pouvons remarquer que pour l'algorithme utilisant la décomposition arborescente, une solution optimale est plus rapide à calculer pour le score de poids que pour le score de coupe. Cela est cohérent avec la différence de complexité qui est montrée par le corollaire 9. Pour la formulation en nombres entiers, il ne semble pas y avoir une fonction de score qui soit plus rapide à calculer.

Pour comparer les temps d'exécution de ces deux algorithmes, la méthode utilisant la décomposition arborescente est plus rapide pour les deux fonctions de score sauf pour l'instance la plus grande `anopheles` : pour cette instance le score de coupe est plus rapide à calculer avec `CPLEX`. Encore une fois, cela est cohérent avec la complexité théorique calculée dans le corollaire 9. Comme `CPLEX` est un outil très puissant, on pourrait penser qu'à terme, la formulation en nombres entiers supplantera l'algorithme utilisant la décomposition arborescente. Cependant, comme la largeur arbores-

TABLE 4.2 – *Résultats pour les méthodes exactes. Les colonnes « ILP » et « Dec. Arb. » indiquent les temps d'exécution en secondes pour la méthode utilisant une programmation linéaire par nombres entiers et pour la méthode utilisant la décomposition arborescente, respectivement.*

Données	Score de coupe			Score de poids		
	Score	ILP	Dec. Arb.	Score	ILP	Dec. Arb.
anopheles	1 093	5.05	5.80	1 387	5.32	4.64
anthrax	12	0.43	0.36	17	0.46	0.35
gloeobacter	39	0.52	0.40	67	0.54	0.39
lactobacillus	13	0.21	0.16	18	0.21	0.16
pandora	5	0.27	0.20	6	0.26	0.20
pseudomonas	36	0.56	0.46	51	0.58	0.45
rice	3	0.01	0.00	3	0.01	0.00
sacchr3	3	0.03	0.02	5	0.03	0.02
sacchr12	12	0.09	0.08	18	0.1	0.08

cente semble rester faible même pour les instances de plus grande taille, ce n'est pas forcément très clair.

4.6.3 Algorithme d'approximation

Les statistiques des résultats pour l'algorithme d'approximation sont résumées dans le tableau 4.3. Dans ce tableau, on a comparé les performances de la 2-approximation avec les deux algorithmes exacts. Une première tendance concernant les scores indique que les facteurs d'approximation sur les instances réelles sont assez éloignés du facteur théorique de deux. La plus grande valeur pour ce facteur est obtenue avec l'instance `rice`, mais comme la différence entre les deux scores n'est que de une unité, cela n'est pas forcément significatif. Pour les autres instances, ce facteur est compris entre 1 et 1.2. Une solution optimale est même obtenue pour quatre instances. On peut donc penser que pour des instances plus grandes, ce facteur restera proche de 1.

Concernant les temps de calculs, la solution approchée est toujours obtenue plus rapidement que celle calculée avec la formulation en nombres entiers mais moins rapidement que celle calculée avec la décomposition arborescente. Un algorithme approché n'a d'intérêt que s'il est plus rapide qu'un algorithme exact. Cependant, il faut garder à l'esprit que les deux bibliothèques `ILOG CPLEX` et `HTD` contiennent énormément d'optimisations. On peut penser que ces optimisations permettent d'être plus rapide pour les petites instances. Comme la complexité des algorithmes exacts est exponentielle et que celle de l'algorithme approché est polynomiale, on peut légitimement penser que la 2-approximation sera à terme plus rapide que les deux algorithmes exacts.

4.7 Conclusion

Dans ce chapitre, nous avons exploré la difficulté du problème COUPE SEMI-BRUTALE, complétant ainsi les résultats donnés dans [88]. Nous avons pu ainsi avoir

TABLE 4.3 – *Comparaison de l'algorithme d'approximation avec les deux méthodes exactes. La valeur donnée par la colonne « Facteur » correspond au quotient du score approché par le score optimal. Les colonnes « ILP » et « Dec. Arb. » indiquent les temps d'exécution en secondes pour la méthode utilisant une programmation linéaire par nombres entiers et pour la méthode utilisant la décomposition arborescente, respectivement.*

Données	Approximation			Exact		
	Score	Temps	Facteur	Score	ILP	Dec. Arb.
anopheles	1 465	5.06	1.06	1 387	5.32	4.64
anthrax	17	0.44	1	17	0.46	0.35
gloeobacter	67	0.5	1	67	0.54	0.39
lactobacillus	18	0.2	1	18	0.21	0.16
pandora	6	0.26	1	6	0.26	0.20
pseudomonas	53	0.57	1.04	51	0.58	0.45
rice	4	0.01	1.33	3	0.01	0.00
sacchr3	6	0.03	1.2	5	0.03	0.02
sacchr12	21	0.09	1.17	18	0.1	0.08

un horizon assez complet sur la difficulté de calcul de solutions exactes ou approchées, que ce soit au niveau des graphes denses ou peu denses. Nous avons également développé des algorithmes approchés et exacts pouvant optimiser les deux fonctions de score. Enfin, des tests ont été effectués afin d'observer la performances de ces algorithmes sur des instances réelles.

Une prolongation de ce travail pourrait être de trouver des algorithmes approchés avec un meilleur facteur d'approximation, particulièrement pour le score de coupe. En effet, pour l'algorithme approché du score de coupe, même s'il permet de statuer quant à l'appartenance de COUPE SEMI-BRUTALE à la classe APX, son facteur d'approximation de quatre est assez élevé et son comportement sur des instances n'est pas satisfaisant car il a tendance à couper presque l'intégralité des sommets. On aimerait également pouvoir effectuer des tests impliquant des instances plus grandes afin de pouvoir confirmer la très légère tendance qui s'est dégagée de notre propre jeu de tests et de pouvoir conclure si une approche est effectivement meilleure que les autres.

Enfin, une autre approche pour résoudre le problème de la linéarisation serait d'utiliser les nouvelles technologies de séquençage, en particulier celle produisant des lectures longues. En effet, si une lecture longue traverse tout un chemin ambigu d'un graphe de solution, cela donne des indications sur la décomposition de ce graphe de solution en séquences ne contenant pas de séquences chimériques, comme l'explique la figure 4.15. Cette méthode aurait l'avantage de ne pas détruire les arêtes du graphe et donc de conserver davantage d'informations biologiques.

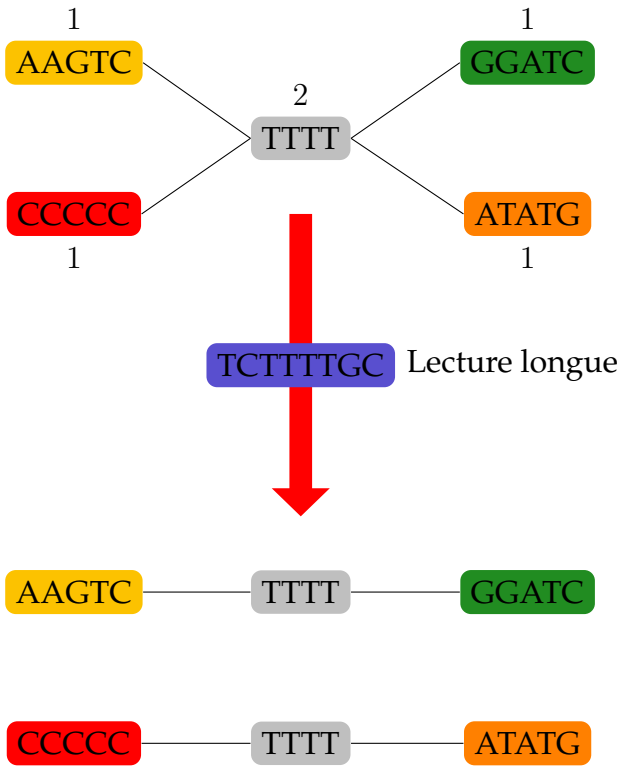


FIGURE 4.15 – *Utilisation des lectures longues pour résoudre le problème de la linéarisation. En haut, tous les contigs ont une multiplicité égale à un, à l'exception du contig* `TTTT` *qui a une multiplicité égale à deux. Le contig* `TTTT` *est donc un chemin ambigu. La lecture longue* `TCTTTTGC` *chevauche à la fois le contig* `AAGTC` *et le contig* `GGATC`*, on peut donc suspecter que ces deux contigs sont dans la même séquence.*

Conclusion

Dans cette thèse, nous avons étudié deux problèmes algorithmiques issus du séquençage de la molécule d'ADN : l'échafaudage et la linéarisation. Le problème de l'échafaudage permet de reconstruire des séquences génomiques à l'aide des contigs issus de l'assemblage. La linéarisation permet d'éviter d'obtenir des séquences chimériques lorsque la multiplicité des contigs est prise en compte dans l'échafaudage. La nature des travaux présentés dans cette thèse est double. D'une part ils ont pour but d'étudier la complexité de ces deux problèmes, sous différentes hypothèses. D'autre part, ils ont permis de créer des algorithmes efficaces, exacts ou approchés, pour la résolution de ces problèmes.

Pour le problème de l'échafaudage, nous avons montré l'appartenance à des classes de complexité pour certains cas restreints de graphes d'échafaudage. La principale contribution pour la résolution de ce problème consiste en la généralisation d'un algorithme approché glouton. Nous avons montré comment un algorithme glouton, s'exécutant originellement sur des graphes complets, pouvait se généraliser sur d'autres classes de graphes, à condition de formuler une fonction de faisabilité pour cette classe de graphe particulière. Nous avons décrit deux fonctions de faisabilité, une dans les graphes de clusters connectés et une dans le cas général. Pour la première fonction de faisabilité, on utilise le fait que la structure d'un graphe de clusters connectés est proche d'un arbre pour créer un algorithme dynamique en temps polynomial. Pour le cas général, une formulation SAT est décrite, permettant ensuite l'utilisation d'un solveur pour la résolution. Les résultats empiriques semblent montrer que plus la fonction de faisabilité est proche de l'instance de départ, meilleurs sont les résultats.

Pour le problème de la linéarisation, nous avons construit un panorama assez large de la complexité de ce problème, que ce soit au niveau de la difficulté de calcul, la difficulté d'approximation ou la difficulté d'obtenir une solution locale. Ce panorama va des graphes peu denses, comme les chemins ou les arbres, aux graphes très denses, comme les graphes complets ou les graphes scindés. En l'occurrence, nous avons montré que la linéarisation peut être résolue de façon polynomiale sur les graphes peu denses. Le problème devient NP-difficile lorsqu'on augmente légèrement la densité, par exemple il n'est pas possible de résoudre de façon polynomiale ce problème dans les graphes bipartis, planaires et sous-cubiques. Lorsqu'on regarde les graphes très denses, le comportement de ce problème diffère suivant la fonction de score utilisée : pour le score de coupe, le problème redevient simple à résoudre, mais ce n'est pas le cas pour le score de poids. Enfin, nous avons également développé des algorithmes exacts et approchés. Un des algorithmes exacts utilise une formulation linéaire en nombres entiers et un deuxième utilise une décomposition arborescente du graphe. Ces deux

algorithmes exacts peuvent calculer des solutions optimales pour les deux fonctions de score. Pour les algorithmes approchés, pour chaque fonction de score, nous avons décrit un algorithme approché avec un facteur d'approximation constant. Le jeu de tests effectués semble montrer que tous ces algorithmes sont performants (à l'exception de l'algorithme approché pour le score de coupe), même sur un génome assez important comme celui du moustique anopheles gambiae.

Les travaux présentés dans cette thèse ouvrent un champ de perspectives pour des futurs résultats, notamment pour l'échafaudage. Nous avons déjà évoqué une question ouverte sur le comportement des problèmes pour lesquels il est à la fois difficile de calculer n'importe quelle solution dans le cas général et à la fois difficile de donner une bonne approximation pour certains cas plus triviaux. Nous avons aussi décrit de la façon dont l'algorithme glouton pouvait être amélioré. En effet, les résultats des tests tendent à montrer qu'utiliser une fonction de faisabilité nécessitant de compléter l'instance avec moins d'arêtes permet d'obtenir de meilleurs résultats. Créer des fonctions de faisabilité qui s'exécutent sur des graphes moins denses permettrait de produire de meilleures solutions, possiblement optimales. Une autre possibilité pourrait être de produire une fonction de faisabilité dans le cas général, qui cette fois-ci ne pourrait pas être polynomiale. Nous avons déjà vu que la formulation SAT semble être une fausse piste (du moins sans prétraitements). On pourrait cependant utiliser d'autres outils pour la produire, quitte à utiliser des méthodes hybrides entre des heuristiques et des formulations en nombres entiers ou réels pour garder l'exécution dans un temps raisonnable.

On pourrait également envisager des variantes de la formulation du problème d'échafaudage. On peut penser notamment à des formulations où le nombre de cycles alternés et de chemins alternés à rechercher n'est pas strict. Par exemple, on pourrait introduire une pénalité sur le score d'une solution en fonction du nombre de la différence entre la cardinalité de cette solution et la cardinalité d'une solution attendue. Cela aurait l'avantage de rendre facile la production d'une solution et ouvrirait des perspectives pour l'utilisation de métaheuristiques. Une autre possibilité serait de prendre en compte les lectures longues ou les possibilités de *saut de contig* (*contig jumps* en anglais). Dans certains cas, quand la taille d'un contig est plus petite que la taille d'insertion de certaines lectures appariées, celui-ci s'intègre mal dans la séquence (voir la figure 4.16); du moins dans la façon dont le problème est formulé. La solution serait de conserver la taille des contigs et une estimation de la distance entre deux contigs afin de repérer d'éventuels sauts de contigs.

Il faut également garder à l'esprit, que la façon dont est modélisé le problème n'est qu'une abstraction de la réalité. Ainsi, il peut arriver que plusieurs solutions optimales soient possibles, ou même que la solution réelle ne corresponde pas à une solution optimale. Il pourrait être intéressant de trouver un moyen d'énumérer toutes les solutions possédant un poids supérieur à une certaine valeur. Bien entendu, comme la plupart des instances ne sont pas résolvables de façon polynomiale, il s'agit d'un calcul difficile à effectuer. Cependant, le calcul d'une bonne solution pourrait permettre de donner des indices sur les arêtes « vitales », c'est-à-dire les arêtes qui doivent être présentes dans une solution si l'on veut que son poids dépasse le seuil fixé.

Enfin, l'idée finale de tous ces travaux serait de pouvoir comparer les algorithmes

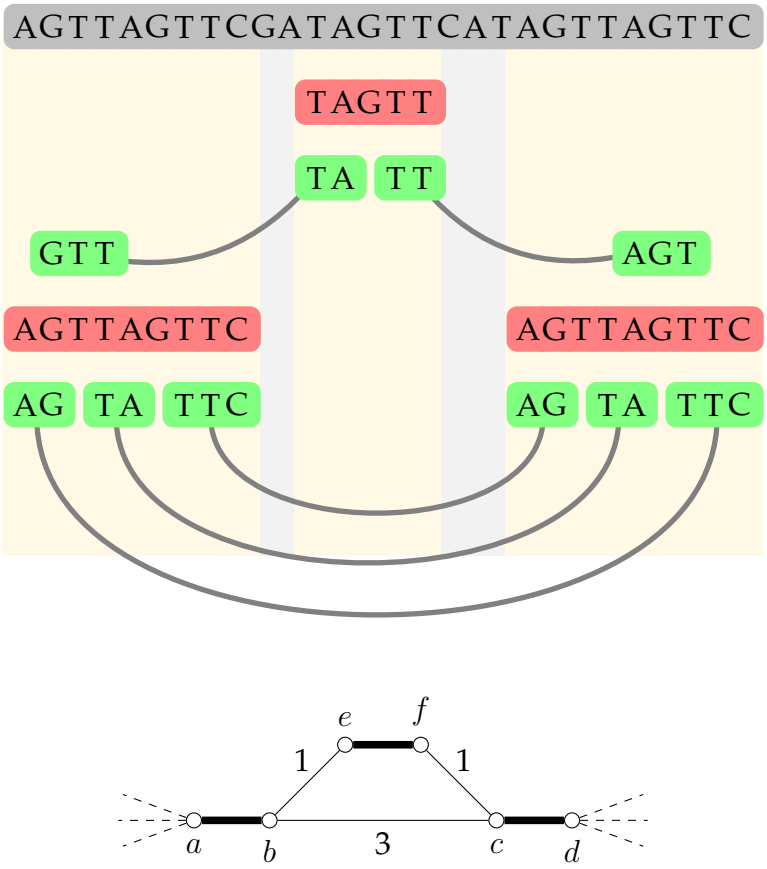


FIGURE 4.16 – *Problème des sauts de contigs. La séquence grise est la séquence originelle, de celle-ci on a extrait trois contigs, représentés par les séquences rouges. Les séquences vertes sont les lectures, on a représenté leur appariemment par une liaison grise. La taille d'insertion des paires de lectures varie dans cet exemple de sept à vingt-cinq. Le contig central de la séquence a une longueur égale à cinq, ce qui fait que sa séquence se trouve dans la zone d'insertion de la plupart des paires. Le graphe d'échafaudage en bas est celui obtenu avec cet ensemble de contigs : les arêtes ab, cd et ef correspondent aux contigs* `AGTTAGTTC`, `AGTTAGTTC`, `TAGTT`, *respectivement. On remarque qu'on obtient un poids plus grand quand on construit deux chemins alternés plutôt qu'un seul, alors que cela ne correspond pas à la séquence originelle.*

développés aux outils existants dans la littérature.

Concernant le problème de la linéarisation, nous avons déjà construit un panel assez large de la complexité de ce problème. Une prolongation naturelle de ce travail serait de le compléter en étudiant d'autres classes de graphes et en établissant d'autres bornes d'approximation. On pourrait également tenter de caractériser structurellement les instances réelles : en effet comme celles-ci ont été construites de façon particulière (*i.e.* en fusionnant des marches alternées), elles possèdent probablement des propriétés structurelles que l'on pourrait utiliser pour améliorer la résolution du problème. Enfin, d'autres tests doivent être effectués afin de déterminer si une des méthodes présentées supplante les autres, à terme.

Bibliographie

[1] Boost C++ Libraries. Disponible sur http://www.boost.org/. 74

[2] bwa. Disponible sur http://bio-bwa.sourceforge.net/. 76

[3] Ibm ilog cplex. Disponible sur https://www.ibm.com/products/ilog-cplex-optimization-studio. 171

[4] megablast. Fait partie de la suite NCBI C++ toolkit. Disponible sur https://ncbi.github.io/cxx-toolkit/. 77

[5] minia. Disponible sur http://minia.genouest.org/. 76

[6] wgsim. Disponible sur https://github.com/lh3/wgsim. 75

[7] Michael Abseher, Nysret Musliu, and Stefan Woltran. HTD - A free, open-source framework for (customized) tree decompositions and beyond. In *Proceedings of the 14th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2017)*, volume 10335 of *LNCS*, pages 376–386. Springer, 2017. 171

[8] Sergey Aganezov, Nadia Sitdykova, AGC Consortium, and Max A. Alekseyev. Scaffold assembly based on genome rearrangement analysis. *Computational Biology and Chemistry*, 57:46-53, 2015. 57

[9] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979. 36

[10] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. 123

[11] Cristina Bazgan, Bruno Escoffier, and Vangelis T. Paschos. Completeness in standard and differential approximation classes: Poly-(d)apx- and (d)ptas-completeness. *Theoretical Computer Science*, 339(2-3):272–292, 2005. 86

[12] Piotr Berman and Marek Karpinski. On some tighter inapproximability results (extended abstract). In *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, pages 200–209, 1999. 153, 156

[13] Piotr Berman, Marek Karpinski, and Alex D. Scott. Approximation hardness and satisfiability of bounded occurrence instances of SAT. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(022), 2003. 150

[14] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 222–233, 2004. 44

[15] Marten Boetzer, Christiaan V. Henkel, Hans J. Jansen, Derek Butler, and Walter Pirovano. Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, 27(4):578-579, 2011. 58

[16] John A. Bondy and Uppaluri S. R. Murty. *Graph Theory with Applications*. North Holland, 1985. 16

[17] Nicolas Briot, Annie Chateau, Remi Coletta, Simon de Givry, Philippe Leleux, and Thomas Schiex. An Integer Linear Programming Approach for Genome Scaffolding. In *Workshop on Constraint-Based Methods in Bioinformatics*, 2014. 58

[18] Joshua N. Burton, Andrew Adey, Rupali P. Patwardhan, Ruolan Qiu, Jacob O. Kitzman, and Jay Shendure. Chromosome-scale scaffolding of de novo genome assemblies based on chromatin interactions. *Biotechnology*, 31(12):1119–1125, December 2013. 57

[19] Jonathan F. Buss and Tarique Islam. Simplifying the weft hierarchy. *Theoretical Computer Science*, 351(3):303–313, 2006. 40

[20] Annie Chateau and Rodolphe Giroudeau. A complexity and approximation framework for the maximization scaffolding problem. *Theoretical Computer Science*, 595:92–106, 2015. 58, 59, 79, 83, 89, 119

[21] Cédric Chauve, Murray Patterson, and Ashok Rajaraman. Hypergraph covering problems motivated by genome assembly questions. In *Combinatorial Algorithms - 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers*, pages 428–432, 2013. 57

[22] Zhi-Zhong Chen, Youta Harada, Eita Machida, Fei Guo, and Lusheng Wang. Better approximation algorithms for scaffolding problems. In *Frontiers in Algorithmics, 10th International Workshop, FAW 2016, Qingdao, China, June 30- July 2, 2016, Proceedings*, pages 17–28, 2016. 83, 126

[23] Shuya Chiba and Shinya Fujita. Covering vertices by a specified number of disjoint cycles, edges and isolated vertices. *Discrete Mathematics*, 313(3):269–277, 2013. 58

[24] R. Chikhi and G. Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In *WABI*, pages 236–248, 2012. 76

[25] Planck Collaboration, Nabila. Aghanim, et al. Planck 2018 results. vi. cosmological parameters, 2018. 31

[26] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971. 35, 36

[27] Clément Dallard, Mathias Weller, Annie Chateau, and Rodolphe Giroudeau. Instance guaranteed ratio on greedy heuristic for genome scaffolding. In *Combinatorial Optimization and Applications - 10th International Conference, COCOA 2016, Hong Kong, China, December 16-18, 2016, Proceedings*, pages 294–308, 2016. 59, 89, 119

[28] Tom Davot, Annie Chateau, Rodolphe Giroudeau, and Mathias Weller. On the hardness of approximating linearization of scaffolds sharing repeated contigs. In *Comparative Genomics - 16th International Conference, RECOMB-CG 2018, Magog-Orford, QC, Canada, October 9-12, 2018, Proceedings*, pages 91–107, 2018. 12

[29] Tom Davot, Annie Chateau, Rodolphe Giroudeau, and Mathias Weller. New polynomial-time algorithm around the scaffolding problem. In *Algorithms for Computational Biology - 6th International Conference, AlCoB 2019, Berkeley, CA, USA, May 28-30, 2019, Proceedings*, pages 25–38, 2019. 12

[30] Tom Davot, Annie Chateau, Rodolphe Giroudeau, and Mathias Weller. Linearizing genomes: Exact methods and local search. In *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, pages 505–518, 2020. 12

[31] Adel Dayarian, Todd P. Michael, and Anirvan M. Sengupta. SOPRA: scaffolding algorithm for paired reads via statistical optimization. *BMC Bioinformatics*, 11:345, 2010. 57

[32] Mark de Berg and Amirali Khosravi. Optimal binary space partitions for segments in the plane. *International Journal of Computational Geometry and Applications*, 22(3):187–206, 2012. 36, 137, 139

[33] Reinhard Diestel. *Graph Theory*. Graduate Texts in Mathematics. Springer, 5 edition, 2017. 16

[34] Irit Dinur and Samuel Safra. On the hardness of approximation minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005. 155

[35] Nilgun Donmez and Michael Brudno. SCARPA: scaffolding reads with practical algorithms. *Bioinformatics*, 29(4):428–434, 2013. 58

[36] Rodney G. Downey and Michael R. Fellows. Fixed parameter tractability and completeness. In *Complexity Theory: Current Research, Dagstuhl Workshop, February 2-8, 1992*, pages 191–225, 1992. 39

[37] Song Gao, Wing-Kin Sung, and Niranjan Nagarajan. Opera: Reconstructing optimal genomic scaffolds with high-throughput paired-end sequences. *Journal of Computational Biology*, 18(11):1681–1691, 2011. 58

[38] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990. 32

[39] Alexey A. Gritsenko, Jurgen F. Nijkamp, Marcel J. T. Reinders, and Dick de Ridder. GRASS: a generic algorithm for scaffolding next-generation sequencing assemblies. *Bioinformatics*, 28(11):1429–1437, 2012. 58

[40] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, 2001. 150, 152, 153

[41] Edward A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000. 41

[42] Martin Hunt, Chris Newbold, Matthew Berriman, and Thomas Otto. A comprehensive evaluation of assembly scaffolding tools. *Genome biology*, 15:R42, 03 2014. 58, 76

[43] Peter Husemann and Jens Stoye. Phylogenetic comparative assembly. *Algorithms for Molecular Biology*, 5:3, 2010. 57

[44] Daniel H. Huson, Knut Reinert, and Eugene W. Myers. The greedy path-merging algorithm for contig scaffolding. *Journal of the ACM*, 49(5):603–615, 2002. 57

[45] Russell Impagliazzo and Ramamohan Paturi. Complexity of k-sat. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity, Atlanta, Georgia, USA, May 4-6, 1999*, pages 237–240, 1999. 40

[46] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. 41, 88

[47] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, 1988. 48

[48] Sanjeev Khanna, editor. *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. SIAM, 2013. 53

[49] Subhash Khot. On the unique games conjecture. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, page 3, 2005. 44

[50] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable csps? *SIAM Journal on Computing*, 37(1):319–357, 2007. 152

[51] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *Journal of Computer and System Sciences*, 74(3):335–349, 2008. 155

[52] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994. 164

[53] Mikhail Kolmogorov, Brian J. Raney, Benedict Paten, and Son K. Pham. Ragout - a reference-assisted assembly tool for bacterial genomes. *Bioinformatics*, 30(12):302–309, 2014. 57

[54] Rastislav Kralovic and Pawel Urzyczyn, editors. *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*. Springer, 2006. 38

[55] Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM Journal on Computing*, 19(4):742–749, 1990. 48

[56] Michael Krivelevich, Zeev Nutov, Mohammad R. Salavatipour, Jacques Yuster, and Raphael Yuster. Approximation algorithms and hardness results for cycle packing problems. *ACM Trans. Algorithms*, 3(4):48, 2007. 58

[57] Richard E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975. 37

[58] Eugene L. Lawler. The traveling salesman problem: A guided tour of combinatorial optimization. *Journal of the Operational Research Society*, 37:535–536, 1985. 58

[59] Euler Leonhard. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 8:128–140, 1736. 16

[60] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981. 28

[61] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. 76

[62] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009. 75

[63] David Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982. 140

[64] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Lower bounds based on the exponential time hypothesis. *Bulletin of the EATCS*, 105:41–72, 2011. 140

[65] Vadim Lozin and Martin Milanič. On the maximum independent set problem in subclasses of planar graphs. *Journal of Graph Algorithms and Applications*, 14(2):269–286, 2010. 80

[66] Nina Luhmann, Cédric Chauve, Jens Stoye, and Roland Wittler. Scaffolding of ancient contigs and ancestral reconstruction in a phylogenetic framework. In *Proceedings of Brazilian Symposium on Bioinformatics*, volume 8826 of *Lecture Notes in Computer Science*, pages 135-143, 2014. 57

[67] Junwei Luo, Jianxin Wang, Zhen Zhang, Min Li, and Fang-Xiang Wu. BOSS: a novel scaffolding algorithm based on an optimized scaffold graph. *Bioinformatics*, 33(2):169–176, 09 2016. 58

[68] Igor Mandric and Alex Zelikovsky. Scaffmatch: Scaffolding algorithm based on maximum weight matching. In Teresa M. Przytycka, editor, *Research in Computational Molecular Biology*, pages 222–223, Cham, 2015. Springer International Publishing. 58

[69] Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L. Madden, Richa Agarwala, , and Alejandro A. Schäffer. Database indexing for production MegaBLAST searches. *Bioinformatics*, 24(16):1757–1764, 2008. 77

[70] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991. 44

[71] Christos H. Papadimitriou and Mihalis Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research*, 18(1):1–11, 1993. 83

[72] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation.*, 2(3):293–304, 1986. 115

[73] Kari-Jouko Räihä and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is np-complete. *Theoretical Computer Science*, 16:187–198, 1981. 53

[74] Ashok Rajaraman, éric Tannier, and Cédric Chauve. FPSAC: Fast Phylogenetic Scaffolding of Ancient Contigs. *Bioinformatics*, 29(23):2987-2994, 2013. 57

[75] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986. 20

[76] Kristoffer Sahlin, Francesco Vezzi, Björn Nystedt, Joakim Lundeberg, and Lars Arvestad. BESST - Efficient scaffolding of large fragmented assemblies. *BMC Bioinformatics*, 15(1):281, 2014. 58

[77] Leena Salmela, Veli Mäkinen, Niko Välimäki, Johannes Ylinen, and Esko Ukkonen. Fast scaffolding with small independent mixed integer programs. *Bioinformatics*, 27:3259-3265, 2011. 57

[78] Leena Salmela and Éric Rivals. LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, August 2014. 52

[79] Frederick Sanger et al. Nucleotide sequence of bacteriophage phi x174 dna, Feb 1977. 50

[80] Koren Sergey, Treangen Todd J, and Pop Mihai. Bambus 2: Scaffolding metagenomes. *Bioinformatics*, 27(21):2964-2971, 2011. 56

[81] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. 123

[82] George Steiner. On the k-path partition of graphs. *Theoretical Computer Science*, 290(3):2147–2155, 2003. 58

[83] Dorine Tabary, Tom Davot, Annie Chateau, Rodolphe Giroudeau, and Mathias Weller. New results about the linearization of scaffolds sharing repeated contigs. In *COCOA 2018*, Lecture Notes in Computer Science, pages 94–107. Springer, 2018. 12, 151

[84] William T. Tutte. A short proof of the factor theorem for finite graphs. 1954. 58

[85] Mathias Weller, Annie Chateau, Clément Dallard, and Rodolphe Giroudeau. Scaffolding problems revisited: Complexity, approximation and fixed parameter tractable algorithms, and some special cases. *Algorithmica*, 80(6):1771–1803, 2018. 59, 88

[86] Mathias Weller, Annie Chateau, and Rodolphe Giroudeau. Exact approaches for scaffolding. *BMC bioinformatics*, 16(Suppl 14):S2, 2015. 59

[87] Mathias Weller, Annie Chateau, and Rodolphe Giroudeau. On the complexity of scaffolding problems: From cliques to sparse graphs. In *Combinatorial Optimization and Applications - 9th International Conference, COCOA 2015, Houston, TX, USA, December 18-20, 2015, Proceedings*, pages 409–423, 2015. 59

[88] Mathias Weller, Annie Chateau, and Rodolphe Giroudeau. On the linearization of scaffolds sharing repeated contigs. In *Combinatorial Optimization and Applications - 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part II*, pages 509–517, 2017. 73, 128, 155, 156, 172

[89] Mathias Weller, Annie Chateau, Rodolphe Giroudeau, and Michael Poss. Scaffolding with repeated contigs using flow formulations. 77

Index

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR

À la recherche de l'échafaudage parfait : efficace, de qualité et garanti

Tom Davot

Résumé

Le séquençage est un processus en biologie qui permet de déterminer l'ordre des nucléotides au sein de la molécule d'ADN. Le séquençage produit un ensemble de fragments, appelés lectures, dans lesquels l'information génétique est connue. Seulement, la séquence génomique n'est connue que de façon parcellaire, pour pouvoir faire son analyse, il convient alors de la reconstituer à l'aide d'un certain nombre de traitements informatiques. Dans cette thèse, nous avons étudié deux problèmes mathématiques issus de ce séquençage : l'échafaudage et la linéarisation.

L'échafaudage est un processus qui intervient après l'assemblage des lectures en contigs. Il consiste en la recherche de chemins et de cycles dans un graphe particulier appelé graphe d'échafaudage. Ces chemins et cycles représentent les chromosomes linéaires et circulaires de l'organisme dont l'ADN a été séquencée. La linéarisation est un problème annexe à l'échafaudage : quand on prend en compte le fait que les contigs puissent apparaitre plusieurs fois dans la séquence génomique, des ambiguïtés surviennent dans le calcul d'une solution. Celles-ci, si elles ne sont pas traitées, peuvent entrainer la production d'une séquence chimérique lors de l'échafaudage. Pour résoudre ce problème, il convient alors de dégrader de façon parcimonieuse une solution calculée par l'échafaudage. Dans tous les cas, ces deux problèmes peuvent être modélisés comme des problèmes d'optimisation dans un graphe.

Dans ce document, nous ferons l'étude de ces deux problèmes en se concentrant sur trois axes. Le premier axe consiste à classifier ces problèmes au sens de la complexité. Le deuxième axe porte sur le développement d'algorithmes, exacts ou approchés, pour les résoudre. Enfin, le dernier axe consiste à implémenter et tester ces algorithmes pour observer leurs comportements sur des instances réelles.

Mot clés : Bio-informatique, Graphes, Complexité, Approximation, Algorithmes

Abstract

Sequencing is a process in biology that determines the order of nucleotides in the DNA. It produces a set of fragments, called reads, in which the genetic information is known. Unfortunatly, the genomic sequence is decomposed in small pieces. In order to analyse it, it is necessary to reconstruct it using a number of computer processes. In this thesis, we studied two mathematical problems arising from this sequencing: the scaffolding and the linearization.

The scaffolding is a process that takes place after the reads assembly into larger subsequences called contigs. It consists in the search of paths and cycles in a particular graph called scaffold graph. These paths and cycles represent the linear and circular chromosomes of the organism whose DNA has been sequenced. The linearization is a problem related to the scaffolding. When we take into account that contigs may appear several times in the genomic sequence, some ambiguities can arise. If this ambiguities are not deleted, then a chimeric sequence may be produced by the scaffolding. To solve this problem, a solution computed by the scaffolding should be wisely deteriorated. In any case, both problems can be modelized as optimization problems in a graph.

In this document, we study both problems focusing on three aspects. The first aspect consists in the study of the complexity of these problems. The second aspect consists in the development of algorithms, exact or approximate, to solve these problems. Finally, the last aspect consists in implementing and testing these algorithms to look at their behaviors on real instances.

Keywords: Bioinformatics, Graphs, Complexity, Approximation, Algorithms

UNIVERSITÉ DE MONTPELLIER