Approches Combinatoires pour le Consensus d'Arbres et de Séquences

Sylvain Guillemot

Académie de Montpellier

Université Montpellier II

— Sciences et Techniques du Languedoc —

THESE

présentée à l'Université des Sciences et Techniques du Languedoc

Spécialité : **INFORMATIQUE**
Formation Doctorale : **Informatique**
École Doctorale : **Information, Structures, Systèmes**

Approches combinatoires pour le consensus d'arbres et de séquences

par

Sylvain Guillemot

Soutenue le 9 décembre 2008 devant le jury composé de :

M. Stéphan Thomassé, Professeur, LIRMM, Montpellier ..... Président
Mlle Marie-France Sagot, Directeur de recherches INRIA, Lyon ..... Rapporteur
M. Stéphane Vialette, Chargé de recherches CNRS, IGM, Marne-la-Vallée ..... Rapporteur
M. Alain Guénoche, Directeur de recherches CNRS, IML, Marseille ..... Examinateur
M. Olivier Gascuel, Directeur de recherches CNRS, LIRMM, Montpellier .... Dir. de Thèse
M. Vincent Berry, Maître de conférences, LIRMM, Montpellier ..... Co-encadrant

Remerciements

J'adresse mes remerciements chaleureux à Vincent Berry, mon co-directeur de thèse, dont les thèmes de recherche ont été la principale source d'inspiration de mes travaux. Je lui suis par ailleurs reconnaissant pour sa sympathie, ses conseils et son soutien constant. Je remercie également Olivier Gascuel, mon directeur de thèse, pour la liberté et la confiance qu'il m'a accordées.

Je tiens à exprimer ma reconnaissance à Michael Fellows, Marie-France Sagot et Stéphane Vialette pour avoir accepté d'évaluer ce mémoire dans des délais relativement courts, ainsi qu'à Alain Guénoche et Stéphan Thomassé pour avoir accepté d'en être les examinateurs.

J'exprime également ma gratitude à Charles Semple, pour m'avoir accueilli chaleureusement lors de séjours à Cambridge et à Napier, et pour m'avoir proposé un séjour post-doctoral en Nouvelle-Zélande.

Je remercie aussi mes coauteurs, Christophe Paul, François Nicolas et Jesper Jansson, pour les nombreuses discussions et pour les bons moments passés ensemble.

J'adresse enfin mes remerciements à mes ex-collègues de l'équipe MAB du LIRMM et de l'équipe SEQUOIA du LIFL, trop nombreux pour tous les citer. Un merci tout particulier à Samuel Blanquart, Jean-François Dufayard, et Philippe Gambette pour m'avoir donné un coup de main pour mon pot de thèse.

Table des matières

Chapitre 1

Introduction

Cette thèse est consacrée à l'étude de différents problèmes de consensus sur des collections d'objets étiquetés. Les problèmes étudiés sont motivés par des applications en bioinformatique, mais également dans d'autres domaines : en bases de données, en fouille de données, en classification, en linguistique... Les problèmes portent sur des objets étiquetés sans répétition d'étiquette ; ces objets peuvent être des arbres enracinés, ou des chaînes de caractères. En bioinformatique, les arbres enracinés sont par exemple utilisés pour représenter l'histoire évolutive d'un ensemble d'espèces biologiques. Les problèmes sur les arbres étudiés dans cette thèse ont alors comme applications la mesure de congruence entre phylogénies, la construction de superarbres et l'identification de transferts latéraux. Pour leur part, les chaînes de caractères peuvent représenter des séquences moléculaires ou des ordres de gènes. Les problèmes sur les chaînes traités dans cette thèse ont alors pour application potentielle le calcul de distances génomiques basées sur les ordres de gènes entre organismes.

On s'est efforcé de proposer des méthodes de résolution efficaces pour les problèmes étudiés, en dépit de leur NP-difficulté. La première approche passe par l'*approximation* de ces problèmes : lorsque chercher une solution exacte est NP-difficile, on peut espérer trouver une solution approchée en temps polynomial, avec une garantie d'erreur quantifiable. La seconde approche passe par la *complexité paramétrique* : face à un problème NP-difficile, on peut espérer que le problème soit soluble efficacement si l'on fixe un paramètre, dont on sait qu'il prend des valeurs faibles en pratique. Si l'on note k le paramètre, on peut ainsi espérer une complexité en $O(n^k)$ ou même en $O(2^k n)$. Dans le dernier cas, on parle d'algorithme fpt et on dit que le problème est FPT (fixed-parameter tractable). Plus généralement, un algorithme fpt est un algorithme s'exécutant en temps $O(f(k)n^c)$; un tel algorithme a donc un intérêt pratique car il reste utilisable sur des données de grande taille, pourvu que le paramètre soit faible.

Cette thèse comporte trois parties distinctes. Dans un premier temps, on se place dans le cadre de collections d'arbres étiquetés définis sur le même ensemble d'étiquettes. On considère les problèmes Sous-Arbre d'accord Maximum (Mast) et Arbre Compatible Maximum (Mct), où l'on recherche un

plus grand ensemble d'étiquettes tel que les arbres sources restreints à cet ensemble soient isomorphes, resp. aient un raffinement commun. On obtient certains résultats nouveaux pour ces problèmes bien connus et déjà très étudiés. On décrit notamment des algorithmes de 3-approximation en temps linéaire, et des algorithmes polynomiaux pour le cas d'arbres de degré borné. On présente également des résultats d'inapproximabilité et de difficulté paramétrique pour ces problèmes. Il s'agit des résultats décrits dans [1, 2, 5, 7].

Dans un second temps, on se place dans le cadre de collections d'objets étiquetés ayant des ensembles d'étiquettes distincts mais qui se chevauchent partiellement. Les objets étudiés sont les séquences et les arbres, ce qui conduit à définir les problèmes Plus longue séquence compatible (Slcs) et Superarbre d'accord maximum (Smast). Ces problèmes consistent à rechercher un plus grand ensemble d'étiquettes tel que les objets sources restreints à cet ensemble soient combinables sans répétition d'étiquettes. On présente des résultats de difficulté, ainsi que des algorithmes efficaces lorsque le nombre d'objets sources est borné : algorithmes polynomiaux basés sur la programmation dynamique, mais aussi algorithmes paramétrés et algorithmes d'approximation. Il s'agit des résultats décrits dans [6, 4].

Dans un troisième temps, on considère des objets combinatoires particuliers, les *collections de triplets enracinés*. Un triplet enraciné est un arbre binaire enraciné à trois feuilles. On étudie trois problèmes de consensus sur les collections de triplets. Le problème Nombre Maximum de Triplets Consistants (Mtc) cherche à rendre une collection de triplets compatible en conservant le nombre maximum de triplets. On présente un nouvel algorithme de 3-approximation pour le problème, ainsi qu'un résultat de NP-difficulté. Le problème Nombre Minimum d'Etiquettes Inconsistantes (Mli), resp. Nombre Minimum de Triplets Inconsistants (Mti), cherche à rendre une collection de triplets compatible en supprimant le nombre minimum d'étiquettes, resp. de triplets. On montre que, dans le cas des collections arbitraires, ces problèmes sont difficiles tant du point de vue de l'approximabilité que de la complexité paramétrique. On montre toutefois que la difficulté des problèmes est moindre dans le cas des collections complètes. Certains de ces résultats sont décrits dans [3].

Cette thèse s'organise selon le plan suivant. Le chapitre 2 introduit les notations et définitions utiles. Le chapitre 3 donne un aperçu des techniques utilisées en complexité paramétrique, pour l'obtention d'algorithmes et de résultats de difficulté. Le chapitre 4 est consacré aux problèmes de consensus sur des objets totalement étiquetés, les problèmes Mast et Mct. Le chapitre 5 est consacré aux problèmes de consensus sur des objets partiellement étiquetés, les problèmes Smast et Slcs. Le chapitre 6 porte sur les collections de triplets, et les problèmes Mtc, Mli et Mti.

1.1 Bibliographie

[1] Berry (V.), Guillemot (S.), Nicolas (F.) et Paul (C.). – Linear time 3-approximation for the MAST problem. 2007. – Accepté pour publication dans Transactions on Algorithms.

[2] Berry (V.), Guillemot (S.), Nicolas (F.) et Paul (C.). – On the approximability of the MAST and MCT problems. 2007. – Accepté pour publication dans Discrete Applied Mathematics.

[3] Byrka (J.), Guillemot (S.) et Jansson (J.). – New Results on Optimizing Rooted Triplets Consistency. *In : Proceedings of ISAAC 2008*, pp. 484–495.

[4] Guillemot (S.). – Parameterized complexity and approximability of the SLCS problem. *In : Proceedings of IWPEC 2008*, pp. 115–128.

[5] Guillemot (S.). – Simpler algorithms for approximating the MAST and MCT problems. 2008. – En préparation.

[6] Guillemot (S.) et Berry (V.). – Fixed-Parameter tractability of the Maximum Agreement Supertree problem. 2007. – Accepté pour publications dans Transactions in Computational Biology and Bioinformatics.

[7] Guillemot (S.) et Nicolas (F.). – Parameterized complexity of the MAST and MCT problems. 2008. – En préparation.

Chapitre 2

Définitions

2.1 Arbres

2.1.1 Notations

On considère dans cette thèse des arbres enracinés étiquetés aux feuilles. Soit L un ensemble d'étiquettes, un *arbre semi-étiqueté sur* L (ou simplement un *arbre sur* L) est formellement une paire $\mathcal{T} = (T, \phi)$, où T est un arbre enraciné sans noeud de degré 2 (autre que la racine), et ϕ est une bijection de l'ensemble des feuilles de T dans L.

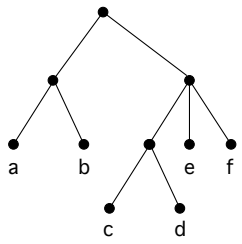


FIG. 2.1 – Un arbre semi-étiqueté sur l'ensemble d'étiquettes $L = \{a, b, c, d, e, f\}$

Par abus de notation, on identifie une étiquette avec le noeud-feuille auquel elle correspond. On note $L(T)$ l'ensemble des étiquettes (= l'ensemble des feuilles) d'un arbre T. On note $N(T)$ l'ensemble des noeuds d'un arbre. On note $|T| := |L(T)|$.

On utilise une notation parenthésée pour les arbres. Etant donnés des arbres $T_1, ..., T_n$ définis sur des ensembles disjoints, on note $(T_1, ..., T_n)$ l'arbre obtenu en ajoutant un nouveau noeud r et des arêtes joignant r à la racine de chaque arbre T_i. On définit $rake(T_1, ..., T_n)$ inductivement par : (i) $rake(T_1) = T_1$, (ii) $rake(T_1, ..., T_n) = (rake(T_1, ..., T_{n-1}), T_n)$. Si $\mathcal{F} = \{T_1, ..., T_n\}$ est une famille d'arbres définis sur des ensembles disjoints, et si $<_I$ est un ordre total tel que $1 <_I 2 <_I ... <_I n$, on note $rake(\mathcal{F}, <_I) = rake(T_1, ..., T_n)$.

Soit T un arbre sur L. On note $r(T)$ sa racine. Etant donné un noeud u de T, on note $T(u)$ le sous-arbre de T enraciné en u, et on note $L(u) := L(T(u))$. Les ensembles $L(u)$ sont appelés les *clades* de T.

Etant donnés deux noeuds u, v de T, on note $u <_T v$ (resp. $u \leq_T v$) pour signifier que v est un ancêtre (resp. ancêtre strict) de u. Etant donnés un ensemble de noeuds N de T, on note $\mathsf{lca}_T(N)$ le plus petit ancêtre commun des noeuds de N.

Etant donné un noeud u de T, on note $\mathsf{parent}_T(u)$ le père de u dans T. Etant donnés deux noeuds u, v tels que.$u <_T v$, on note $\mathsf{child}_T(u, v)$ le fils de u le long du chemin joignant u à v.

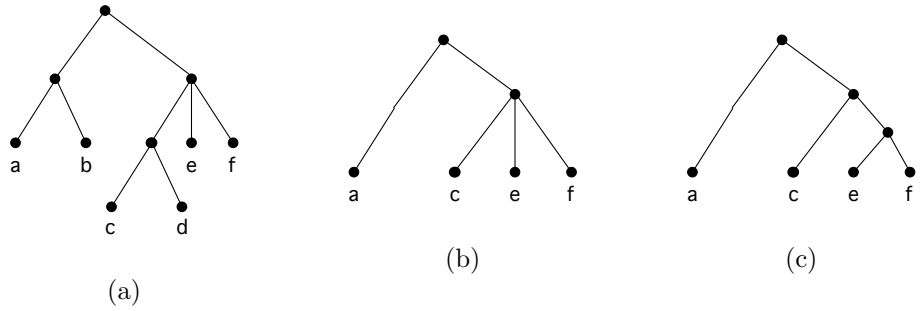
2.1.2 Notions diverses

Soient S, T deux arbres. La contraction d'une arête uv de T consiste à créer un nouveau sommet w adjacent aux voisins de u et v, et à supprimer les sommets u, v dans le graphe résultant. On dit que S raffine T ssi T peut-être obtenu à partir de S par une suite de contractions d'arêtes.

Etant donné un ensemble $L' \subseteq L$, la *restriction* de T à L' est obtenue en supprimant de T les noeuds de $L \backslash L'$, et en supprimant les noeuds de degré 2 ainsi créés.

On dit que S est un *sous-arbre* de T ssi $S = T|L(S)$. On dit que S est un *pré-sous-arbre* de T ssi S raffine $T|L(S)$. On note $S \leq T$ si S est un sous-arbre de T, et $S \trianglelefteq T$ si S est un pré-sous-arbre de T.

Ces définitions sont illustrées dans la figure suivante.



(a) (b) (c)

FIG. 2.2 – (a) Un arbre T ; (b) Un sous-arbre de T ; (c) Un pré-sous-arbre de T.

On appelle *triplet enraciné* un arbre de la forme $((x, y), z)$, qu'on abrège en $xy|z$. On appelle *fan* un arbre de la forme (x, y, z), qu'on abrège en xyz. L'ensemble des triplets (resp. fans) sous-arbres d'un arbre T est noté $rt(T)$ (resp. $f(T)$). On dit qu'un triplet $t = xy|z$ est *consistant* avec un arbre T ssi $t \in rt(T)$, c'est-à-dire si $\mathsf{lca}_T(x, y) <_T \mathsf{lca}_T(x, z) = \mathsf{lca}_T(y, z)$. On dit qu'un fan $f = xyz$ est *consistant* avec T ssi $f \in f(T)$, c'est-à-dire si $\mathsf{lca}_T(x, y) = \mathsf{lca}_T(x, z) = \mathsf{lca}_T(y, z)$.

2.2 Approximabilité

2.2.1 Problèmes d'optimisation

On définit formellement la notion de *problème d'optimisation*. Un problème d'optimisation consiste en un ensemble d'*instances*, en un ensemble de *solutions* associées à chaque instance, et en une fonction évaluant le coût d'une solution. L'objectif du problème est de rechercher une solution de coût optimal (minimal ou maximal).

Définition 2.1. *Un problème d'optimisation est un tuple* $\Pi = (I_\Pi, \mathsf{sol}_\Pi, \mathsf{m}_\Pi, \mathsf{opt})$, *où*

- I_Π *est un ensemble d'instances ;*
- *pour chaque instance* $x \in I_\Pi$, $\mathsf{sol}_\Pi(x)$ *est l'ensemble des* solutions *de* x *;*
- *pour chaque instance* $x \in I_\Pi$ *et pour chaque solution* $s \in \mathsf{sol}_\Pi(x)$, $\mathsf{m}_\Pi(x, s)$ *est le* coût *de la solution* s *pour l'instance* x *;*
- opt *est soit* max *(auquel cas* Π *est un problème de maximisation), soit* min *(auquel cas* Π *est un problème de minimisation).*

Définition 2.2. *Un problème d'optimisation* Π *est un problème de* NP*-optimisation si les conditions suivantes sont vérifiées :*

- I_Π *est reconnaissable en temps polynomial ;*
- *il existe un polynôme* P *tel que : pour tout* $x \in I_\Pi$, *chaque* $s \in \mathsf{sol}_\Pi(x)$ *est de longueur* $\leq P(|x|)$.
- *il existe un polynôme* Q *tel que : pour tout* $x \in I_\Pi, s \in \mathsf{sol}_\Pi(x)$, $\mathsf{m}_\Pi(x, s)$ *est calculable en temps* $\leq Q(|x|)$.

Soit $\Pi = (I_\Pi, \mathsf{sol}_\Pi, \mathsf{m}_\Pi, \mathsf{opt})$ un problème d'optimisation. Etant donnée une instance $x \in I_\Pi$, l'*optimum pour* x est

$$\mathsf{opt}_\Pi(x) = \mathsf{opt}_{s \in \mathsf{sol}_\Pi(x)} \mathsf{m}_\Pi(x, s)$$

Définition 2.3. *Une* solution optimale *pour* x *est une solution de coût* $\mathsf{opt}_\Pi(x)$.

Le but d'un problème d'optimisation Π consiste, étant donnée une instance $x \in I_\Pi$, à trouver une solution optimale pour x.

2.2.2 Algorithmes d'approximation

Un algorithme d'approximation pour un problème de NP-optimisation est un algorithme qui en temps polynomial calcule une solution approchée du problème, avec garantie d'erreur. Les définitions suivantes formalisent cette notion.

Définition 2.4. *Soit* Π *un problème d'optimisation, et soit* $\rho > 1$. *Une* solution ρ-approchée *pour* x *est un élément* $s \in \mathsf{sol}_\Pi(x)$ *tel que :*

- *si* Π *est un problème de minimisation :* $\mathsf{m}_\Pi(x, y) \leq \rho \times \mathsf{opt}_\Pi(x)$ *;*
- *si* Π *est un problème de maximisation :* $\mathsf{m}_\Pi(x, y) \geq \frac{\mathsf{opt}_\Pi(x)}{\rho}$.

Définition 2.5. *Soit* Π *un problème de* NP*-optimisation, et soit* $\rho : I_\Pi \to \mathbb{R}$. *Un* algorithme de ρ-approximation pour Π *est un algorithme* A *qui à chaque instance* x *de* Π *associe une solution* $\rho(x)$*-approchée pour* x*, calculée en temps polynomial.*

2.2.3 Classes et réductions

On note PTAS la classe des problèmes de NPO qui admettent un algorithme de $1+\epsilon$-approximation pour tout $\epsilon > 0$ (on parle de *schéma d'approximation polynomial*). On note APX la classe des problèmes de NPO admettant un algorithme de c-approximation pour une certaine constante $c > 1$. On peut définir une notion de problème APX*-dur* Une conséquence du théorème PCP [1] est qu'un problème APX-dur ne peut pas avoir de PTAS si P $\neq$ NP.

On introduit la notion de L-réduction qui permet d'obtenir des résultats d'APX-difficulté, et donc d'éliminer la possibilité d'un PTAS.

Définition 2.6. *Soient* Π, Π' *deux problèmes de* NPO*. Une* L-réduction *de* Π *à* Π' *est une paire de fonctions* (R, S) *calculables en temps polynomial telles que :*

1. *si* x *est une instance de* Π*, alors* $R(x)$ *est une instance de* Π' *;*
2. *si* y *est une solution de* $R(x)$*, alors* $S(x, y)$ *est une solution de* x *;*
3. *il existe une constante* $\alpha > 0$ *telle que : pour toute* x *instance de* Π*, on ait :* $\mathsf{opt}_{\Pi'}(R(x)) \leq \alpha \times \mathsf{opt}_\Pi(x)$ *;*
4. *il existe une constante* $\beta > 0$ *telle que : pour toute* x *instance de* Π*,* y *solution de* x*, on ait :* $|\mathsf{opt}_\Pi(x) - \mathsf{m}_\Pi(x, S(x, y))| \leq \beta \times |\mathsf{opt}_{\Pi'}(R(x)) - \mathsf{m}_{\Pi'}(R(x), y)|$.

Etant donnés deux problèmes Π, Π' de NPO, notons $\Pi \leq_L \Pi'$ s'il existe une L-réduction de Π à Π'.

Proposition 2.7. *Si* $\Pi \leq_L \Pi'$ *et* $\Pi' \leq_L \Pi''$ *alors* $\Pi \leq_L \Pi''$.

Démonstration. Supposons qu'on ait une L-réduction (R_1, S_1) de Π à Π' (de constantes associées α_1, β_1) et une L-réduction (R_2, S_2) de Π' à Π'' (de constantes associées α_2, β_2). Définissons la paire de fonctions (R, S) par :

$$R(x) = R_2(R_1(x))$$
$$S(x, y) = S_2(R_1(x), S_1(x, y))$$

On vérifie que (R, S) est une L-réduction de Π à Π'' de constantes associées $\alpha = \alpha_1 \times \alpha_2$ et $\beta = \beta_1 \times \beta_2$. □

Proposition 2.8. *Si* Π *admet un algorithme de* $1+\epsilon$*-approximation, et si* $\Pi' \leq_L \Pi$ *(avec les constantes* α, β*), alors* Π' *admet un algorithme de* $1 + \alpha\beta\epsilon$*-approximation.*

Démonstration. On suppose que Π, Π' sont des problèmes de minimisation, les autres cas étant analogues.

Soit A un algorithme de $1+\epsilon$-approximation pour Π, et soit (R, S) une L-réduction de Π' à Π de constantes associées α, β. Considérons l'algorithme A' suivant : étant donnée x instance de Π',

1. calculer $x' = R(x)$ instance de Π ;
2. calculer $y' = A(x')$ solution $1+\epsilon$-approchée pour x' ;
3. renvoyer $y = S(x, y')$.

On a alors :

$$\begin{aligned}
\mathsf{m}_{\Pi'}(x,y) &\leq \mathsf{opt}_{\Pi'}(x) + \beta \times (\mathsf{m}_{\Pi}(x',y') - \mathsf{opt}_{\Pi}(x')) \\
&\leq \mathsf{opt}_{\Pi'}(x) + \beta \times ((1+\epsilon)\mathsf{opt}_{\Pi}(x') - \mathsf{opt}_{\Pi}(x')) \\
&= \mathsf{opt}_{\Pi'}(x) + \beta\epsilon \times \mathsf{opt}_{\Pi}(x') \\
&\leq \mathsf{opt}_{\Pi'}(x) + \alpha\beta\epsilon \times \mathsf{opt}_{\Pi'}(x) \\
&= (1+\alpha\beta\epsilon)\mathsf{opt}_{\Pi'}(x)
\end{aligned}$$

et on conclut que y est une solution $1+\alpha\beta\epsilon$-approchée pour x. □

Corollaire 2.9. *Si* $\Pi \in \mathsf{PTAS}$ *et si* $\Pi' \leq_L \Pi$*, alors* $\Pi' \in \mathsf{PTAS}$ *; Si* Π' *est* APX*-dur et si* $\Pi' \leq_L \pi$*, alors* Π *est* APX*-dur.*

2.3 Complexité paramétrique

2.3.1 Problèmes paramétrés et algorithmes FPT

On présente les définitions centrales en complexité paramétrique : problèmes paramétrés, algorithmes fpt et classe FPT.

Définition 2.10. *Soient* Σ, Λ *deux alphabets. Un* problème paramétré *est un ensemble* $\Pi \subseteq \Sigma^* \times \Lambda^*$*. Une* instance de Π *est un couple* $(x,k) \in \Sigma^* \times \Lambda^*$ *; c'est une* instance positive *si* $(x,k) \in \Pi$*, une* instance négative *si* $(x,k) \notin \Pi$*.*

La seconde composante de l'instance, k, est appelée le *paramètre*. Pour les problèmes considérés dans cette thèse, k sera un entier ou un tuple d'entiers.

Etant donné un problème de décision $\Pi \subseteq \Sigma^*$ et une fonction $\kappa : \Sigma^* \to \Lambda^*$, le *paramétrage* de Π par κ est le problème paramétré :

$$\Pi[\kappa] = \{(x, \kappa(x)) : x \in \Pi\}$$

En complexité paramétrique, sont considérés comme faciles les problèmes admettant un algorithme fpt. Il s'agit d'un algorithme dont la partie exponentielle de la complexité ne dépend pas de la taille de l'instance mais uniquement d'un paramètre.

Définition 2.11. *Soit* Π *un problème paramétré. Un* algorithme fpt *pour* Π *est un algorithme* A *qui résout* Π *et dont le temps d'exécution sur une instance* (x, k) *est borné par* $f(k)|x|^c$*, pour un certain* $f : \Lambda^* \to \mathbb{N}$ *fonction calculable et un certain* $c \in \mathbb{N}$*.*

On note FPT la classe des problèmes paramétrés admettant un algorithme fpt.

Dans cette thèse, on sera amené à considérer des problèmes d'optimisation sous l'angle de la complexité paramétrique. Dans un souci de simplicité, on utilisera la même notation pour désigner le problème d'optimisation et le problème paramétré correspondant. Dans le cas d'un problème de maximisation, on notera q le paramètre correspondant. Dans le cas d'un problème de minimisation, on notera p le paramètre correspondant.

2.3.2 Classes et réductions

On montre qu'un problème est dans FPT en exhibant un algorithme fpt. Dans le cas de problèmes extérieurs à FPT, une technique standard pour établir l'absence d'algorithmes fpt repose sur la définition de classes de complexité paramétrique, et sur une notion de réduction appropriée, la fpt-réduction.

Définition 2.12. *Soient* Π, Π' *deux problèmes paramétrés. Une* fpt-réduction *de* Π *à* Π' *est un algorithme* A *qui à chaque instance* $I = (x, k)$ *de* Π *associe une instance* $I' = (x', k')$ *de* Π'*, telle que*

- *I est une instance positive de Π ssi I' est une instance positive de Π' ;*
- *$k' \leq g(k)$ pour une certaine fonction croissante $g : \Lambda^* \to \Lambda^*$;*
- *A s'exécute en temps $f(k)|x|^c$ pour une certaine fonction croissante $f : \Lambda^* \to \mathbb{N}$ et pour un certain $c \in \mathbb{N}$.*

Etant donnés deux problèmes paramétrés Π, Π', notons $\Pi \leq_{fpt} \Pi'$ s'il existe une fpt-réduction de Π à Π'.

Cette notion de réduction préserve l'appartenance à FPT :

Proposition 2.13. *Si* $\Pi \in$ FPT *et* $\Pi' \leq_{fpt} \Pi$ *alors* $\Pi' \in$ FPT*.*

Proposition 2.14. *Si* $\Pi \leq_{fpt} \Pi'$ *et* $\Pi' \leq_{fpt} \Pi''$ *alors* $\Pi \leq_{fpt} \Pi''$*.*

La théorie définit un ensemble de classes de complexité paramétrique qui sont conjecturées différentes de FPT.

$$\mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] ... \subseteq \mathsf{W}[\mathsf{SAT}] \subseteq \mathsf{W}[\mathsf{P}]$$

Il est conjecturé que les inclusions ci-dessus sont strictes. Etablir qu'un problème est dur pour l'une de ces classes (au moyen d'une fpt-réduction) élimine donc la possibilité d'un algorithme fpt pour le problème.

On va s'intéresser principalement à deux classes de complexité : la classe W[1], ainsi qu'une classe WNL que nous introduisons. Ces deux classes sont définies formellement dans ce qui suit.

La classe W[1] a été définie par [5] en termes de problèmes sur des circuits booléens. On adopte ici une définition alternative en termes de problèmes sur des machines de Turing. Cette caractérisation équivalente de la classe W[1] a été présentée dans [2] et utilisée notamment dans [3].

On considère le problème suivant :

Nom : Nondeterministic Turing Machine Computation (Ntmc)
Instance : une machine de Turing non-déterministe M, un entier k, un entier q en unaire
Question : est-ce que M accepte en q pas et en visitant $\leq k$ cellules du ruban ?

Ce problème sert de problème canonique pour la définition des classes W[1] et WNL. Etant donné un problème paramétré Π, on note $[\Pi]^{fpt}$ l'ensemble des problèmes paramétrés Π' tels que $\Pi' \leq_{fpt} \Pi$. On pose alors :

Définition 2.15. $\mathsf{W}[1] = [\textsc{Ntmc}[k,q]]^{fpt}$, $\mathsf{WNL} = [\textsc{Ntmc}[k]]^{fpt}$.

Notons qu'on a également $\mathsf{W}[1] = [\textsc{Ntmc}[q]]^{fpt}$: en effet, les problèmes Ntmc$[q]$ et Ntmc$[k,q]$ sont équivalents, du fait que le temps d'exécution est toujours supérieur ou égal au nombre de cellules visitées. Le problème Ntmc$[q]$ est généralement appelé Short Nondeterministic Turing Machine Computation ou aussi k-Halting.

2.3.3 Résultats d'optimalité

Une preuve de W[1]-difficulté élimine la possibilité d'un algorithme FPT, de temps d'exécution $\Phi(k)n^c$, mais n'exclut pas l'éventualité d'algorithmes en temps $n^{O(\sqrt{k})}$ ou $n^{O(\log k)}$. Des outils existent pour obtenir de tels résultats négatifs, éliminant en fait la possibilité d'un algorithme de temps d'exécution $\Phi(k)n^{o(k)}$, et prouvant ainsi l'optimalité d'un algorithme en $n^{O(k)}$. Ils reposent sur la notion de fpt-réduction linéaire et sur une conjecture baptisée *Exponential Time Hypothesis* (ETH), plus forte que l'hypothèse $\mathsf{FPT} \neq \mathsf{W}[1]$. Ces outils ont été introduits par [4].

Définition 2.16. *Une fpt-réduction est linéaire ssi la fonction g est telle que $g(k) = O(k)$.*

Définition 2.17 (Hypothèse ETH, [6]). *Le problème* 3SAT *à n variables n'est pas soluble en temps $2^{o(n)}$.*

On utilisera une classe de complexité $\mathsf{W_I}[1]$ qu'on se dispense de définir formellement. Simplement, on utilise une notion de problème $\mathsf{W_I}[1]$-dur, et on transfère des résultats de $\mathsf{W_I}[1]$-difficulté à l'aide de fpt-réductions linéaires.

Proposition 2.18 ([4]). *Si Π est $\mathsf{W_I}[1]$-dur, alors Π n'est pas soluble en temps $\Phi(k)|x|^{o(k)}$, sous l'hypothèse ETH.*

2.4 Bibliographie

[1] Arora (S.) et Safra (S.). – Probabilistic Checking of Proofs : A New Characterization of NP. *Journal of the Association for Computing Machinery*, vol. 45, n° 3, 1998, pp. 501–555.

[2] Cai (L.), Chen (J.), Downey (R.G.) et Fellows (M.R.). – On the parameterized complexity of short computation and factorization. *Archive for Mathematical Logic*, vol. 36, n° 4–5, 1997, pp. 321–337.

[3] Cesati (M.). – The Turing way to parameterized complexity. *Journal of Computer and System Sciences*, vol. 67, n° 4, 2003, pp. 654–685.

[4] Chen (J.), Huang (X.), Kanj (I.A.) et Xia (G.). – On the computational hardness based on linear FPT-reductions. *Journal of Combinatorial Optimization*, vol. 11, n° 2, 2006, pp. 231–247.

[5] Downey (R.G.) et Fellows (M.R.). – Fixed-Parameter Tractability and Completeness II : On Completeness for W[1]. *Theoretical Computer Science*, vol. 141, n° 1–2, 1995, pp. 109–131.

[6] Impagliazzo (R.), Paturi (R.) et Zane (F.). – Which Problems Have Strongly Exponential Complexity ? *Journal of Computer and System Sciences*, vol. 63, n° 4, 2001, pp. 512–530.

Chapitre 3

Techniques en complexité paramétrique

L'objectif de ce chapitre est d'illustrer différentes techniques mises en oeuvre en complexité paramétrique, tant pour l'obtention de résultats négatifs (Section 3.1) que pour l'obtention d'algorithmes FPT (Section 3.2). Ces techniques sont illustrées à travers des problèmes combinatoires classiques, notamment des problèmes portant sur les graphes et les hypergraphes. On pourra se référer aux monographies [11, 14, 22] pour une présentation plus détaillée de certaines techniques.

3.1 Résultats de difficulté

Comme on l'a vu en Section 2.3, la théorie de la complexité paramétrique fournit des outils pour établir qu'un problème paramétré n'admet pas d'algorithme FPT. Il faut pour cela montrer que le problème est complet pour une classe de complexité paramétrique, au moyen d'une réduction paramétrée (ou fpt-réduction). On va présenter cette méthodologie, à travers des exemples de réductions pour divers problèmes combinatoires. On introduit d'abord des problèmes complets canoniques pour les classes W[1] et WNL, alternatifs au problème NTMC. A l'aide de ces problèmes, on établit ensuite la W[1]-complétude de divers problèmes sur les graphes.

3.1.1 Problème complets canoniques pour W[1] et WNL

Le premier problème, baptisé EXISTENTIAL PEBBLE GAME, porte sur des *jeux de jetons existentiels*. Un *jeu de jetons* à k jetons est un tuple $G = (V, T, R, I, F)$, où V est un ensemble de sommets, T est un ensemble de transitions, $R = \{\rightarrow_t : t \in T\}$ est un ensemble de relations binaires sur V, $I \in V^k$ est la configuration initiale et $F \in V^k$ est la configuration finale. Une *configuration* de G est un tuple $C \in V^k$, où $C[i]$ est la position du $i$ème jeton.

Etant données deux configurations C, C' de G et une transition $t \in T$, on note $C \rightarrow_t C'$ si et seulement si $C[i] \rightarrow_t C'[i]$ pour tout $i \in [k]$. Une *partie* de G est une chaîne de configurations $C_0 \rightarrow_{t_1} C_1 \rightarrow_{t_2} ... \rightarrow_{t_q} C_q$ telle que $C_0 = I, C_q = F$. La longueur de la partie est q.

On considère le problème suivant :

Nom : Existential Pebble Game (Epg)
Instance : des entiers k, q, un jeu de jetons à k jetons G
Question : est-ce que G admet une partie de longueur q ?

On montre l'équivalence de ce problème avec le problème Ntmc. Introduisons d'abord les notations suivantes. Etant donnés deux problèmes biparamétrés Π, Π', une *sfpt-réduction* de Π à Π' est une fpt-réduction qui transforme une instance (I, k, q) de Π en une instance $(I', f(k), g(k, q))$ de Π', i.e. le premier paramètre est « préservé ». On note $\Pi \leq_{sfpt} \Pi'$ pour indiquer l'existence d'une sfpt-réduction de Π à Π'.

Proposition 3.1. *(i)* $\text{Ntmc}[k, q] \leq_{sfpt} \text{Epg}[k, q]$*, (ii)* $\text{Epg}[k, q] \leq_{sfpt} \text{Ntmc}[k, q]$.

Démonstration. Point (i). Soit $I = (M, k, q)$ une instance de $\text{Ntmc}[k, q]$, avec $M = (\Sigma, Q, \perp, q_i, q_f, \Delta)$. On construit une instance $I' = (G, k', q')$ de $\text{Epg}[k, q]$ de la façon suivante.

On pose $q' = 4q$ et $k' = k + 1$. On définit $G = (V, T, R, I, F)$ comme suit. On pose $V = V_0 \cup V_1 \cup ... \cup V_k$, où :

- V_0 contient un ensemble de paires $(0, s)$, où s est un terme qui représente l'état courant de la machine, et peut prendre les valeurs suivantes : (i) $s = idleStep(q, i)$ avec $q \in Q, i \in [k]$; (ii) $s = readStep(q, i, v)$ avec $q \in Q, i \in [k], v \in \Sigma$; (iii) $s = writeStep(q, i, j, v)$ avec $q \in Q, i, j \in [k], v \in \Sigma$.
- pour chaque $i \in [k]$, V_i contient l'ensemble des paires (i, x) avec $x \in \Sigma$.

Une configuration de G sera un tuple $C = (v_0, v_1, ..., v_k)$ avec $v_i \in V_i$. La configuration initiale est $I = ((0, idleStep(q_i, 1)), (1, \perp), ..., (k, \perp))$, la configuration finale est $F = ((0, idleStep(q_f, 1)), (1, \perp), ..., (k, \perp))$.

Les transitions de T ont une forme particulière. Chaque transition peut effectuer le déplacement d'un seul jeton i de x à y, conditionné par la présence du jeton j en z. Une telle transition sera notée $t = (i, j, x, y, z)$. La relation $\rightarrow_t$ associée sera donc définie de la façon suivante : (i) $(i, x) \rightarrow_t (i, y)$, (ii) $(j, z) \rightarrow_t (j, z)$, (iii) pour tout $p \neq i, j$, pour tout $v \in V_p$, $v \rightarrow_t v$.

On va maintenant définir les transitions de T. L'intuition est qu'un pas de M en position i du ruban est décomposé en quatre étapes : (i) lecture de la lettre a en position i qui provoque le passage du jeton 0 de $(0, idleStep(q, i))$ à $(0, readStep(q, i, a))$, (ii) choix d'une transition $q \rightarrow_{a/b,d} q'$ de M, qui provoque le passage du jeton 0 dans l'état $(0, writeStep(q', i, i', b))$ avec $i' = s(i, d)$, (iii) écriture de la lettre b en position i qui provoque le passage du jeton i de (i, a) à (i, b), (iv) une fois que l'écriture a été effectuée, passage du jeton 0 dans l'état $(0, idleStep(q', i'))$. Les transitions de T sont donc les suivantes :

- $(0, i, idleStep(q, i), readStep(q, i, a), a)$ pour $q \in Q, i \in [k], a \in \Sigma$.

- $(0, i, readStep(q, i, a), writeStep(q', i, i', b), a)$ pour $q \in Q, i \in [k], a \in \Sigma$, et pour une transition $q \rightarrow^{a/b,d} q'$ de M, avec $i' = s(i, d)$.
- $(i, 0, a, b, writeStep(q', i, i', b))$ pour $q' \in Q, i, i' \in [k], a, b \in \Sigma$.
- $(0, i, writeStep(q', i, i', b), idleStep(q', i'), b)$ pour $q' \in Q, i, i' \in [k], b \in \Sigma$.

On vérifie que la réduction est polynomiale, et que : M accepte en q pas en utilisant un espace $\leq k$ si et seulement si G a une partie de longueur q'.

Point (ii). Soit $I = (G, k, q)$ une instance de Epg$[k, q]$, où $G = (V, T, R, I, F)$ est un jeu de jetons à k jetons. On construit une instance $I' = (M, k', q')$ de Ntmc$[k, q]$ où $k' = 2k$, $q' = O(k^2 q)$, et M est une machine de Turing qui procède comme suit. Les k premières cellules de son ruban mémorisent une configuration C de G. M effectue q rounds ; chaque round simule une transition de G, et consiste à : (i) écrire de façon non déterministe une configuration C' de G dans les cellules $k+1$ à $2k$, (ii) vérifier que $C \rightarrow_t C'$, (iii) remplacer C par C'. Clairement, chaque round nécessite $O(k^2)$ pas. On obtient donc que : G a une partie de longueur q si et seulement si M accepte en temps $q' = O(k^2 q)$ et en utilisant un espace $k' = 2k$. □

On déduit du résultat précédent des résultats de complétude pour W[1] et WNL.

Proposition 3.2. *(i)* Epg$[k, q]$ *est* W[1]*-complet ; (ii)* Epg$[k]$ *est* WNL*-complet.*

On présente maintenant un second problème canonique. Il s'agit d'un problème d'étiquetage sur une grille, appelé Grid Labeling. A chaque sommet de la grille est associé un ensemble de valeurs possibles, à chaque arête est associée une contrainte sur les valeurs aux extrémités, et on veut choisir une valeur pour chaque sommet de façon à satisfaire toutes les contraintes.

La définition formelle du problème est la suivante :

Nom : Grid Labeling (Gl)
Instance : une $k \times q$-grille $G = (V, E)$, un ensemble S, une partition $P = \{S_v : v \in V\}$ de S, pour chaque arête $e = uv \in E$ une relation d'équivalence R_e sur $S_u \cup S_v$.
Question : existe t-il un étiquetage de chaque $v \in V$ par une valeur $l_v \in S_v$, tel que : pour tout $e = uv \in E$, $(l_u, l_v) \in R_e$? Un tel étiquetage est appelé un *étiquetage admissible* de G.

Proposition 3.3. *(i)* Epg$[k, q] \leq_{sfpt}$ Gl$[k, q]$*, (ii)* Gl$[k, q] \leq_{sfpt}$ Ntmc$[k, q]$.

Démonstration. Point (i) : soit $I = (G, k, q)$ instance de Epg$[k, q]$, où $G = (V, T, R, I, F)$, avec V ensemble de sommets, T ensemble de transitions, $R = \{\rightarrow_t : t \in T\}$, $I \in V^k$ configuration initiale, $F \in V^k$ configuration finale. On construit une instance I' de Gl$[k, q]$ comme suit. I' consiste en :

- une $k \times q$-grille H, dont le sommet en ligne i, colonne j est noté $v_{i,j}$;

- un ensemble $S = \{M_{i,t,j,x,y} : i \in [q], t \in T, j \in [k], x, y \in V$ tels que $x \rightarrow_t y\}$. Pour $i = 1$, seules les valeurs $M_{i,t,j,x,y}$ avec $x = I[j]$ sont présentes. Pour $i = q$, seules les valeurs $M_{i,t,j,x,y}$ avec $y = F[j]$ sont présentes.
- la partition P de S est définie par : pour tout $i \in [q], j \in [k]$, $S_{v_{i,j}} = \{M_{i',t,j',x,y} \in S : i' = i, j' = j\}$.
- pour chaque arête e de G, une relation d'équivalence R_e définie comme suit. Si $e = v_{i,j}v_{i,j+1}$, alors $M_{i,t,j,x,y}, M_{i,t',j+1,x',y'}$ sont équivalents par R_e si et seulement si $t = t'$. Si $e = v_{i,j}v_{i+1,j}$, alors $M_{i,t,j,x,y}, M_{i+1,t',j,x',y'}$ sont équivalents par R_e si et seulement si $y = x'$.

La réduction est clairement polynomiale, et on vérifie que : I est une instance positive de Epg si et seulement si I' est une instance positive de Gl.

Point (ii) : soit I une instance de Gl$[k, q]$, consistant en une $k \times q$-grille $G = (V, E)$, un ensemble S, une partition $P = \{S_v : v \in V\}$ de S, et pour chaque $e = uv \in E$ une relation d'équivalence R_e sur $S_u \cup S_v$. Pour tout $1 \leq i \leq q, 1 \leq j \leq k$, notons $v_{i,j}$ le sommet de G en ligne i, colonne j. Pour tout $1 \leq i \leq q, 1 \leq j < k$, notons $e_{i,j}$ l'arête horizontale joignant $v_{i,j}$ à $v_{i,j+1}$; pour tout $1 < i \leq q, 1 \leq j \leq k$, notons $e'_{i,j}$ l'arête verticale joignant $v_{i-1,j}$ à $v_{i,j}$.

On définit une instance $I' = (M, k', q')$ de Ntmc$[k, q]$, où $k' = 2k, q' = O(k^2q)$, et M est une machine de Turing qui effectue les opérations suivantes. Elle effectue q rounds, et à la fin du round i les k premières cellules de son ruban contiennent les valeurs des k sommets de la ligne i. Le round 1 consiste à écrire de façon non-déterministe k valeurs $x_1, ..., x_k$ dans les k premières cellules, à vérifier que : (i) pour tout $j \in [k]$, $x_j \in S_{v_{1,j}}$, (ii) pour tout $1 \leq j < k$, $(x_j, x_{j+1}) \in R_{e_{1,j}}$.

Pour tout $i > 1$, le round i procède comme suit. Soit $x_1, ..., x_k$ les valeurs des k premières cellules au début du round i, alors M écrit de façon non-déterministe k valeurs $x'_1, ..., x'_k$ dans les k cellules suivantes. Ensuite, elle vérifie que : (i) pour tout $j \in [k]$, $x'_j \in S_{v_{i,j}}$, (ii) pour tout $j \in [k]$, $(x_j, x'_j) \in R_{e'_{i,j}}$, (iii) pour tout $1 \leq j < k$, $(x'_j, x'_{j+1}) \in R_{e_{i,j}}$. Finalement, elle écrit les valeurs $x'_1, ..., x'_k$ en tête du ruban, et passe au round $i + 1$.

Clairement, chaque round nécessite $O(k^2)$ pas, et on conclut qu'il existe un étiquetage admissible de G si et seulement si M accepte en temps $q' = O(k^2q)$ et en utilisant un espace $k' = 2k$. □

On déduit des résultats de complétude pour W[1] est WNL :

Proposition 3.4. *(i)* Gl$[k, q]$ *est* W[1]*-complet ; (ii)* Gl$[k]$ *est* WNL*-complet.*

3.1.2 Quelques autres résultats de complétude pour W[1]

On montre d'abord la W[1]-complétude du problème Clique défini comme suit.

Nom : Clique
Instance : un graphe $G = (V, E)$, un entier k

Paramètre : k
Question : est-ce que G contient une clique de cardinal k ?

Proposition 3.5. Clique *est* W[1]*-complet.*

Démonstration. Appartenance à W[1] : par réduction à Ntmc$[k, q]$. Etant donnée $I = (G, k)$ instance de Clique avec $G = (V, E)$, on construit une instance $I' = (M, k', q')$ de Ntmc$[k, q]$, où $k' = k, q' = O(k^2)$, et M est une machine de Turing qui effectue les opérations suivantes. Au cours d'une première étape, elle écrit sur le ruban k éléments $v_1, ..., v_k \in V$. Au cours d'une seconde étape, elle vérifie que pour tout $i, j \in [k]$ $(i < j)$, $v_i v_j \in E$.

W[1]-difficulté : par réduction depuis Gl$[k, q]$. Soit I une instance du problème, consistant en une $k \times q$-grille $G = (V, E)$, en un ensemble S, en une partition $P = \{S_v : v \in V\}$ de S, et pour chaque $e = uv \in E$ en une relation d'équivalence R_e sur $S_u \cup S_v$. On construit une instance $I' = (G, k')$ de Clique, où $k' = kq$, et où G est un graphe k'-parti d'ensemble de sommets S et de partition P. Disons que deux éléments x, y $(x \in S_u, y \in S_v)$ sont *compatibles* si et seulement si $u \neq v$ et : (i) soit $uv \notin E$, (ii) soit $uv \in E$ et $(x, y) \in R_e$. Alors G contient l'arête xy si et seulement si x, y sont compatibles. La réduction est polynomiale, et on vérifie que : I est une instance positive de Grid Labeling si et seulement si I' est une instance positive de Clique. □

Observons que la preuve précédente montre également la W[1]-complétude du problème Partitioned Clique défini comme suit :

Nom : Partitioned Clique
Instance : un entier k, un graphe k-parti G de partition $V_1, ..., V_k$
Paramètre : k
Question : est-ce que G contient une clique C telle que $|C \cap V_i| = 1$ pour tout i ?

Une conséquence immédiate des résultats précédents est la W[1]-difficulté des problèmes Independent Set et Partitioned Independent Set, définis de façon analogue.

On présente maintenant un résultat de W[1]-complétude pour le problème Perfect Code. Soit un graphe $G = (V, E)$. Un *code parfait* de G est un ensemble $V' \subseteq V$ tel que pour chaque $v \in V$, $|N_G^+(v) \cap V'| = 1$. De manière équivalente, V' est un code parfait si et seulement si les ensembles $N_G^+(v)$ $(v \in V')$ forment une partition de V.

Le problème est défini comme suit :

Nom : Perfect Code
Instance : un graphe $G = (V, E)$, un entier k
Paramètre : k

Question : est-ce que G a un code parfait de cardinal k ?

Cette définition est illustrée dans la figure suivante :

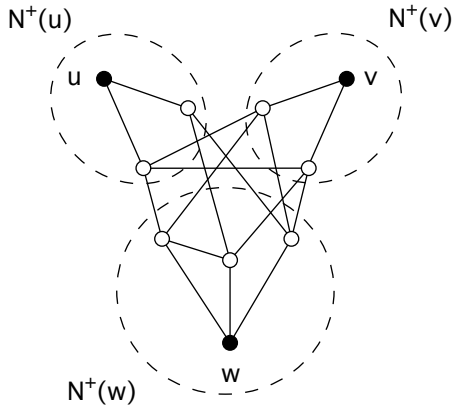


FIG. 3.1 – Un graphe G, un code parfait de G formé des trois sommets u, v, w. On vérifie que les ensembles $N_G^+(u), N_G^+(v), N_G^+(w)$ forment une partition de l'ensemble des sommets de G.

Proposition 3.6. PERFECT CODE *est* W[1]*-complet.*

Démonstration. Appartenance à W[1] : par réduction à NTMC$[k, q]$. Etant donnée $I = (G, k)$ instance de PERFECT CODE avec $G = (V, E)$, on construit une instance $I' = (M, k', q')$ de NTMC$[k, q]$, où $k' = k$, $q' = O(k^2)$, et M est une machine de Turing qui fait les opérations suivantes. Au cours d'une première étape, elle écrit sur le ruban k éléments $v_1, ..., v_k \in V$. Au cours d'une seconde étape, elle vérifie que pour tout $i, j \in [k]$ $(i < j)$,

- v_i, v_j sont distincts ;
- v_i, v_j sont non adjacents dans G ;
- v_i, v_j n'ont pas de voisin commun.

A l'issue de cette deuxième étape, on sait que les ensembles $N_G^+(v_i)$ sont deux à deux disjoints. Pour s'assurer qu'ils couvrent V, on vérifie lors d'une troisième étape que $\sum_i |N_G^+(v_i)| = n$ (où $n = |V|$). Ceci est réalisé en précalculant pour chaque $v \in V$ l'entier $n_v = |N_G^+(v)|$, et en ajoutant à la machine un gadget qui lit les k éléments du ruban et accumule les valeurs n_{v_i}, l'information étant stockée dans l'état courant. La machine accepte alors si et seulement si la somme obtenue est égale à n.

W[1]-difficulté : par réduction depuis GL$[k, q]$. Soit I une instance du problème, consistant en une $k \times q$-grille $G = (V, E)$, en un ensemble S, en une partition $P = \{S_v : v \in V\}$ de S, et pour chaque $e = uv \in E$ en une relation d'équivalence R_e sur $S_u \cup S_v$. Soit $<$ un ordre total sur V. On construit une instance $I' = (G, k')$ de PERFECT CODE, où $k' = kq$. L'ensemble de sommets

de G est formé d'un élément a_v (pour chaque $v \in V$), d'un élément b_x (pour chaque $x \in S$) et d'éléments $c_{e,C}, d_{e,C}$ (pour chaque $e \in E$, C classe de R_e). G contient les arêtes suivantes :

- pour chaque $v \in V$, des arêtes formant une clique sur l'ensemble de sommets $\{a_v\} \cup \{b_x : x \in S_v\}$;
- pour chaque $e = uv \in E$ $(u < v)$, C classe de R_e : $c_{e,C}$ est adjacent à l'ensemble de sommets b_x pour $x \in (V_u \cap C) \cup (V_v \backslash C)$;
- pour chaque $e = uv \in E$ $(u < v)$, C classe de R_e : $d_{e,C}$ est adjacent à l'ensemble de sommets b_x pour $x \in (V_v \cap C) \cup (V_u \backslash C)$.

On vérifie que la réduction est polynomiale, et que : I est une instance positive de Grid Labeling si et seulement si I' est une instance positive de Perfect Code. □

On montre un dernier résultat de W[1]-complétude, pour le problème Colored Grid Embedding.

Un *graphe c-coloré* est un graphe dont les sommets sont colorés avec c couleurs (éléments de $1, ..., c$). Soient deux graphes c-colorés G, H. Soit $V = V(G)$ et $V_1, ..., V_c$ les classes de couleur de G. Soit $V' = V(H)$ et $V'_1, ..., V'_c$ les classes de couleur de H. Un *plongement coloré de G dans H* est une fonction $\phi : V(G) \to V(H)$ telle que (i) ϕ est injective, (ii) $uv \in E(G)$ implique $\phi(u)\phi(v) \in E(H)$, (iii) pour tout $i \in [c]$, $\phi(V_i) \subseteq V'_i$.

On considère le problème suivant :

Nom : c-Colored Grid Embedding
Instance : une $k \times q$-grille c-colorée G, un graphe c-coloré H, un entier k
Paramètre : k, q
Question : existe t-il un plongement coloré de G dans H ?

Cette définition est illustrée dans la figure suivante :

Proposition 3.7. *c-Colored Grid Embedding est W[1]-complet, pour tout $c \geq 2$.*

Démonstration. L'appartenance à W[1] se montre facilement par réduction à Ntmc$[k, q]$.

W[1]-difficulté : par réduction depuis la restriction de Gl$[k, q]$ aux instances (I, k, q) telles que $k = q$. Notons que cette restriction du problème reste W[1]-difficile. Soit I une instance du problème, consistant en une $k \times k$-grille $G = (V, E)$, en un ensemble S, en une partition $P = \{S_v : v \in V\}$ de S, et pour tout $e = uv \in E$ en une relation d'équivalence R_e sur $S_u \cup S_v$.

On construit une instance $I' = (G', H', k', k')$ du problème, où $k' = 3k$, et G', H' sont les graphes suivants. G' est une $k' \times k'$-grille 2-colorée, composée de blocs 3x3 $B_{i,j}$ $(i, j \in [k])$ dont le sommet central est de couleur 1 et les autres sommets sont de couleur 0. H' est défini comme suit.

Etant donnée une arête $e = uv$ de G, une *assignation* de e est une fonction ϕ qui à u associe à u une valeur $\phi(u) \in S_u$ et à v une valeur $\phi(v) \in S_v$, telle que $(\phi(u), \phi(v)) \in R_e$. Etant donnée une face f de G, une *assignation* de f est

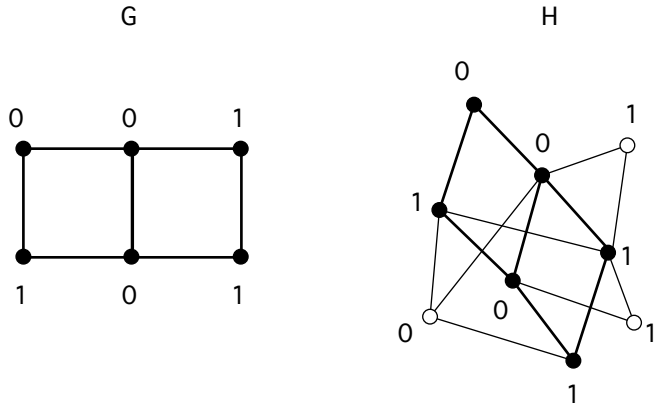


FIG. 3.2 – Une 3x2 grille 2-colorée G, un graphe 2-coloré H, un plongement coloré de G dans H. Le plongement est matérialisé en représentant en noir les sommets impliqués et en trait gras les arêtes impliquées.

une fonction ϕ qui à chaque sommet u de f associe une valeur $\phi(u) \in S_u$, telle que pour toute arête $e = uv$ de f, $(\phi(u), \phi(v)) \in R_e$.

H' contient les sommets suivants : pour chaque face f de G, (i) pour chaque sommet u de f, pour chaque $x \in S_u$, un sommet $a_{f,u,x}$; (ii) pour chaque e de f, pour chaque ϕ assignation de e, un sommet $b_{f,e,\phi}$; (iii) pour chaque assignation ϕ de f, un sommet $c_{f,\phi}$. Les sommets de H' sont coloriés de la façon suivante : les sommets $a_{f,u,x}, b_{f,e,\phi}$ ont la couleur 0, et les sommets $c_{f,\phi}$ ont la couleur 1.

H' contient les arêtes suivantes : (i) pour chaque face f de G, pour chaque assignation ϕ de f, pour chaque arête $e = uv$ de f, une arête $c_{f,\phi}b_{f,e,\phi|e}$; (ii) pour chaque face f de G, pour chaque arête $e = uv$ de f, pour chaque assignation ϕ de e, une arête $b_{f,e,\phi}a_{f,u,\phi(u)}$; (iii) pour chaque arête e de G, pour chaque assignation ϕ de e : si f, f' sont les deux faces de G contenant e, une arête $b_{f,e,\phi}b_{f',e,\phi}$; (iv) pour chaque sommet u de G, pour chaque $x \in S_u$: si f, f' sont deux faces adjacentes de G contenant u, une arête $a_{f,u,x}a_{f',u,x}$.

Il est clair que la réduction est polynomiale, et sa correction résulte du fait que : I est une instance positive de GRID LABELING si et seulement si I' est une instance positive de c-COLORED GRID EMBEDDING. □

3.1.3 Historique et contribution

On a introduit des nouveaux problèmes canoniques pour W[1] et WNL, les problèmes EXISTENTIAL PEBBLE GAME et GRID LABELING. Le problème GRID LABELING nous a ensuite permis d'obtenir des preuves de W[1]-difficulté simples pour plusieurs problèmes. Bien que les résultats de complétude présentés pour ces problèmes étaient déjà connus, l'utilisation du problème GRID LABELING fournit des réductions plus simples.

La W[1]-difficulté de CLIQUE a été établie dans [10] en utilisant la définition

originelle de W[1] en termes de circuits. La W[1]-difficulté de Perfect Code a été établie dans [10] par réduction depuis Clique. Le problème c-Colored Grid Embedding ne semble pas avoir été étudié, en revanche sa variante c-Colored Grid Homomorphism (où l'on n'exige plus l'injectivité de la fonction) a été étudiée dans [16], et montrée W[1]-dur pour tout $c \geq 2$.

3.2 Techniques pour l'obtention d'algorithmes FPT

Il n'existe pas de méthode générale pour la conception d'algorithmes FPT : l'obtention d'un algorithme FPT pour un problème nécessite d'identifier une propriété structurelle du problème, qui peut être difficile à mettre en évidence. On peut toutefois identifier plusieurs techniques standard pour la conception d'algorithmes FPT : la recherche bornée, la kernelisation, le color-coding, la compression itérative, ainsi que d'autres techniques telles que les décompositions arborescentes. Cette section est consacrée à une présentation de ces différentes techniques.

3.2.1 Recherche bornée et kernelisation

La méthode de *recherche bornée* pour un problème paramétré Π consiste à construire un arbre de recherche dont la taille est bornée par une fonction du paramètre. Typiquement, le degré de l'arbre est une constante C, la hauteur de l'arbre est bornée par k, et chaque noeud de l'arbre est traité en temps $O(n^c)$, ce qui aboutit à un algorithme de temps $O(C^k n^c)$ pour le problème.

Une *kernelisation* pour un problème paramétré Π est un algorithme qui transforme une instance en une instance équivalente de taille bornée. Autrement dit, c'est un algorithme qui à partir d'une instance $I = (x, k)$ de Π, produit une instance équivalente $I' = (x', k')$ avec $k' \leq k$ et $|x'| \leq f(k)$. Alors que la recherche bornée conduit à des algorithmes de complexité $O(f(k)n^c)$, la kernelisation permet d'obtenir une complexité $O(n^c + f(k))$.

On va illustrer ces deux techniques sur le problème suivant :

Nom : Hitting Set (HS)
Instance : un hypergraphe H, un entier p
Paramètre : p
Question : H admet-il un transversal de cardinal $\leq p$?

On note k-HS la restriction de HS aux hypergraphes k-uniformes. On va présenter un algorithme de recherche bornée pour le problème (Proposition 3.8), un algorithme de kernelisation (Proposition 3.10), et présenter finalement la technique d'*entrelacement* consistant à combiner recherche bornée et kernelisation (Proposition 3.11).

Proposition 3.8. *k-HS est soluble en temps $O(k^p|H|)$.*

Démonstration. Soit $I = (H, p)$ une instance de k-HS, avec $H = (V, E)$. L'algorithme construit un arbre T, dont chaque noeud est étiqueté par une paire (H', p') avec H' hypergraphe et $p' \leq p$. Initialement T est réduit à une feuille étiquetée par (H, p). L'algorithme construit progressivement T en répétant la règle suivante :

tant que T comporte une feuille u étiquetée par (H', p') avec H' non vide et $p' > 0$: on trouve alors e arête de H', et pour chaque $x \in e$ on ajoute à u un fils u_x d'étiquette $(H' \backslash \{x\}, p' - 1)$.

Lorsque l'algorithme précédent termine, on peut marquer par Succès les feuilles de T dont l'hypergraphe associé est vide, et par Echec les autres feuilles. On a alors une correspondance entre : (i) les transversaux minimaux de H de cardinal $\leq p$, (ii) les feuilles de T marquées Succès.

Analysons le temps d'exécution de l'algorithme. L'arbre T construit par l'algorithme est de hauteur $\leq p$ et de degré $\leq k$, donc de taille $\leq k^p$. Chaque noeud interne de l'arbre étant traité en temps $O(|H|)$, l'algorithme a donc le temps d'exécution annoncé $O(k^p|H|)$. □

On représente en Figure 3.3 un graphe instance de 2-HS, et un arbre de recherche obtenu en appliquant l'algorithme précédent à cette instance.

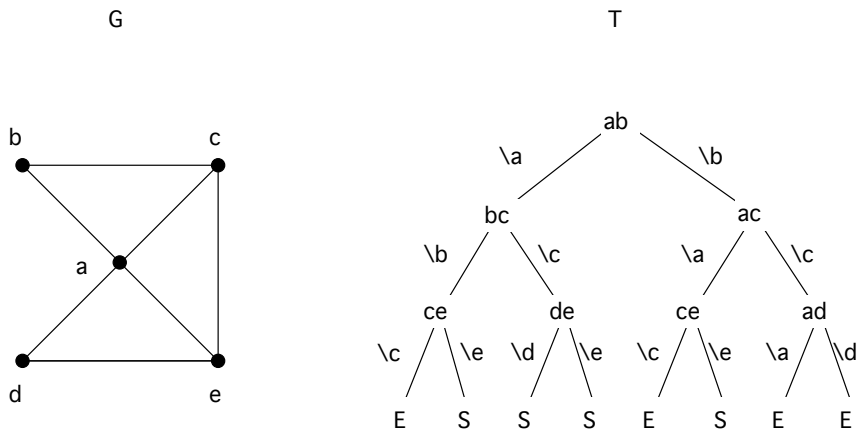


FIG. 3.3 – Un exemple d'exécution de l'algorithme de recherche borné pour 2-HS sur l'instance $I = (G, p)$ avec G représenté à gauche et $p = 3$. On a représenté l'arbre de recherche T. On a indiqué sur chaque noeud interne de T l'arête de G considérée à cette étape, et sur chaque arête le sommet de G qu'on a choisi de supprimer. Pour chaque feuille l de T, on considère S_l ensemble des sommets supprimés pour arriver à cette feuille ; l est alors étiquetée par Succès si S_l est un transversal de G, par Echec sinon. Ainsi, la deuxième feuille en partant de la gauche est étiquetée par Succès car $\{a, b, e\}$ est un transversal de G.

Notons que l'algorithme de la Proposition 3.8 permet en fait d'énumérer les transversaux minimaux de cardinal $\leq p$. La construction de l'arbre peut

s'implémenter par exemple par un parcours en largeur ou par un parcours en profondeur. Toutefois, dans le cas où on souhaite uniquement un algorithme de décision et non d'énumération, il peut être préférable d'effectuer un parcours en profondeur et de s'arrêter dès obtention d'une solution, ce qui donne lieu à un algorithme récursif simple décrit en Annexe A.

On va maintenant décrire une kernelisation pour le problème k-HS, qui repose sur un lemme combinatoire appelé *lemme du tournesol* ou *lemme d'Erdos-Rado*. Etant donné un hypergraphe $H = (V, E)$, un *tournesol* de H est une famille $T = \{e_1, ..., e_k\}$ d'arêtes telle que chaque paire d'arêtes a la même intersection. En d'autres termes, il existe un ensemble $C \subseteq V$, appelé le *coeur* de T, satisfaisant

$$e_i \cap e_j = C \text{ pour tous } i, j \text{ distincts}$$

Lemme 3.9 (Lemme du tournesol). *Soit $k, d \in \mathbb{N}$, et soit $H = (V, E)$ un hypergraphe d-uniforme comportant plus de $(k-1)^d d!$ arêtes. Alors il existe un tournesol de cardinal k dans H.*

Démonstration. Par récurrence sur d. Si $d = 1$, alors chaque ensemble de k arêtes forme un tournesol de coeur $\emptyset$. Supposons maintenant $d > 1$. Soit $D = \{f_1, ..., f_m\}$ un ensemble maximal d'arêtes deux à deux disjointes. Si $|D| \geq k$, alors D est un tournesol de coeur $\emptyset$. Sinon, soit $W = f_1 \cup ... \cup f_m$, alors $|W| \leq (k-1)d$ puisque $|D| < k$. Par maximalité de D, on a $W \cap f \neq \emptyset$ pour tout $f \in E$. Pour chaque $x \in W$, notons F_x l'ensemble des arêtes de E contenant x. Il existe alors $x \in W$ tel que

$$|F_x| \geq \frac{|E|}{|W|} \geq \frac{(k-1)^d d!}{(k-1)d} = (k-1)^{d-1}(d-1)!$$

Soit $F' = \{e \backslash \{x\} : e \in F_x\}$. Alors $H' = (V, F')$ est un hypergraphe $(d-1)$-uniforme comportant plus de $(k-1)^{d-1}(d-1)!$ arêtes. Par hypothèse de récurrence, H' comporte un tournesol de cardinal k, $S = \{e_1, ..., e_k\}$ de coeur C. On obtient alors un tournesol de cardinal k dans H : $S' = \{e_1 \cup \{x\}, ..., e_k \cup \{x\}\}$ de coeur $C \cup \{x\}$. □

Ce résultat fournit une kernelisation pour $k -$ HS :

Proposition 3.10. *Il existe une kernelisation pour $k -$ HS qui à partir d'une instance (H, p) produit en temps $O(k|H|^2)$ une instance (H', p) avec $|H'| \leq O(p^k.k!.k^2)$.*

Démonstration. Commençons par remarquer que la preuve du Lemme 3.9 fournit un algorithme qui : étant donné un hypergraphe d-uniforme H comportant plus de $(k-1)^d d!$ arêtes, en temps $O(d|H|)$ trouve un tournesol de cardinal k dans H.

Décrivons maitenant la kernelisation pour $k -$ HS. Elle repose sur l'observation suivante. Si $H = (V, E)$ contient un tournesol $T = \{e_1, ..., e_m\}$ de coeur C, un transversal de H doit soit intersecter C, soit intersecter chaque ensemble

$e_i \backslash C$. En particulier, si $m \geq p+1$, alors un transversal de H de cardinal $\leq p$ doit nécessairement intersecter C. Dans ce cas, si l'on pose $H' = (V, E')$ où $E' = E \backslash \{e_1, ..., e_m\} \cup \{C\}$, alors les instances (H, p) et (H', p) sont équivalentes. L'algorithme de kernelisation consiste donc à répéter l'opération suivante :

identifier un tournesol de cardinal $\geq p+1$, et remplacer les arêtes du tournesol par leur coeur,

tant que l'hypergraphe a plus de $O(p^k.k!.k)$ arêtes, puis à supprimer les sommets isolés de l'hypergraphe obtenu. Chaque opération de la sorte peut s'effectuer en temps $O(k|H|)$ par l'algorithme fourni par la preuve du Lemme 3.9, et le nombre d'itérations est borné par $O(|H|)$, d'où un temps total en $O(k|H|^2)$ pour l'algorithme de kernelisation. □

Le résultat précédent implique que pour k fixé, le problème $k-$HS est soluble en temps $O(|H|^2 + p^k k^p)$. La technique d'*entrelacement* permet d'abaisser cette complexité à $O(|H|^2 + k^p)$. Le principe de cette technique consiste à réappliquer l'algorithme de kernelisation à chaque étape de la recherche bornée avec la valeur courante du paramètre.

Proposition 3.11. *Pour k fixé, $k-$*HS *est soluble en temps $O(|H|^2 + k^p)$.*

Démonstration. Soit K la kernelisation de $k-$HS décrite en Proposition 3.10. On adapte l'algorithme de recherche bornée de la Proposition 3.8, de la façon suivante : au lieu d'étiqueter une feuille u de T par (H', p'), on étiquette u par $K(H', p')$.

Notons $T(k, p')$ le temps nécessaire au traitement du sous-arbre issu d'une feuille d'étiquette (H', p'). Pour k fixé, $T(k, p)$ satisfait alors la récurrence :

$$\begin{cases} T(k,0) & = O(1) \\ T(k,p) & = kT(p-1) + O(p^{2k}) \end{cases}$$

et on peut montrer que $T(k, p) = O(k^p)$. Le temps nécessaire à la première kernelisation est $O(|H|^2)$, d'où la complexité en $O(|H|^2 + k^p)$ annoncée. □

Les techniques de recherche bornée et de kernelisation ont d'abord été appliquées au problème Vertex Cover [3]. La kernelisation de k-HS est dûe à [14]. La technique d'entrelacement est dûe à [23]. Ces techniques ont été appliquées à de nombreux problèmes FPT, on peut mentionner par exemple le problème Multicut In Trees [18] et le problème Closest String [15].

3.2.2 Color-coding

On présente maintenant la technique du *color-coding*. Cette technique s'applique à des problèmes consistant à rechercher une sous-structure de taille q dans une structure donnée, q étant le paramètre. Ce problème est généralement W[1]-dur par rapport à q, l'exemple type étant la recherche d'une clique ou d'un indépendant dans un graphe.

Toutefois, dans certains cas particuliers on sait résoudre le problème en temps FPT par color-coding. Ceci consiste à générer un coloriage aléatoire de la structure avec q couleurs, et sur la structure colorée obtenue à rechercher une sous-structure de taille q *proprement colorée*, c'est-à-dire impliquant q éléments de couleurs distinctes. La propriété importante est que s'il existe une sous-structure de taille q, alors cette sous-structure va être proprement colorée avec probabilité élevée, comme l'indique le lemme suivant.

Lemme 3.12. *Soit V un ensemble d'éléments, soit $C \subseteq V$ de cardinal q. Soit $c : V \to [q]$ généré selon une loi aléatoire uniforme. Alors c est injective sur C avec probabilité $\geq \frac{1}{e^q}$.*

Démonstration. La probabilité qu'une fonction $f : [q] \to [q]$ soit injective est $\frac{q!}{q^q}$, et on conclut en utilisant le fait que $q! \geq (q/e)^q$. □

Cette observation fournit alors un algorithme FPT randomisé pour le problème, si l'on dispose d'un algorithme FPT pour la recherche d'une sous-structure proprement colorée. En outre, un tel algorithme randomisé peut généralement être dérandomisé en utilisant des *familles de fonction de hachage k-parfaites*.

On donne deux applications de la technique du color-coding. La première application est le problème Long Path : étant donné un graphe G et un entier q, décider si G a un chemin de longueur $\geq q$.

Proposition 3.13. Long Path *est soluble par un algorithme randomisé en temps $O((2e)^q m)$.*

Démonstration. On considère d'abord le problème Colored Long Path : étant donné un graphe $G = (V, E)$, un entier q, et un coloriage $c : V \to C$ (où C est un ensemble de q couleurs), existe t-il un chemin proprement coloré de longueur q ? Etant donné $S \subseteq C$, appelons S-chemin un chemin P de G tel que les couleurs associées aux sommets de P soient deux à deux distinctes et forment l'ensemble S. On doit donc décider s'il existe un C-chemin.

On va décrire un algorithme A qui résout ce problème en temps $O(2^q m)$. L'algorithme calcule par programmation dynamique un prédicat $Path(i, S, x)$ pour chaque $i \in [q]$, $S \subseteq C$ de cardinal i, $x \in V$, tel que $Path(i, S, x)$ soit vrai si et seulement si il existe un S-chemin d'extrémité x. On procède par induction sur $i = |S|$: si l'on a trouvé un S-chemin d'extrémité x, alors pour chaque sommet y voisin de x et tel que $c(y) \notin S$, on obtient un $S \cup \{c(y)\}$-chemin d'extrémité y. A l'issue de cette étape de programmation dynamique, l'algorithme répond positivement si et seulement si il existe $x \in V$ tel que $Path(q, C, x)$ soit vrai.

La complexité temporelle de l'algorithme A est :

$$\sum_{i=0}^{q} \sum_{x \in V} (d_G(x) \binom{q}{i}) = 2^q m$$

On décrit maintenant un algorithme A' qui résout Long Path en temps $O((2e)^q m)$. Soit une instance $I = (G, q)$ de Long Path. Soit un ensemble C de cardinal q. Considérons le processus suivant :

– générer un coloriage aléatoire uniforme $c : V \to C$;
– générer une instance $I_c = (G, q, c)$ de Colored Long Path ;
– exécuter l'algorithme A sur I_c.

Observons que : (i) si I est une instance négative de Long Path, alors A refuse I_c avec probabilité 1 ; (ii) si I est une instance positive de Long Path, alors A accepte I_c avec probabilité $\geq 1/e^q$. Le point (i) est clair. Le point (ii) résulte du fait que si P est un chemin de longueur q, alors il induit une solution de Colored Long Path dès lors que c est injective sur l'ensemble des sommets de P. Par le Lemme 3.12, cet évènement survient avec probabilité $\geq 1/e^q$.

En faisant e^q répétitions du processus précédent, on obtient A' algorithme randomisé pour Long Path de temps d'exécution $O((2e)^q m)$. □

Le pseudocode de l'algorithme A est décrit en annexe, où il est appelé Find-Colored-Long-Path.

La deuxième application est le problème k-Dimensional-Matching (k-DM). Etant donnés deux tuples $t, t' \in V^k$, on dit que t, t' sont *orthogonaux*, noté $t \perp t'$, si et seulement si $t[i] \neq t'[i]$ pour tout $i \in [k]$. Un ensemble $T \in V^k$ est un *couplage* si et seulement si $t \perp t'$ pour tous $t, t' \in T$ distincts. Le problème k-DM demande : étant donné un ensemble V, un ensemble de tuples $S \subseteq V^k$, et un entier q, existe t-il un couplage $T \subseteq S$ de cardinal $\geq q$?

Posons $n = |S|$ et $m = |V|$. On montre :

Proposition 3.14. *k-DM peut se résoudre en temps $O((2e)^{kq} kn)$.*

Démonstration. On va d'abord présenter un algorithme A résolvant k-DM en temps $O(2^{km} kn)$. On aura besoin des définitions suivantes. Si $0 \leq j \leq m$, un *j-filtre* est un tuple $F = (f_1, ..., f_k)$, où chaque f_i est un sous-ensemble de V de cardinal j. Etant donné un j-filtre $F = (f_1, ..., f_k)$ et un tuple $t = (t_1, ..., t_k)$, on dit que t est *inclus dans* F si et seulement si pour tout $i \in [k]$, $t_i \in f_i$, et on dit que t est *orthogonal* à F si et seulement si pour tout $i \in [k], t_i \notin f_i$. Etant donné un j-filtre $F = (f_1, ..., f_k)$, F est *réalisé* par S si et seulement si il existe un couplage de cardinal j dont tous les éléments sont inclus dans F.

L'algorithme A calcule par programmation dynamique un prédicat $Pred(i, F)$ (pour $0 \leq i \leq m$, et pour un i-filtre F), tel que $Pred(i, F)$ est vrai si et seulement si F est réalisé par S. On procède par induction sur i : si l'on a trouvé un i-filtre F tel que $Pred(i, F)$ soit vrai, on recherche un tuple de S orthogonal à F, et pour chaque tuple de la sorte on obtient un $(i+1)$-filtre F' tel que $Pred(i+1, F')$ est vrai. A l'issue de cette étape de programmation dynamique, l'algorithme calcule la taille M d'un couplage maximum comme le plus grand $i \in [m]$ pour lequel il existe F tq $Pred(i, F)$ est vrai, et renvoie vrai si et seulement si $M \geq q$.

Analysons la complexité de l'algorithme. Le nombre de i-filtres examinés à l'étape i est $\leq \binom{m}{i}^k$. Chaque filtre F est traité en temps $O(kn)$. La complexité temporelle de l'algorithme A est donc :

$$\sum_{i=0}^{m-1} \binom{m}{i}^k \times kn \leq (\sum_{i=0}^{m-1} \binom{m}{i})^k \times kn \leq 2^{km} kn$$

On décrit maintenant un algorithme A' qui résout k-Dimensional Matching en temps $O((2e)^{kq}kn)$. Soit une instance $I = (V, S, q)$ du problème. Soit C un ensemble de cardinal q. Considérons le processus suivant :

- générer de façon aléatoire un tuple $\Phi = (f_1, ..., f_k)$, où chaque $f_i : V \to C$ est généré selon une loi aléatoire uniforme ;
- générer une instance $I_\Phi = (C, S_\Phi, q)$ de k-Dimensional-Matching, où $S_\Phi = \{(f_1(t_1), ..., f_k(t_k)) : (t_1, ..., t_k) \in S\}$;
- exécuter l'algorithme A sur I_Φ.

Observons que : (i) si I est une instance négative de k-DM, alors A refuse I_Φ avec probabilité 1 ; (ii) si I est une instance positive de k-DM, alors A accepte I_Φ avec probabilité $\geq 1/e^{qk}$. Le point (i) est clair, et le point (ii) résulte du fait que si $T = \{t_1, ..., t_q\}$ est un couplage inclus dans S, alors il induit un couplage de S_Φ dès que chaque fonction f_i est injective sur $\{t_1[i], ..., t_q[i]\}$. Par le Lemme 3.12, cet évènement survient avec probabilité $\geq 1/e^{qk}$.

En faisant e^{qk} répétitions du processus précédent, on obtient A' algorithme randomisé pour k-DM, de temps d'exécution $O((2e)^{kq}kn)$. □

Le pseudocode de l'algorithme A est décrit en annexe, où il est appelé Find-k-DM.

L'introduction du color-coding, ainsi que l'application au problème Long Path, est dûe à [1] ; la technique y est également appliquée à la recherche d'autres types de sous-structures : cycles, arbres, et plus généralement graphes de largeur arborescente bornée, améliorant un résultat de [25]. La technique du color-coding a été également mise en oeuvre dans [13, 20, 12].

3.2.3 Compression itérative

On présente finalement la technique de *compression itérative*. Cette technique s'applique à des problèmes du type suivant : étant donné un objet étiqueté, on cherche à supprimer au plus p étiquettes tel que l'objet résultant vérifie une certaine propriété P.

Formellement, on se donne une opération de *suppression*, c'est-à-dire que si O est un objet sur un ensemble d'étiquettes L et si $L' \subseteq L$, on sait définir l'objet $O \backslash L'$ obtenu en supprimant les étiquettes de L' dans O. On se donne également une propriété héréditaire P : si O vérifie P alors $O \backslash L'$ vérifie P. Etant donné O objet étiqueté sur L, on dit qu'un *casseur* de O est un ensemble $L' \subseteq L$ tel que $O \backslash L'$ vérifie P. On considère alors un problème Π de la forme suivante : étant donné O objet étiqueté sur L, et un entier p, est-ce que O admet un casseur de cardinal $< p$?

La technique de compression itérative consiste à résoudre Π de façon inductive. Etant donné un objet O sur L dont on cherche un casseur de taille $< p$, on construit de façon inductive un sous-objet O' de O et un casseur C de taille $< p$ de O'. Lors de l'examen d'un nouvel élément x, on ajoute x à C, et l'ensemble C obtenu est un casseur de taille $\leq p$ du nouvel objet. Le cas problématique est lorsque C est de taille p, car alors l'invariant n'est plus vérifié. A l'aide d'un algorithme de *compression* on est alors en mesure, soit de trouver un autre

casseur C' de taille $< p$, soit de conclure qu'il n'existe pas de tel casseur. Cet algorithme de compression procède généralement en examinant chaque possibilité pour $C \cap C'$, et pour chaque partition de C en $C_1 \cup C_2$, en cherchant un casseur $C' = C_1 \cup C'_2$ avec C'_2 disjoint de C_2 et $|C'_2| < |C_2|$. Ceci est illustré ci-dessous.

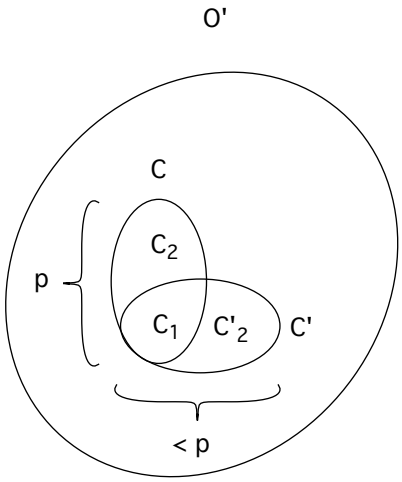


FIG. 3.4 – Etant donné C casseur de cardinal p, et étant donnée une partition de C en $C_1 \cup C_2$, on cherche $C'_1 = C_1 \cup C'_2$ casseur de cardinal $< p$.

On va formaliser cette technique dans la proposition suivante. On associe à Π son *problème dérivé* Π' défini comme suit : étant donné O objet étiqueté sur L, un entier p, et C casseur de O de cardinal p, existe t-il C' casseur de O disjoint de C et de cardinal $< p$? Le problème Π' est plus simple à résoudre que Π : en effet, on sait que $O \backslash C$ vérifie P, et l'objet O a donc une structure particulière qui peut aider à résoudre le problème. Il se trouve qu'on peut déduire un algorithme fpt pour Π à partir d'un algorithme fpt pour Π'.

Proposition 3.15. *Si Π' est soluble en temps $O(\alpha^p T(n))$, alors Π est soluble en temps $O((\alpha+1)^p n T(n))$.*

Démonstration. Considérons une instance de Π, c'est la donnée d'un objet O sur L et d'un entier p. Supposons qu'on ait un algorithme SOLVE-$\Pi'(O, L, p, C)$ qui résolve Π' en temps $O(\alpha^p T(n))$. On résout Π par l'algorithme de compression itérative, SOLVE-$\Pi(O, L, p)$ décrit ci-après. On maintient un ensemble $L' \subseteq L$, et un casseur C de $O|L'$ de cardinal $< p$. Initialement $L' = \emptyset$ est l'objet vide et $C = \emptyset$. L'algorithme va effectuer n étapes, chaque étape consistant à insérer un nouvel élément dans L'.

Lors d'une étape, on choisit $x \in L \backslash L'$, on ajoute x à L' et à C. Le nouvel ensemble C obtenu est un casseur de $O|L'$ de cardinal $\leq p$, et on souhaite qu'à la fin de l'étape C soit un casseur de cardinal $< p$. Si l'ensemble vérifie déjà cette propriété, il n'y a rien à faire. Sinon, on appelle un algorithme COMPRESSION(O, L', p, C) qui :

(i) soit renvoie un casseur C' de $O|L'$ de cardinal $< p$;
(ii) soit répond qu'il n'existe pas de tel casseur.

Dans le cas (i), l'algorithme principal passe à l'étape suivante avec $C = C'$. Dans le cas (ii), il répond que O n'admet pas de casseur de cardinal $< p$: notons que sa réponse est correcte du fait que P est héréditaire.

L'algorithme Compression procède comme suit. Rappelons qu'il reçoit un casseur C de $O|L'$ de cardinal p et qu'il doit décider s'il existe un casseur C' de $O|L'$ de cardinal $< p$. L'algorithme va alors examiner chaque possibilité pour $C' \cap C$. Pour chaque partition de C en C_1, C_2, il va rechercher un casseur $C' = C_1 \cup C'_2$ en se ramenant au problème Π' : chercher C' de la sorte revient à chercher C'_2 casseur de $O|(L'\backslash C_1)$ de cardinal $< |C_2|$. Il va résoudre ce problème par un appel à Solve-$\Pi'(O, L'\backslash C_1, |C_2|, C_2)$.

Analysons le temps d'exécution de l'algorithme. On va justifier que chaque appel à Compression requiert un temps $O((\alpha+1)^p T(n))$. Soit $0 \leq i \leq p$. Considérons les partitions de C en C_1, C_2 telles que $|C_2| = i$. Ces partitions sont en nombre $\binom{p}{i}$. Pour chaque partition de la sorte, l'appel à Solve-Π' prend un temps $O(\alpha^i T(n))$. Le temps pris par l'appel à Compression est donc :

$$O(\sum_{i=0}^{p} \binom{p}{i} \alpha^i T(n)) = O((\alpha+1)^p T(n))$$

On en déduit que chaque étape de l'algorithme Solve-Π prend un temps $O((\alpha+1)^p T(n))$, et l'algorithme prend donc un temps total $O((\alpha+1)^p n T(n))$. □

Les pseudocodes des algorithmes Solve-Π et Compression décrits ci-dessus sont donnés en annexe.

On va maintenant illustrer la technique de compression itérative à travers deux exemples simples. Le premier exemple est le problème Cluster Vertex Deletion (CVD). Un *cluster graph* est un graphe dont les composantes connexes sont des cliques. Le problème CVD prend un graphe G, et demande s'il est possible de transformer G en un cluster graph en supprimant moins de p sommets.

Il est facile de voir qu'un graphe est un cluster graph si et seulement si il ne contient pas de P_3 induit. En outre, il est possible en temps $O(m)$, étant donné un graphe G, de décider si G est un cluster graph, ou de trouver un P_3 induit. Le problème CVD est donc soluble en temps $O(3^p m)$ par recherche bornée. On montre que la compression itérative permet d'obtenir un algorithme plus rapide.

Proposition 3.16. CVD *est soluble en temps* $O(2^p n^3)$.

Ebauche. Considérons le problème CVD − Compression qui demande : étant donné un graphe $G = (V, E)$ et un ensemble $X \subseteq V$ tel que $G\backslash X$ est un cluster graph, trouver $Y \subseteq V\backslash X$ de cardinal maximum tel que $G[X \cup Y]$ soit un cluster graph. Notons $s(G, X)$ la taille d'un tel ensemble. On va montrer que CVD − Compression peut se résoudre en temps $O(n^2)$. En observant que CVD$'$ est le problème complémentaire de CVD − Compression, on conclura

que CVD′ est soluble en temps $O(n^2)$, ce qui impliquera le résultat annoncé pour CVD par la Proposition 3.15.

Soit G, X comme dans l'énoncé. Si $G[X]$ n'est pas un cluster graph, on échoue (puisque G doit pouvoir être transformé en un cluster graph en conservant tous les éléments de X). Sinon, on a la propriété que $G[X]$ et $G\backslash X$ sont tous deux des clusters graphs, représentés par des partitions P_1, P_2 respectivement. On va alors calculer $s(G, X)$ comme le poids d'un couplage maximum pondéré d'un graphe H.

On commence par éliminer les sommets $u \in V\backslash X$ tels que : (i) soit $N_X(u)$ est non vide et est inclus strictement dans une classe de P_1, (ii) soit $N_X(u)$ intersecte deux classes de P_1. L'instance résultante a alors la propriété suivante : pour tout $u \in V\backslash X$, $N_X(u)$ est soit $\emptyset$, soit une classe de P_1. On partitionne $V\backslash X$ en :

$$V_0 = \{u \in V\backslash X : N_X(u) = \emptyset\}$$
$$\forall C \text{ classe de } P_1, V_C = \{u \in V\backslash X : N_X(u) = C\}$$

On construit un graphe biparti pondéré $H = (A \cup B, w)$ comme suit : les sommets de A sont les classes de P_1 ainsi qu'un ensemble de $n = |V\backslash X|$ sommets $x_1, ..., x_n$, les sommets de B sont les classes de P_2, et pour chaque $u \in A, v \in B$, on pose :

$$\text{si } u = x_i, v = C' \text{ classe de } P_2 : w(u, v) = |C' \cap V_0|$$
$$\text{si } u = C \text{ classe de } P_1, v = C' \text{ classe de } P_2 : w(u, v) = |C' \cap V_C|$$

On calcule alors $s(G, X)$ comme le poids d'un couplage maximum de H. On obtient donc un algorithme résolvant CVD−Compression en temps $O(n^2)$. □

Le second exemple est le problème Feedback Vertex Set for Tournaments (Fvst). Un *tournoi* est une orientation d'un graphe complet, autrement dit c'est un digraphe $T = (V, A)$ tel que pour chaque $x, y \in V$ distincts, $|A \cap \{(x, y), (y, x)\}| = 1$. Un tournoi $T = (V, A)$ est acyclique si et seulement si il existe une énumération $\pi = v_1...v_n$ de V telle que pour tous i, j, $i < j$, on a $(v_i, v_j) \in A$. Le problème Fvst prend un tournoi T, et demande s'il est possible de rendre T acyclique en supprimant moins de p sommets.

Un résultat bien connu est qu'un tournoi est acyclique si et seulement si il ne comporte pas de cycle orienté de longueur 3 (C_3). De plus, il est possible en temps $O(m)$, étant donné un tournoi T, de décider si T est acyclique, ou de trouver un C_3. Ceci permet donc de résoudre Fvst en temps $O(3^p m)$ par recherche bornée. La compression itérative permet d'obtenir un algorithme plus efficace, dû à [8].

Proposition 3.17. Fvst *est soluble en temps* $O(2^p n^3)$.

Ebauche. Considérons le problème Fvst − Compression qui demande : étant donné un tournoi $T = (V, A)$ et un ensemble $X \subseteq V$ tel que $T\backslash X$ soit un tournoi, trouver $Y \subseteq V\backslash X$ de cardinal maximum, tel que $T[X \cup Y]$ soit un tournoi. Notons $s(T, X)$ la taille d'un tel ensemble. On va montrer que Fvst −

Compression peut se résoudre en temps $O(n^2)$. En observant que Fvst$'$ est le problème complémentaire de Fvst $-$ Compression, on conclura que Fvst$'$ est soluble en temps $O(n^2)$, ce qui impliquera le résultat annoncé pour Fvst par la Proposition 3.15.

Soit G, X comme dans l'énoncé. On échoue si $G[X]$ n'est pas acyclique. Sinon, on a la propriété que $G[X]$ et $G\backslash X$ sont tous deux acycliques, soient s_1, s_2 les énumérations associées. On va alors calculer $s(T, X)$ comme la taille d'une plus longue sous-séquence commune à deux séquences.

On commence par éliminer les sommets $u \in V\backslash X$ qui forment un C_3 avec deux sommets de X. Le tournoi résultant a alors la propriété suivante. Pour tout $u \in V\backslash X$, soit $P_u = \{v \in X : (v, u) \in A\}$ et $S_u = \{v \in X : (u, v) \in A\}$. On a alors $(x, y) \in A$ pour tout $x \in P_u, y \in S_u$ (sinon x, y, u formeraient un C_3). Soit $p_u = |P_u|$, et supposons que $s_1 = x_1...x_p$, alors $P_u = \{x_i : i \leq p_u\}$ et $S_u = \{x_i : i > p_u\}$.

On définit alors s'_1 en énumérant les éléments de $V\backslash X$ selon l'ordre suivant : $u < v$ si et seulement si $p_u < p_v$ ou ($p_u = p_v$ et $u <_{s_2} v$). On calcule alors $s(T, X)$ comme la longueur d'une plus longue sous-séquence commune à s'_1, s_2. Ceci peut se faire en temps $O(n^2)$, d'où la complexité annoncée pour Fvst $-$ Compression. □

La technique de compression itérative a été introduite dans [26] pour obtenir un algorithme FPT pour le problème Vertex Bipartization, a été proposée comme technique générale dans [7], et a été appliquée par la suite à de nombreux problèmes de type vertex-deletion (Feedback Vertex Set [17, 6], Directed Feedback Vertex Set [4], Cluster Vertex Deletion [19], Chordal Vertex Deletion [21], Feedback Vertex Set in Tournaments [8]...).

3.2.4 Autres techniques

On peut également mentionner plusieurs autres techniques pour l'obtention d'algorithmes FPT :

- les techniques de *localisation gloutonne* et de *méthode extrémale* ont été appliquées à plusieurs problèmes de maximisation [7, 20] ;
- les *décompositions arborescentes* : de nombreux problèmes NP-durs sur les graphes peuvent se résoudre en temps $f(w)n^c$ sur les graphes de largeur arborescente $\leq w$. L'obtention d'algorithmes FPT par rapport à la largeur arborescente peut se faire soit de manière ad hoc, en développant un algorithme de programmation dynamique adapté au problème, soit en montrant que le problème est exprimable en logique monadique du second ordre et en appliquant le théorème de Courcelle [5]. Un autre outil important dans ce cadre est l'algorithme de Bodlaender [2] qui permet de reconnaître les graphes de largeur arborescente $\leq w$ en temps $2^{O(w^3)}n$, ou l'algorithme de Reed [24] qui permet de 4-approximer la largeur arborescente en temps $2^{O(w)}n^2$. Notons qu'il existe également d'autres mesures alternatives à la largeur arborescente possédant des propriétés similaires.

- l'utilisation du *théorème des mineurs* de Robertson-Seymour [27] et des résultats algorithmiques associés. Le résultat central est que étant donnés deux graphes G, H d'ordre m, n, décider si G est un mineur de H est FPT par rapport à m. Ceci implique l'existence d'un algorithme FPT (non uniforme) pour tout problème paramétré sur des graphes dont chaque tranche soit une famille de graphes close par mineur. Cependant, cette technique est non-constructive : elle ne fournit qu'un résultat d'existence théorique, et ne donne pas lieu à des algorithmes utilisables en pratique.
- la technique "Win/Win" : cette technique consiste à exploiter une structure particulière des instances positives du problème considéré. On teste alors si l'instance présente cette structure, on rejette si ce n'est pas le cas, et sinon on exploite cette structure pour résoudre le problème par un algorithme FPT. Cette technique a été notamment mise en oeuvre pour obtenir des algorithmes en $2^{O(\sqrt{p})}n^c$ pour des problèmes sur les graphes planaires, et sur les graphes excluant un mineur (voir [9] pour un article de synthèse). On utilise ici le fait qu'une instance positive du problème doit être de largeur arborescente $O(\sqrt{p})$. La résolution du problème comporte alors deux étapes. Lors d'une première étape, on obtient une décomposition arborescente de largeur $O(\sqrt{p})$, par exemple par l'algorithme de Bodlaender ou de Reed. Lors d'une seconde étape, on résout le problème par programmation dynamique sur la décomposition arborescente obtenue.

3.3 Annexe

Annexe A

SolveHS(H, p)
  si H ne contient aucune arête **alors**
    renvoyer *oui*
  sinon si $p = 0$ **alors**
    renvoyer *non*
  sinon
    choisir e arête de H
    pour chaque $x \in e$ **faire**
      $r \leftarrow$ SolveHS$(H \backslash x, p-1)$
      si $r = oui$ alors renvoyer r
    fin pour
  fin si
  renvoyer *non*

Annexe B

FIND-COLORED-LONG-PATH(G, c)

pour chaque $x \in V$ **faire**
  initialiser les valeurs $Path(i, S, x)$ à $vrai$ si $i = 0$, $faux$ sinon
fin pour
pour i de 0 à $q-1$, et pour chaque $x \in V$ **faire**
  pour chaque $y \in N_G(x)$ et pour chaque S de cardinal i tel que $Path(i, S, x) = vrai$ **faire**
    si $c(y) \notin S$ **alors**
      $Path(i+1, S \cup \{c(y)\}, y) \leftarrow vrai$
    fin si
  fin pour
fin pour
renvoyer $vrai$ si $\exists x, Path(q, C, x) = vrai$, $faux$ sinon

---

FIND-k-DM(V, S, q)

pour chaque $i \geq 1$, pour chaque i-filtre F **faire**
  $Pred(i, F) \leftarrow faux$
fin pour
soit le 0-filtre $F = (\emptyset, ..., \emptyset)$
$Pred(0, F) \leftarrow vrai$
pour i de 0 à $m-1$ **faire**
  pour chaque i-filtre F tel que $Pred(i, F) = vrai$ **faire**
    pour chaque $t \in S$ orthogonal à F **faire**
      supposons que $F = (f_1, ..., f_k)$ et que $t = (t_1, ..., t_k)$
      définir $F' = (f_1 \cup \{t_1\}, ..., f_k \cup \{t_k\})$
      $Pred(i+1, F') \leftarrow vrai$
    fin pour
  fin pour
fin pour
calculer $M = \max\{i \in [m] : \exists F, Pred(i, F) = vrai\}$
renvoyer $(M \geq q)$

Annexe C

COMPRESSION(O, L, p, C)
  pour chaque partition de C en C_1, C_2 **faire**
    $r, C' \leftarrow$ SOLVE-$\Pi'(O, L \backslash C_1, |C_2|, C_2)$
    si $r = oui$ **alors**
      renvoyer $oui, C_1 \cup C'$
    fin si
  fin pour
  renvoyer $non, \_$

SOLVE-$\Pi(O, L, p)$
  $C \leftarrow \emptyset$, $L' \leftarrow \emptyset$
  tant que $L' \subset L$ **faire**
    choisir $x \in L \backslash L'$
    $C \leftarrow C \cup \{x\}$, $L' \leftarrow L' \cup \{x\}$
    si $|C| \geq p$ **alors**
      $r, C' \leftarrow$ COMPRESSION(O, L', p, C)
      si $r = non$ **alors**
        renvoyer non
      sinon
        $C \leftarrow C'$
      fin si
    fin si
  fin tant que
  renvoyer oui, C

3.4 Bibliographie

[1] Alon (N.), Yuster (R.) et Zwick (U.). – Color-coding. *Journal of the Association for Computing Machinery*, vol. 42, n° 4, 1995, pp. 844–856.

[2] Bodlaender (H.L.). – A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM Journal on Computing*, vol. 25, n° 6, 1996, pp. 1305–1317.

[3] Buss (J.F.) et Goldsmith (J.). – Nondeterminism within P. *SIAM Journal on Computing*, vol. 22, n° 1, 1993, pp. 560–572.

[4] Chen (J.), Liu (Y.), Lu (S.), O'Sullivan (B.) et Razgon (I.). – A fixed-parameter algorithm for the directed feedback vertex set problem. *In : Proceedings of STOC'08*, pp. 177–186.

[5] Courcelle (B.). – The monadic second-order logic of graphs I : Recognizable sets of finite graphs. *Information and Computation*, vol. 85, 1990, pp. 12–75.

[6] Dehne (F.), Fellows (M.), Langston (M.), Rosamond (F.) et Stevens (K.). – An $O(2^{O(k)}n^3)$ FPT Algorithm for the Undirected Feedback Vertex Set Problem. *In : Proceedings of COCOON'05*, pp. 859–869.

[7] Dehne (F.), Fellows (M.), Rosamond (F.) et Shaw (P.). – Greedy Localization, Iterative Compression, Modeled Crown Reductions : New FPT Techniques, an Improved Algorithm for Set Splitting, and a Novel $2k$ Kernelization for Vertex Cover. *In : Proceedings of IWPEC'04*, pp. 271–280.

[8] Dom (M.), Guo (J.), Hüffner (F.), Niedermeier (R.) et Truß(A.). – Fixed-Parameter Tractability Results for Feedback Set Problems in Tournaments. *In : Proceedings of CIAC'06*, pp. 320–331.

[9] Dorn (F.), Fomin (F.) et Thilikos (D.). – Subexponential Parameterized Algorithms. *In : Proceedings of ICALP'07*, pp. 15–27.

[10] Downey (R.G.) et Fellows (M.R.). – Fixed-Parameter Tractability and Completeness II : On Completeness for W[1]. *Theoretical Computer Science*, vol. 141, n° 1–2, 1995, pp. 109–131.

[11] Downey (R.G.) et Fellows (M.R.). – *Parameterized Complexity*. – Springer-Verlag, 1999.

[12] Fellows (M.), Fertin (G.), Hermelin (D.) et Vialette (S.). – Sharp Tractability Borderlines for Finding Connected Motifs in Vertex-Colored Graphs. *In : Proceedings of ICALP'07*, pp. 340–351.

[13] Fellows (M.), Knauer (C.), Nishimura (N.), Ragde (P.), Rosamond (F.), Stege (U.), Thilikos (D.) et Whitesides (S.). – Faster Fixed-Parameter Tractable Algorithms for Matching and Packing Problems. *In : Proceedings of ESA'04*, pp. 311–322.

[14] Flum (J.) et Grohe (M.). – *Parameterized Complexity Theory*. – Springer-Verlag, 2006.

[15] Gramm (J.), Niedermeier (R.) et Rossmanith (P.). – Fixed-Parameter Algorithms for Closest String and Related Problems. *Algorithmica*, vol. 37, n° 1, 2003, pp. 25–42.

[16] Grohe (M.), Schwentick (T.) et Segoufin (L.). – When is the evaluation of conjunctive queries tractable ? *In : Proceedings of STOC'01*, pp. 657–666.

[17] Guo (J.), Gramm (J.), Huffner (F.), Niedermeier (R.) et Wernicke (S.). – Improved Fixed-Parameter Algorithms for Two Feedback Set Problems. *In : Proceedings of WADS'05*, pp. 158–168.

[18] Guo (J.) et Niedermeier (R.). – Fixed-parameter tractability and data reduction for multicut in trees. *Networks*, vol. 46, n° 3, 2005, pp. 124–135.

[19] Huffner (F.), Komusiewicz (C.), Moser (H.) et Niedermeier (R.). – Fixed-Parameter Algorithms for Cluster Vertex Deletion. *In : Proceedings of LATIN'08*, pp. 711–722.

[20] Liu (Y.), Lu (S.), Chen (J.) et Sze (S.). – Greedy Localization and Color-Coding : Improved Matching and Packing Algorithms. *In : Proceedings of IWPEC'06*, pp. 84–95.

[21] Marx (D.). – Chordal Deletion Is Fixed-Parameter Tractable. *In : Proceedings of WG'06*, pp. 37–48.

[22] Niedermeier (R.). – *Invitation to Fixed-Parameter Algorithms.* – Oxford University Press, 2006.

[23] Niedermeier (R.) et Rossmanith (P.). – A general method to speed up fixed-parameter tractable algorithms. *Information Processing Letters*, vol. 73, n° 3–4, 2000, pp. 125–129.

[24] Perkovic (L.) et Reed (B.). – An Improved Algorithm for Finding Tree Decompositions of Small Width. *International Journal of Foundations of Computer Science*, vol. 11, n° 3, 2000, pp. 365–371.

[25] Plehn (J.) et Voigt (B.). – Finding minimally weighted subgraphs. *In : Proceedings WG'90*, pp. 18–29.

[26] Reed (B.), Smith (K.) et Vetta (A.). – Finding odd cycle transversals. *Operations Research Letters*, vol. 32, n° 4, 2004, pp. 299–301.

[27] Robertson (N.) et Seymour (P.D.). – Graph Minors. XX. Wagner's conjecture. *J. Comb. Theory, Ser. B*, vol. 92, n° 2, 2004, pp. 325–357.

Chapitre 4

Les problèmes Mast et Mct

On considère dans ce chapitre des collections d'arbres ayant tous le même ensemble d'étiquettes. Le problème Mast prend une collection d'arbres sur un même ensemble d'étiquettes, et recherche un plus grand arbre qui soit sous-arbre commun des arbres sources. Le problème Mct est une variante de Mast où l'on recherche un *pré-sous-arbre* commun. De manière équivalente, Mast, resp. Mct, recherche un plus grand ensemble d'étiquettes tel que les arbres sources restreint à cet ensemble soient isomorphes, resp. aient un raffinement commun. Ces problèmes rencontrent plusieurs applications en phylogénie, où les arbres étiquetés sont utilisés pour représenter l'histoire évolutive d'un ensemble d'espèces : les étiquettes sont en bijection avec les espèces, et la topologie indique les relations de spéciation. Les problèmes Mast et Mct sont alors utilisés : pour atteindre un consensus entre arbres obtenus par des méthodes différentes, pour définir une mesure de similarité entre arbres ou encore pour identifier des transferts de gènes horizontaux. Le problème Mct est plus adapté que Mast pour l'application à la phylogénie, où les noeuds de degré supérieur à 3 s'interprètent généralement comme une incertitude sur le branchement et non comme une spéciation multiple ; l'arbre obtenu par Mct peut ainsi lever certaines incertitudes présentes dans les arbres sources.

4.1 Problème Mast

Cette section est consacrée au problème Mast. Le plan adopté est le suivant. La section 4.1.1 contient des définitions préliminaires, ainsi qu'un historique des résultats obtenus sur Mast. En section 4.1.2, on décrit un algorithme de 3-approximation en temps linéaire pour le problème complémentaire. En section 4.1.3, on décrit des algorithmes exacts (polynomiaux ou FPT) pour Mast. On termine par des résultats de complexité (approximabilité et complexité paramétrique) présentés en section 4.1.4.

4.1.1 Préliminaires

Sous-arbres et plongements

On donne une définition équivalente en termes de plongement. Un *plongement* de S dans T est une application $\phi : N(S) \to N(T)$ telle que :

1. ϕ préserve feuilles et noeuds internes : (i) si l est une feuille de S d'étiquette x, alors $\phi(l)$ est une feuille de T d'étiquette x, (ii) si u est un noeud interne de S, alors $\phi(u)$ est un noeud interne de T ;
2. (i) si u, v sont deux noeuds de S tels que $u <_S v$, alors $\phi(u) <_T \phi(v)$; (ii) si u, v sont deux noeuds de S incomparables par $<_S$, alors $\phi(\mathsf{lca}_S(u, v)) = \mathsf{lca}_T(\phi(u), \phi(v))$.

On a alors : $S \leq T$ si et seulement si il existe un plongement de S dans T.
Cette définition est illustrée ci-dessous :

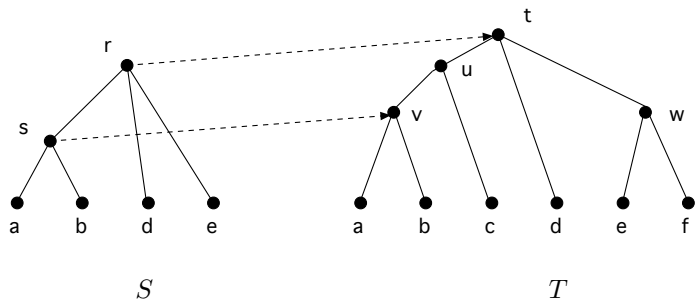


FIG. 4.1 – Deux arbres S, T tels que $S \leq T$. Le plongement ϕ de S dans T est tel que $\phi(s) = v, \phi(r) = t$, comme indiqué par les flèches en pointillé.

On a les propriétés suivantes :

Observation 4.1. *1. L'identité est un plongement de T dans T ;*

2. Si ϕ est un plongement de S dans T, et ϕ' est un plongement de T dans U, alors $\phi' \circ \phi$ est un plongement de S dans U ;

3. Si ϕ est un plongement de S dans T, alors $\phi | N(S|L')$ est un plongement de $S|L'$ dans $T|L'$.

Du lemme 4.1, il résulte que la relation *sous-arbre* jouit des propriétés suivantes :

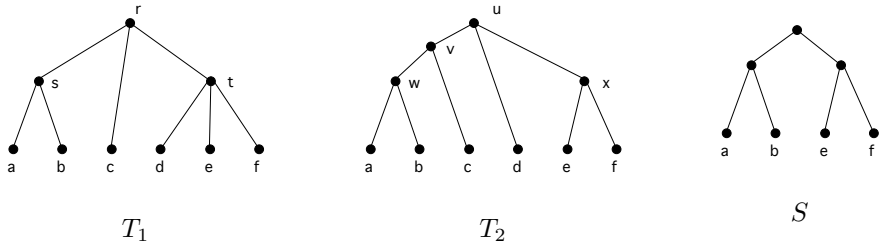
Lemme 4.2. *(i) $\leq$ est une relation d'ordre ; (ii) si $S \leq T$, alors $S|L' \leq T|L'$.*

Arbres d'accord, problème MAST

Soit $\mathcal{T} = \{T_1, ..., T_k\}$ une collection d'arbres sur un même ensemble d'étiquettes L. Soit S un arbre sur $L' \subseteq L$. On dit que S est un *arbre d'accord* de $\mathcal{T}$ si et seulement si $S \leq T_i$ pour tout i. On dit que $\mathcal{T}$ est *isomorphe* si et seulement si elle admet un arbre d'accord S tel que $L(S) = L$ (alors tous les arbres T_i sont isomorphes). Observons que ces notions ont les propriétés suivantes :

Lemme 4.3. *(i) Si T est un arbre d'accord de $\mathcal{T}$ et si $S \leq T$, alors S est un arbre d'accord de $\mathcal{T}$;*
(ii) Si T est un arbre d'accord de $\mathcal{T}$, alors $T|L'$ est un arbre d'accord de $\mathcal{T}|L'$;
(iii) Si $\mathcal{T}$ est isomorphe, alors $\mathcal{T}|L'$ est isomorphe.

L'exemple suivant illustre la notion d'arbre d'accord :



T_1 T_2 S

FIG. 4.2 – Une collection de deux arbres $\mathcal{T} = \{T_1, T_2\}$ sur l'ensemble d'étiquettes $L = \{a, b, c, d, e, f\}$, et un arbre d'accord maximum S.

Le problème MAST consiste à chercher un arbre d'accord de taille maximum d'une collection donnée en entrée. En remarquant qu'un arbre d'accord est déterminé par son ensemble d'étiquettes, on choisit de définir formellement le problème de la manière suivante : étant donnée une collection d'arbres $\mathcal{T}$ sur L, trouver $L' \subseteq L$ de cardinal maximum tel que $\mathcal{T}|L'$ soit isomorphe. On définit également le problème complémentaire CMAST : étant donnée une collection d'arbres $\mathcal{T}$ sur L, trouver $L' \subseteq L$ de cardinal minimum tel que $\mathcal{T} \backslash L'$ soit isomorphe. On note $MAST(\mathcal{T})$, resp. $CMAST(\mathcal{T})$, le coût d'une solution optimale pour MAST, resp. CMAST.

On considérera également MAST comme un problème paramétré, les paramètres d'intérêt étant les suivants : le nombre d'étiquettes n, le nombre d'arbres k, le degré maximum d. Etant donné un entier $k \geq 2$, on notera $k-$MAST la restriction du problème MAST aux collections de k arbres.

Plusieurs algorithmes pour MAST présentés dans le suite reposent sur caractérisation simple des collections isomorphes en terme de conflits. On définit deux types de conflits : un *s-conflit* entre $\mathcal{T}$ est un triplet uvw tel que $uv|w \in rt(T_i)$ et $uvw \in f(T_j)$; un *h-conflit* entre $\mathcal{T}$ est un triplet uvw tel que $uv|w \in rt(T_i)$ et $vw|u \in rt(T_j)$; un *hs-conflit* entre $\mathcal{T}$ est soit un s-conflit, soit un h-conflit entre $\mathcal{T}$. On peut montrer :

Lemme 4.4 ([4]). *(i) Deux arbres T_1, T_2 sont isomorphes si et seulement si il n'existe pas de hs-conflit entre T_1, T_2 ; (ii) Une collection $\mathcal{T}$ est isomorphe si et seulement si il n'existe pas de hs-conflit entre $\mathcal{T}$.*

Historique et nouveaux résultats

Le problème MAST sur deux arbres a été introduit dans [20], où est également décrite une heuristique pour le problème de complexité temporelle $O(n^5)$. [34]

ont décrit un algorithme sous-exponentiel pour $2-$MAST restreint aux arbres binaires, de complexité temporelle $O(n^{c \log n})$. Le premier algorithme polynomial exact pour $2-$MAST a été décrit dans [37] ; grâce à une optimisation décrite par [18], sa complexité temporelle est $O(n^{2,5} \log n)$ pour des arbres de degré non borné, et $O(d^{2,5} n^2 \log n)$ pour des arbres de degré borné.

Par ailleurs, plusieurs auteurs [18, 14, 36, 32, 33] ont présenté des algorithmes sous-quadratiques pour $2-$MAST, et des algorithmes quasi-linéaires pour le cas du degré borné. Ainsi, [18] décrivent un algorithme de complexité temporelle $O(n^{1,5} \log n)$; [14] décrivent un algorithme de complexité $O(n \log n)$ pour deux arbres binaires, et [36] a tenté de généraliser cet algorithme aux arbres de degré d, obtenant un algorithme de complexité $O(\sqrt{d} n \log n)$. Le meilleur algorithme pour le problème est dû à [33], sa complexité est $O(n^{1,5})$ dans le cas d'arbres de degré non borné, et $O(\sqrt{d} n \log \frac{2n}{d})$ dans le cas d'arbres de degré borné. L'étude du problème MAST sur un nombre d'arbres non borné a été initiée par [4, 17]. [4] ont montré la NP-difficulté de $3-$MAST dans le cas d'arbres de degré non borné. [4, 11, 17] ont montré que le problème était soluble en temps polynomial si l'un des arbres était de degré borné d, aboutissant à un algorithme de complexité $O(n^d + kn^3)$.

L'approximabilité du problème a été étudiée par plusieurs auteurs. Dans le cas d'arbres de degré non borné, [4] montrent que $3-$MAST est APX-dur, et [29] montrent que $3-$MAST n'est pas approximable à un facteur constant si $\mathsf{P} \neq \mathsf{NP}$. En s'inspirant des résultats de [22], [8] décrivent des algorithmes de 3-approximation pour le problème complémentaire CMAST : ils utilisent la caractérisation du Lemme 4.4 pour résoudre le problème par élimination de conflits disjoints, et obtiennent des algorithmes de complexité temporelle $O(kn^3)$. Dans [6] et [24], nous avons adapté la stratégie d'identification des conflits de manière à obtenir un algorithme en temps linéaire $O(kn)$.

Le problème a été également étudié sous l'angle de la complexité paramétrique. En utilisant le Lemme 4.4, [12] obtiennent des algorithmes FPT par rapport à p, de complexité respective $O(3^p kn \log n), O(kn^3 + 2.27^p)$. [8] améliorent le premier algorithme pour obtenir une complexité temporelle $O(3^p kn)$. Dans [25], nous obtenons de nouveaux résultats sur la complexité paramétrique de MAST. D'une part, nous décrivons des algorithmes FPT pour les paires de paramètres (k, d) et (k, q), de complexité respective $O(2^{2kd} n^3)$ et $O(2^{O(kq)} n^3)$. D'autre part, nous montrons que le problème est $\mathsf{W_I}[1]$-dur pour la paire de paramètre (q, d). Une conséquence de ce résultat est que MAST ne peut pas être résolu en temps $\Phi(d) N^{o(d)}$ sous l'hypothèse ETH (où N est la taille de l'instance). Ceci suggère l'optimalité asymptotique de l'algorithme en $O(n^d + kn^3)$.

4.1.2 Algorithme d'approximation

On décrit dans cette section un algorithme de 3-approximation en temps linéaire $O(kn)$ pour le problème CMAST.

Une observation simple [12, 8] est que le Lemme 4.4 permet de réduire CMAST à 3HS. Comme 3HS est 3-approximable, on obtient immédiatement un

algorithme de 3-approximation pour CMast. Cependant, la complexité temporelle de cet algorithme est $O(kn^3)$: en effet, la construction de l'ensemble des hs-conflits s'effectue en temps $O(kn^3)$, et l'exécution de l'algorithme de 3-approximation s'effectue en temps $O(n^3)$. Pour obtenir un algorithme de complexité temporelle linéaire, on adapte la stratégie de détection des conflits, de façon à pouvoir identifier et éliminer un conflit en temps constant.

Introduisons les définitions suivantes.

Définition 4.5. *Soit T_1, T_2 deux arbres sur le même ensemble d'étiquettes L. Un whs-conciliateur de T_1, T_2 est un ensemble $\mathcal{C}$ de hs-conflits disjoints entre T_1, T_2 tels que $T_2 \backslash L(\mathcal{C})$ raffine $T_1 \backslash L(\mathcal{C})$.*

Définition 4.6. *Soit $\mathcal{T}$ une collection sur un ensemble d'étiquettes L. Un hs-conciliateur de $\mathcal{T}$ est un ensemble $\mathcal{C}$ de hs-conflits disjoints entre $\mathcal{T}$ tels que $\mathcal{T} \backslash L(\mathcal{C})$ soit isomorphe.*

Le lemme 4.4 implique qu'un hs-conciliateur fournit une 3-approximation du problème CMast :

Lemme 4.7. *Si $\mathcal{C}$ est un hs-conciliateur de $\mathcal{T}$, alors $L(\mathcal{C})$ est une 3-approximation de* CMast *pour $\mathcal{T}$.*

On va décrire un algorithme TrouverWhsConciliateur qui, étant donnés deux arbres T_1, T_2, trouve un whs-conciliateur de T_1, T_2 en temps $O(n+c)$, où c est le nombre de conflits trouvés (Proposition 4.9). En utilisant cet algorithme, on obtient un algorithme de 3-approximation pour CMast en temps linéaire :

Proposition 4.8. CMast *est 3-approximable en temps $O(kn)$.*

Démonstration. Etant donnée une collection $\mathcal{T} = \{T_1, ..., T_k\}$, on détermine un hs-conciliateur de $\mathcal{T}$ de la manière suivante. Pour chaque i de 1 à $k-1$, on identifie un hs-conciliateur $\mathcal{C}_i$ de T_i, T_{i+1} , et on supprime ses étiquettes de la collection. Identifier un hs-conciliateur de T_i, T_{i+1} se fait par deux appels symétriques à TrouverWhsConciliateur, le premier avec les arguments (T_i, T_{i+1}), et le second avec les arguments (T_{i+1}, T_i).

Observons d'abord que cet algorithme construit bien un hs-conciliateur de $\mathcal{T}$: en effet, notons $\mathcal{C}'_i$ l'ensemble des conflits identifiés à la fin de l'étape i, on montre par récurrence que $\mathcal{C}'_i$ est un hs-conciliateur de $\{T_1, ..., T_i\}$. Ceci résulte du Lemme 4.3 et du fait que $\mathcal{C}'_i = \mathcal{C}'_{i-1} \cup \mathcal{C}_i$, où $\mathcal{C}'_{i-1}$ est un hs-conciliateur de $\{T_1, ..., T_{i-1}\}$ et $\mathcal{C}_i$ est un hs-conciliateur de T_{i-1}, T_i. Finalement, le fait que l'algorithme calcule une 3-approximation de CMast pour $\mathcal{T}$ résulte du Lemme 4.7.

Comme l'algorithme fait $2(k-1)$ appels à TrouverWhsConciliateur, et que chaque appel prend un temps $O(n)$, le temps d'exécution est clairement $O(kn)$. □

On montre maintenant :

Proposition 4.9. *Il existe un algorithme* TrouverWhsConciliateur(T_1, T_2) *qui détermine un whs-conciliateur de T_1, T_2 en temps $O(n+c)$.*

Bien qu'on cherche ici un whs-conciliateur et non pas un hs-conciliateur, l'algorithme utilise des idées similaires à l'algorithme de [6]. Décrivons dans un premier temps le principe général de l'algorithme, pour décrire ensuite les optimisations qui conduisent à une complexité temporelle linéaire. L'algorithme maintient un ensemble de hs-conflits disjoints $\mathcal{C}$ entre T_1, T_2, initialisé à $\emptyset$. Il effectue un parcours ascendant de T_1, en cherchant à apparier chaque noeud u de T_1 avec un noeud v de T_2 tel que $T_2(v)\backslash L(\mathcal{C})$ raffine $T_1(u)\backslash L(\mathcal{C})$.

Pour rendre le processus plus intuitif, on considère que les sous-arbres de T_1, T_2 sont élagués au fur à mesure des appariements. Ainsi, lorsqu'un noeud u de T_1 est apparié à un noeud v de T_2, ils sont considérés comme des feuilles pour la suite de l'algorithme. Chaque feuille x de T_2 correspond donc à un sous-arbre de l'arbre initial, et une variable $I(x)$ indique son ensemble d'étiquettes.

Lorsqu'un noeud u de T_1 a été apparié avec un noeud v de T_2, cette information est mémorisée par une variable $match(u)$ égale à v. Initialement, chaque feuille de T_1 est appariée avec la feuille de T_2 de même étiquette. Lors de l'examen d'un nouveau u de T_1, on appelle la procédure ApparierNoeud(u) qui cherche à apparier T_1 avec un noeud de T_2. Cette procédure va chercher à supprimer des hs-conflits disjoints $ll'l''$ avec $l, l' \in I(u), l'' \notin I(u)$; elle peut renvoyer $\perp$, si ces éliminations de conflits ont conduit à supprimer toutes les étiquettes de $I(u)$, ou renvoyer v, si les suppressions ont permis d'apparier u avec un noeud v de T_2.

L'algorithme a donc la structure suivante :

TrouveWhsConciliateur(T_1, T_2)
  {initialisation}
  InitialiserStructures()
  pour chaque feuille u de T_1 **faire**
    soit v la feuille de T_2 de même étiquette
    $match(u) \leftarrow v$
  fin pour
  pour chaque noeud interne u de T_1 examiné en ordre postfixe **faire**
    $v \leftarrow$ ApparierNoeud(u)
    si $v \neq \perp$ alors $match(u) \leftarrow v$
  fin pour

Décrivons maintenant le processus d'appariement. Lors de l'appel de la procédure ApparierNoeud(u), les fils $u_1, ..., u_d$ de u ont été appariés à des noeuds $v_1, ..., v_d$ de T_2, et on cherche à apparier u avec un noeud v de T_2, en procédant de la façon suivante. Colorons en noir les feuilles v_i, et en blanc les autres feuilles de T_2. L'appariement sera possible si et seulement si :

Condition 4.10. *Soit $v = \mathsf{lca}_{T_2}(v_1, ..., v_d)$, alors $T_2(v)$ ne contient que des feuilles noires.*

Tant que cette condition n'est pas vérifiée, on a la propriété que le sous-arbre $T_2(v)$ contient au moins une feuille blanche. Une procédure TrouveConflit

trouve alors deux feuilles noires x, y et une feuille blanche $z \in T_2'$ telle que $xy|z \notin rt(T_2)$. En choisissant $l \in I(x), l' \in I(y), l'' \in I(z)$, on obtient un hs-conflit $\{l, l', l''\}$ entre T_1, T_2. On supprime alors les trois étiquettes l, l', l'' par trois appels à une procédure *supprimeEtiquette*, chaque appel se chargeant de mettre à jour l'ensemble d'étiquettes de la feuille correspondante : par exemple, *supprimeEtiquette*(l) fait la mise à jour $I(x) \leftarrow I(x) - \{l\}$, et supprime la feuille x de T_2 si jamais $I(x)$ devient vide.

On va maintenant montrer que l'identification de conflits par la procédure TrouveConflit peut se faire en temps amorti $O(1)$, en utilisant des structures de données appropriées. Pour ce faire, on *ordonne* T_2, c'est-à-dire qu'on maintient une permutation $\mathcal{I}$ des feuilles de T_2 telle que pour chaque noeud u de T_2, l'ensemble des feuilles descendant de u est un intervalle de $\mathcal{I}$. La permutation $\mathcal{I}$ est représentée par une liste chaînée de feuilles. Si u est une feuille de T_2, on écrira $u <_{\mathcal{I}} v$ pour indiquer que u précède v dans $\mathcal{I}$, et on notera $pred_{\mathcal{I}}(u)$ (resp. $succ_{\mathcal{I}}(u)$) le prédécesseur (resp. successeur) de u dans $\mathcal{I}$, ou bien $\perp$ si u est le premier (resp. dernier) élément de $\mathcal{I}$.

On a alors la propriété suivante :

Lemme 4.11. *Soient x, y, z trois feuilles de T_2 telles que $x <_{\mathcal{I}} y <_{\mathcal{I}} z$. Alors : $\mathsf{lca}_{T_2}(x, y) \leq_{T_2} \mathsf{lca}_{T_2}(x, z)$ et $\mathsf{lca}_{T_2}(y, z) \leq_{T_2} \mathsf{lca}_{T_2}(x, z)$.*

Chaque feuille v de T_2 est représentée par une structure comportant divers champs. Outre les champs nécessaires au chaînage dans $\mathcal{I}$ et au chaînage entre noeuds dans T_2, elle comporte les champs suivants : (i) un champ $I(v)$ qui mémorise l'ensemble d'étiquettes associé à v, (ii) un champ $couleur(v)$ qui mémorise la couleur de v (blanc ou noir). En outre, chaque noeud v de T_2 comporte un champ $nbFeuillesNoires(v)$ qui mémorise le nombre de feuilles noires fils de v. Par souci de simplicité, on suppose que les feuilles et les noeuds internes sont représentés par la même structure, dont certains champs pourront être inutilisés.

On va maintenir l'invariant suivant.

Invariant 4.12. *Au début de chaque appel à* TrouveConflit*, (i) la liste $FeuillesNoires$ contient les feuilles noires de T_2, (ii) pour chaque noeud v de T_2, $nbFeuillesNoires(v)$ compte le nombre de feuilles noires fils de v.*

Pour maintenir cet invariant, on utilise deux procédures : (i) $devientFeuilleNoire(v)$ est appelée lorsque v devient une feuille noire, elle ajoute v à la liste $FeuillesNoires$ et incrémente le compteur $nbFeuillesNoires$ du père de v ; (ii) $supprimerFeuille(v)$ qui supprime la feuille v de T_2, et si v était noire supprime v de la liste $FeuillesNoires$ et décrémente le compteur $nbFeuillesNoires$ du père de v. On suppose que l'incrémentation et la décrémentation du compteur $nbFeuillesNoires$ d'un noeud v est effectué par des procédures auxiliaires $incrNbFeuillesNoires(v)$, $decrNbFeuillesNoires(v)$.

Le pseudocode des procédures $devientFeuilleNoire$ et $supprimerFeuille$ est donné ci-dessous :

Observons que les procédures $devientFeuilleNoire$ et $supprimerFeuille$ peuvent s'implémenter de façon à avoir un temps d'exécution constant. Cela est

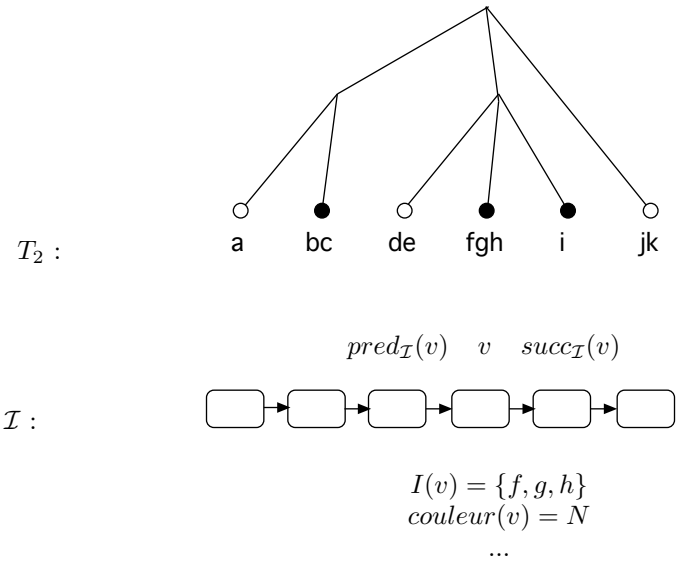


FIG. 4.3 – L'arbre T_2 et la liste $\mathcal{I}$.

---

devientFeuilleNoire(v)
  $couleur(v) \leftarrow noir$
  $FeuillesNoires \leftarrow FeuillesNoires \cup \{v\}$
  $p \leftarrow pere(v)$
  si $p \neq \perp$ alors *incrNbFeuillesNoires*(p)

---

clair pour la procédure *devientFeuilleNoire*. Pour la procédure *supprimerFeuille*, cela est possible à condition d'utiliser un champ auxiliaire dans la structure associée à v, qui maintienne un pointeur vers la cellule correspondante de la liste chaîne $FeuillesNoires$ (de façon à ce que supprimer v de la liste puisse se faire en temps constant).

On décrit maintenant un second invariant. Disons qu'un noeud interne de T_2 est *plein* s'il ne contient que des feuilles noires. On va maintenir l'invariant suivant :

Invariant 4.13. *Au début de chaque appel à* TROUVECONFLIT*, T_2 ne comporte aucun noeud plein.*

Pour maintenir l'invariant 4.13, on fait en sorte que : dès qu'un noeud interne de T_2 devient plein, il est remplacé par une feuille noire. Ceci est détecté par la procédure *incrNbFeuillesNoires*, qui appelle alors une procédure *devientPlein*. Les deux procédures nécessaires au maintien de l'invariant sont les procédures *devientPlein*(v) (v noeud de T_2) et *commenceAppariement*(u) (u noeud de T_1). La procédure *devientPlein*(v), appelée si v est devenu plein, se charge de le remplacer par une unique feuille noire, mettant à jour les listes

---

supprimerFeuille(v)
  {mettre à jour T_2}
  ...
  si *couleur*$(v) =$ *noir* **alors**
    $FeuillesNoires \leftarrow FeuillesNoires - \{v\}$
    $p \leftarrow pere(v)$
    si $p \neq \perp$ alors *decrNbFeuillesNoires*(p)
  fin si

$\mathcal{I}$ et $FeuillesNoires$ en conséquence. La procédure *commenceAppariement*(u), appelée au début de l'appariement de u par *ApparieNoeud*(u), se charge d'initialiser la liste $FeuillesNoires$ en y ajoutant les noeuds $v_1, ..., v_d$ de T_2 correspondant aux fils $u_1, ..., u_d$ de u. Le pseudocode de ces deux procédures est donné ci-dessous :

---

devientPlein(v)
  soient $v_1, ..., v_d$ les fils de v
  $\{v_1, ..., v_d$ forment un intervalle de $\mathcal{I}\}$
  {on les remplace par v}
  $I(v) \leftarrow I(v_1) \cup ... \cup I(v_d)$
  supprimer $v_1, ..., v_d$ de $\mathcal{I}$ et les remplacer par v
  {on supprime les feuilles $v_1, ..., v_d$, on note que v devient feuille noire}
  pour chaque $i \in [d]$, *supprimerFeuille*(v_i)
  devientFeuilleNoire(v)

---

commenceAppariement(u)
  soient $u_1, ..., u_d$ les fils de u
  soit $v_1 = match(u_1), ..., v_d = match(u_d)$ les noeuds de T_2 associés
  $FeuillesNoires \leftarrow \emptyset$
  pour chaque $i \in [d]$, *devientFeuilleNoire*(v_i)

Analysons le temps d'exécution de ces procédures.

Notons que si p est père de v, un appel à la procédure *devientPlein*(v) peut déclencher un appel à la procédure *devientPlein*(u). Appelons temps d'exécution propre d'un appel à *devientPlein*(v) le temps passé dans le corps de la procédure elle-même, ainsi que dans les procédures appelées *supprimerFeuille*, *devientFeuilleNoire*, *incrNbFeuillesNoires*, *decrNbFeuillesNoires*, en excluant le temps pris par un éventuel appel récursif *devientPlein*(u). Une analyse immédiate permet de voir que le temps d'exécution propre d'un appel à *devientPlein*(v) est en $O(d(v))$, où $d(v)$ est le degré de v dans T_2.

De même, appelons temps d'exécution propre d'un appel à la procédure *commenceAppariement*(u) le temps passé dans le corps de la procédure, ainsi que dans les procédures $devientFeuilleNoire, incrNbFeuillesNoires$, sans prendre

en compte le temps pris par un éventuel appel à *devientPlein*. Une analyse immédiate permet de voir que le temps d'exécution propre d'un appel à la procédure *commenceAppariement*(u) est en $O(d(u))$, où $d(u)$ est le degré de u dans T_1.

Sans faire d'hypothèse pour l'instant sur la stratégie de détection des conflits, on représente ci-dessous le processus d'appariement d'un noeud u :

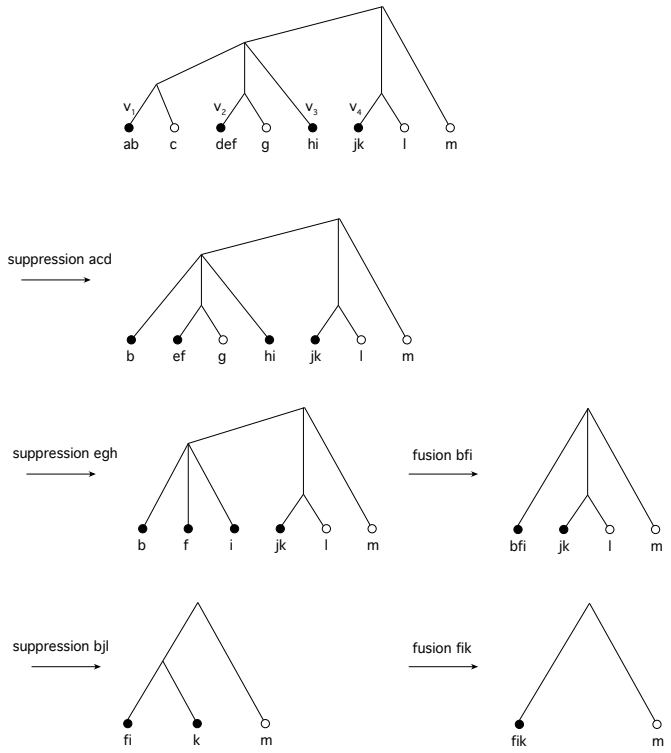


FIG. 4.4 – Le processus d'appariement d'un noeud u ayant quatre fils u_1, u_2, u_3, u_4 avec $I(u_1) = \{a, b\}, I(u_2) = \{d, e, f\}, I(u_3) = \{h, i\}, I(u_4) = \{j, k\}$.

Observons que les invariants 4.12 et 4.13 permettent de détecter quand l'appariement de u a réussi ou échoué : en effet, lorsqu'il reste une seule feuille noire v, nécessairement u doit être apparié à v, et l'appariement a réussi ; au contraire, lorsqu'il ne reste aucune feuille noire, c'est qu'on a supprimé toutes les étiquettes de $I(u)$, et l'appariement a donc échoué. Il reste à justifier que l'on peut identifier un conflit en temps constant lorsque $|FeuillesNoires| \geq 2$. Ceci va être rendu possible par la définition des feuilles *bornes*, qui sont des feuilles noires de T_2 comportant un voisin blanc (pour l'ordre $<_{\mathcal{I}}$) dans une configuration particulière.

Introduisons la définition suivante :

Définition 4.14. *Soit $x \in \mathcal{I}$, et soit $p = pred_{\mathcal{I}}(x)$ et $s = succ_{\mathcal{I}}(x)$. On dit que x est une* borne *si et seulement si x est noire, et l'une des deux conditions suivantes est vérifiée :*

- *soit x est une* borne gauche *: $p \neq \perp$, p blanche et ($s = \perp$ ou $sx|p \notin rt(T_2)$) ;*
- *soit x est une* borne droite *: $s \neq \perp$, s blanche et ($p = \perp$ ou $px|s \notin rt(T_2)$).*

Ainsi, dans la Figure 4.11, v_1, v_2, v_4 sont des bornes droites, tandis que v_3 est une borne gauche. Notons qu'une feuille noire n'est pas nécessairement une borne, par exemple si ses deux voisins sont noirs.

On va voir que les feuilles bornes permettent de détecter un conflit. On a donc besoin de maintenir la liste des feuilles bornes au cours du processus d'appariement :

Invariant 4.15. *Au début de chaque appel à* TROUVECONFLIT*, la liste $FeuillesBornes$ contient les feuilles bornes de T_2.*

Pour maintenir cet invariant, on modifie les deux procédures $devientFeuilleNoire$ et $supprimerFeuille$ de la manière suivante. Dans la procédure $devientFeuilleNoire$, on teste si la nouvelle feuille noire est une borne, et si oui on l'ajoute à $FeuillesBornes$. Dans la procédure $supprimerFeuille$, on examine chaque voisin noir de la feuille supprimée l pour voir s'il peut changer de statut : en effet, il est possible par exemple que le successeur de l ait été une borne gauche avant suppression de l, mais ne soit plus une borne gauche après suppression de l. Notons que tester si une feuille est une borne se fait en temps constant via deux requêtes lca, et donc la mise à jour de la liste $FeuillesBornes$ par les procédures $devientFeuilleNoire$ et $supprimerFeuille$ s'effectue en temps constant.

On donne maintenant une série de lemmes, aboutissant aux lemmes 4.19 et 4.20 qui fournissent une méthode pour identifier un conflit tant que $|FeuillesNoires| \geq 2$.

Appelons *intervalle fort* de $\mathcal{I}$ un intervalle de $\mathcal{I}$ de la forme $L(u)$ pour un certain noeud u de T_2. Les deux lemmes suivants caractérisent les intervalles de $\mathcal{I}$ ne contenant pas d'intervalle fort.

Lemme 4.16. *Soit $I = a_1...a_m$ un intervalle de $\mathcal{I}$ qui ne contient pas d'intervalle fort. Soit $p = pred_{\mathcal{I}}(a_1)$ et $s = succ_{\mathcal{I}}(a_m)$. On a alors l'alternative suivante :*

1. *soit $p \neq \perp$ et $p <_{T_2} \mathsf{lca}_{T_2}(a_1, a_2)$;*
2. *soit $s \neq \perp$ et $s <_{T_2} \mathsf{lca}_{T_2}(a_{m-1}, a_m)$.*

Démonstration. Soit I_1 l'intervalle fort minimum contenant a_1, a_2 (ie $I_1 = L(u_1)$ pour $u_1 = \mathsf{lca}_{T_2}(a_1, a_2)$). Soit I_2 l'intervalle fort minimum contenant a_{m-1}, a_m (ie $I_2 = L(u_2)$ pour $u_2 = \mathsf{lca}_{T_2}(a_{m-1}, a_m)$). Alors I_1, I_2 ne sont pas inclus dans I.

On va montrer qu'on a : soit $p \in I_1$, soit $s \in I_2$. Supposons par contradiction qu'on ait $p \notin I_1$ et $s \notin I_2$. Comme I_1 n'est pas inclus dans I, on a $a_{m-1}, a_m, s \in I_1$. Comme I_2 n'est pas inclus dans I, on a de même $p, a_1, a_2 \in I_2$.

Mais alors $a_1, a_2 \in L(u_2)$ implique que $u_1 \leq_{T_2} u_2$, et $a_{m-1}, a_m \in L(u_1)$ implique que $u_2 \leq_{T_2} u_1$. On conclut que $u_1 = u_2$, et on obtient que $p, s \in I_1 = I_2$, contradiction.

Si $p \in I_1$, on a $p \neq \perp$ et $p <_{T_2} \mathsf{lca}_{T_2}(a_1, a_2)$, et le point 1 est vérifié. Si $p \in I_2$, on a $s \neq \perp$ et $s <_{T_2} \mathsf{lca}_{T_2}(a_{m-1}, a_m)$, et le point 2 est vérifié. On a donc bien l'alternative annoncée. □

Soit une séquence de feuilles $\sigma = v_1, ..., v_m$ de T_2, et pour tout i soit $u_i = parent_{T_2}(v_i)$. On dit que σ est une *brosse ascendante* si et seulement si pour tout i, $u_{i+1} = u_i$ ou $u_{i+1} >_{T_2} u_i$. On dit que σ est une *brosse descendante* si et seulement si pour tout i, $u_{i+1} = u_i$ ou $u_{i+1} <_{T_2} u_i$.

Lemme 4.17. *Soit $I = a_1...a_m$ un intervalle de $\mathcal{I}$ qui ne contient pas d'intervalle fort. Alors il existe $j \leq m$ tel que $a_1, ..., a_j$ est une brosse ascendante et $a_{j+1}, ..., a_m$ est une brosse descendante.*

Démonstration. Pour tout i tel que $1 \leq i \leq m$, définissons $u_i = \mathsf{lca}_{T_2}(a_1, ..., a_i)$ et $v_i = \mathsf{lca}_{T_2}(a_i, ..., a_m)$. Alors $u_1 \leq_{T_2} u_2 \leq_{T_2} ... \leq_{T_2} u_m$ et $v_m \leq_{T_2} v_{m-1} \leq_{T_2} ... \leq_{T_2} v_1$. De plus, u_m, v_1 sont égaux à un même noeud u. Soit i minimal tel que $u_{i+1} = u$.

On montre que $a_1, ..., a_i$ est une brosse ascendante. Soit j fixé, $2 \leq j \leq i$, et soit $p_j = parent_{T_2}(a_j)$, on montre que $p_j = u_j$. On a $p_j \leq_{T_2} u_j$ par définition de u_j. Supposons (absurde) que $p_j <_{T_2} u_j$. Soit $I' = L(p_j)$, alors I' est un intervalle de $\mathcal{I}$. De plus, $a_1 \notin I'$ (sinon $u_j = p_j$) et $a_{i+1} \notin I'$ (sinon $u_j = u_{i+1}$, contredisant la définition de i). Comme $a_j \in I'$, on conclut que $I' \subset I$. Mais alors I contient un intervalle fort, contradiction.

On montre que $a_{i+1}, ..., a_m$ est une brosse descendante par un raisonnement similaire. □

Appelons *intervalle noir* de $\mathcal{I}$ un intervalle maximal de $\mathcal{I}$ constitué de feuilles noires. Le lemme suivant caractérise les intervalles noirs :

Lemme 4.18. *Soit $I = a_1...a_m$ un intervalle noir de $\mathcal{I}$, alors*

1. *l'une des extrémités de I est une borne ;*
2. *il existe $j \leq m$ tel que $a_1, ..., a_j$ est une brosse ascendante et $a_{j+1}, ..., a_m$ est une brosse descendante.*

Démonstration. Observons que I ne contient pas d'intervalle fort : cela contredirait l'invariant 4.13.

Montrons le point 1. Posons $p = pred_{\mathcal{I}}(a_1)$ et $s = succ_{\mathcal{I}}(a_m)$. Par maximalité de I, les feuilles p, s sont toutes deux blanches (à moins qu'ils n'aient la valeur $\perp$). On distingue les cas suivants :

- Premier cas : $m = 1$. Soit $x = a_1$ l'unique élément de $\mathcal{I}$. Comme $|FeuillesNoires| \geq 2$, on ne peut pas avoir p, s tous deux égaux à $\perp$. On a donc les trois cas suivants :
  - $p = \perp, s \neq \perp$: alors s est blanc, et x est une borne droite ;
  - $p \neq \perp, s = \perp$: alors p est blanc, et x est une borne gauche ;

- $p, s \neq \perp$: comme $p <_{\mathcal{I}} x <_{\mathcal{I}} s$, par le lemme 4.11 l'arbre $T_2|\{p, x, s\}$ a trois topologies possibles. Si c'est pxs ou $px|s$, alors x est une borne gauche. Si c'est $p|xs$, alors x est une borne droite.

– Deuxième cas : $m \geq 2$. Comme I ne contient pas d'intervalle fort, le Lemme 4.16 s'applique, et on a l'alternative suivante :
  - soit $p \neq \perp$ et $p <_{T_2} \mathsf{lca}_{T_2}(a_1, a_2)$: alors a_1 est une borne gauche ;
  - soit $n \neq \perp$ et $n <_{T_2} \mathsf{lca}_{T_2}(a_{m-1}, a_m)$: alors a_m est une borne droite.

On conclut que l'une des extrémités de I est une borne.

Le point 2 résulte du Lemme 4.17, qui s'applique puisque I ne contient pas d'intervalle fort. □

Le lemme suivant montre qu'on peut trouver une feuille borne tant que u est non apparié :

Lemme 4.19. *Au début de l'appel à* TrouveConflit*, on a : si* $|FeuillesNoires| \geq 2$*, alors* $FeuillesBornes \neq \emptyset$*.*

Démonstration. Soit I le premier intervalle noir de $\mathcal{I}$, par le Lemme 4.18 l'une des extrémités de I est une borne. □

Le lemme suivant montre qu'on peut identifier un hs-conflit tant que u est non apparié :

Lemme 4.20. *Soit a une feuille borne, et soit b une feuille noire $\neq a$. Soit $p = pred_{\mathcal{I}}(a)$ et $s = succ_{\mathcal{I}}(a)$. Alors :*
- *si a est une borne gauche : alors $\{a, b, p\}$ est un hs-conflit entre T_1, T_2 ;*
- *si a est une borne droite : alors $\{a, b, s\}$ est un hs-conflit entre T_1, T_2.*

Démonstration. Par symétrie, on peut supposer que a est une borne gauche. Alors p est blanche, et on a (i) soit $s = \perp$, (ii) soit $s \neq \perp$ et $sa|p \notin rt(T_2)$. Considérons une feuille noire $b \neq a$, on a soit $b \leq_{\mathcal{I}} p$, soit $b \geq_{\mathcal{I}} s$. Examinons chaque cas :

- si $b \leq_{\mathcal{I}} p$: alors $b <_{\mathcal{I}} p$ (puisque b est noire et p est blanche, on ne peut pas avoir $b = p$). On a alors $b <_{\mathcal{I}} p <_{\mathcal{I}} a$, et par le lemme 4.11 $T_2|\{a, b, p\}$ ne peut pas être l'arbre $ab|p$.
- si $b \geq_{\mathcal{I}} s$: alors $s \neq \perp$, et on a donc $sa|p \notin rt(T_2)$, et par suite $p <_{T_2} \mathsf{lca}_{T_2}(a, s)$. Par le lemme 4.11, on a $\mathsf{lca}_{T_2}(a, s) \leq_{T_2} \mathsf{lca}_{T_2}(a, b)$, et donc $p <_{T_2} \mathsf{lca}_{T_2}(a, b)$. Donc $T_2|\ \{a, b, p\}$ ne peut pas être l'arbre $ab|p$.

Dans chaque cas, on obtient que $T_2|\{a, b, p\}$ est un arbre distinct de $ab|p$, donc $\{a, b, p\}$ est un hs-conflit entre T_1, T_2. □

En utilisant la méthode de détection de conflits fournie par le lemme 4.20, on peut donc donner la description suivante de l'algorithme d'appariement :

Observons qu'à la fin de la procédure ApparierNoeud, les listes $FeuillesNoires$, $FeuillesBornes$ sont réinitialisées à la liste vide, et les compteurs $nbFeuillesNoires$ sont remis à 0.

Justifions que la procédure ApparierNoeud(u) s'exécute en temps $O(d_u + c_u)$, où d_u est le degré de u dans T_1, et c_u le nombre de conflits éliminés par cette

---

TrouveConflit()
  choisir $a \in FeuillesBornes$
  choisir $b \in FeuillesNoires \backslash \{a\}$
  $p \leftarrow pred_{\mathcal{I}}(a), n \leftarrow succ_{\mathcal{I}}(a)$
  si a est une borne gauche **alors**
    renvoyer $\{a, b, p\}$
  sinon si a est une borne droite **alors**
    renvoyer $\{a, b, n\}$
  fin si

ApparierNoeud(u)
  commenceAppariement(u)
  tant que $|FeuillesNoires| \geq 2$ **faire**
    $\{x, y, z\} \leftarrow$ TrouveConflit()
    choisir $l \in I(x), l' \in I(y), l'' \in I(z)$
    supprimer les étiquettes l, l', l''
  fin tant que
  si $|FeuillesNoires| = 0$ **alors**
    {l'appariement a échoué}
    renvoyer $\perp$
  fin si
  si $|FeuillesNoires| = 1$ **alors**
    {l'appariement a réussi}
    soit v l'unique élément de $FeuillesNoires$
    $FeuillesNoires \leftarrow \emptyset$
    $FeuillesBornes \leftarrow \emptyset$
    $p \leftarrow pere(v)$, $nbFeuillesNoires(p) \leftarrow 0$
    renvoyer v
  fin si

---

procédure. Soit S le temps d'exécution de la procédure, alors S est la somme de S_1 (temps d'exécution propre de *commenceAppariement*(u)), S_2 (temps passé dans les procédures ApparierNoeud, TrouveConflit, *supprimerEtiquette* et *supprimerFeuille*) et S_3 (somme des temps d'exécution propres des appels à *devientPlein*).

On a vu que $S_1 = O(d_u)$. Analysons S_2. Chaque appel à TrouveConflit, *supprimerEtiquette* et *supprimerFeuille* prend un temps constant, et le nombre d'appels à ces procédures est $O(c_u)$, donc $S_2 = O(c_u)$. Analysons maintenant S_3. Considérons l'ensemble N des noeuds de T_2 qui deviennent pleins au cours de l'appariemment de u. Pour chaque noeud $v \in N$, notons d'_v son degré au moment où il devient plein. Alors $S_3 = \sum_{v \in N} O(d'_v) = O(d_u)$. On obtient donc que $S = O(d_u + c_u)$ comme annoncé.

On conclut que la procédure TrouveWhsConciliateur(T_1, T_2) s'exécute en temps $O(n + c)$.

4.1.3 Algorithmes polynomiaux et algorithmes FPT

Algorithmes polynomiaux

Le premier algorithme polynomial pour $2-$MAST est dû à [37]. Sa complexité temporelle est $O(n^{4,5}\log n)$, mais [18] montrent qu'une adaptation simple de l'algorithme permet d'abaisser sa complexité à $O(n^{2,5}\log n)$. On décrit maintenant cet algorithme.

Proposition 4.21. $2-$MAST *est soluble en temps* $O(n^{2,5}\log n)$.

Démonstration. Soit $\mathcal{T}=\{T_1,T_2\}$ une collection instance de $2-$MAST. Pour toute paire (u,v) (u noeud de T_1, v noeud de T_2), on calcule $\text{MAST}(u,v)$ par les relations de récurrence suivantes : (i) si u est une feuille de T_1, alors $\text{MAST}(u,v)=1$ si $u\in L(v)$, $\text{MAST}(u,v)=0$ sinon ; (ii) si v est une feuille de T_2, alors $\text{MAST}(u,v)=1$ si $v\in L(u)$, $\text{MAST}(u,v)=0$ sinon ; (iii) si u,v sont des noeuds internes, alors $\text{MAST}(u,v)$ est calculé comme suit. Soit A l'ensemble des fils de u, soit B l'ensemble des fils de v. On calcule les valeurs suivantes :

- $\text{MAST}_1(u,v)$ est le maximum des valeurs $\text{MAST}(u',v)$ pour tout $u'\in A$;
- $\text{MAST}_2(u,v)$ est le maximum des valeurs $\text{MAST}(u,v')$ pour tout $v'\in B$;
- $\text{MAST}_3(u,v)$ est calculé à partir d'un graphe $G(u,v)$ défini comme suit. On construit le graphe biparti pondéré $G(u,v)=(A\cup B,E)$, contenant pour chaque $u'\in A, v'\in B$ une arête $u'v'$ de poids $\text{MAST}(u',v')$. Alors $\text{MAST}_3(u,v)$ est le poids d'un couplage maximum pondéré de $G(u,v)$.

On obtient finalement $\text{MAST}(u,v)$ comme le maximum des valeurs $\text{MAST}_i(u,v)$.

On montre que l'algorithme est correct en prouvant par induction que : pour tout $u\in N(T_1), v\in N(T_2)$, $\text{MAST}(u,v)$ est la taille maximale d'un arbre d'accord de $\{T_1(u),T_2(v)\}$. Les cas (i), (ii) sont clairs. Le cas (iii) se montre en remarquant que S est un arbre d'accord de $\{T_1(u),T_2(v)\}$ si et seulement si on a l'alternative suivante :

- soit S est un arbre d'accord de $\{T_1(u'),T_2(v)\}$ pour un certain $u'\in A$;
- soit S est un arbre d'accord de $\{T_1(u),T_2(v')\}$ pour un certain $v'\in B$;
- soit $S=(S_1,...,S_p)$, et il existe $u_1,...,u_p\in A$ distincts, et $v_1,...,v_p\in B$ distincts, tels que : pour tout i, S_i est un arbre d'accord de $\{T_1(u_i),T_2(v_i)\}$.

Justifions le temps d'exécution de l'algorithme. Soient u,v donnés, considérons le temps nécessaire au calcul de $\text{MAST}(u,v)$. Dans les cas (i) et (ii), ce temps est $O(1)$, en utilisant des structures de données appropriées. Dans le cas (iii), on peut construire le graphe $G(u,v)$ en temps $O(d(u)d(v))$, et on peut trouver un couplage maximum de $G(u,v)$ en temps $O(d(u)d(v)\sqrt{d(u)+d(v)}\log n)$ en utilisant l'algorithme de [21]. En sommant sur les paires (u,v), on obtient un temps d'exécution total en $O(n^{2,5}\log n)$. □

Notons que l'algorithme précédent a un temps d'exécution $O(d^{2,5}n^2\log n)$ si les arbres sont de degré borné.

On présente maintenant des algorithmes pour le problème MAST général, dûs à [4, 11, 17]. Plus précisément, on décrit : (i) un algorithme résolvant MAST en temps $O(kn^3)$ pour les collections d'arbres binaires (Proposition 4.23), (ii) un

algorithme résolvant MAST en temps $O(n^d + kn^3)$ pour les collections d'arbres de degré $\leq d$ (Proposition 4.25).

Les algorithmes reposent sur les définitions suivantes. Soit $\mathcal{T} = \{T_1, ..., T_k\}$ une collection d'arbres sur un ensemble d'étiquettes L. Un *$\mathcal{T}$-tuple* est un tuple $\pi = (u_1, ..., u_k)$, où u_i est un noeud de T_i. On définit la relation $\preceq_{\mathcal{T}}$ sur les $\mathcal{T}$-tuples par : $\pi \preceq_{\mathcal{T}} \pi'$ si et seulement si $\pi[i] \leq_{T_i} \pi'[i]$ pour tout $i \in [k]$.

Etant donné $L' \subseteq L$, on définit le $\mathcal{T}$-tuple $\mathsf{lca}^{\mathcal{T}}(L') = (\mathsf{lca}_{T_1}(L'), ..., \mathsf{lca}_{T_k}(L'))$. Un *$\mathsf{lca}^{\mathcal{T}}$-tuple* est un $\mathcal{T}$-tuple de la forme $\mathsf{lca}^{\mathcal{T}}(\{a, b\})$, qu'on note $\mathsf{lca}^{\mathcal{T}}(a, b)$. Etant donné un arbre T tel que $L(T) \subseteq L$, on notera $\mathsf{lca}^{\mathcal{T}}(T) := \mathsf{lca}^{\mathcal{T}}(L(T))$. Par exemple, pour la collection $\mathcal{T}$ représentée Figure 4.2, on a : $\mathsf{lca}^{\mathcal{T}}(a, b) = (s, w)$ et $\mathsf{lca}^{\mathcal{T}}(S) = \mathsf{lca}^{\mathcal{T}}(a, f) = (r, u)$.

Etant donné un arbre T, une *paire couvrante* de T est une paire d'étiquettes $a, b \in L(T)$ (non nécessairement distinctes) telles que $\mathsf{lca}_T(a, b) = r(T)$. Clairement, tout arbre a une paire couvrante : si T a une unique feuille d'étiquette a, alors aa est une paire couvrante de T ; si $|T| \geq 2$, alors en choisissant des feuilles a, b issues de deux fils distincts de la racine, on obtient que ab est une paire couvrante de T.

L'idée commune aux algorithmes de [17, 11] consiste à calculer par programmation dynamique, pour chaque paire d'étiquettes a, b, la taille d'un plus grand arbre d'accord de $\mathcal{T}$ dont ab est paire couvrante. On choisit de présenter l'algorithme de [11], plus simple.

On expose d'abord l'algorithme dans le cas d'arbres binaires. Etant donnée $\mathcal{T}$, on définit $\mathcal{R} = \cap_i rt(T_i)$. Etant donné $a, b \in L$, notons $AST(a, b)$ l'ensemble des arbres d'accord de $\mathcal{T}$ dont ab est paire couvrante. Le lemme suivant fournit une description récursive de ces ensembles :

Lemme 4.22. *Supposons que $a \neq b$. Les points suivants sont équivalents :*

1. *$S \in AST(a, b)$;*
2. *il existe $x, y \in L$ tels que $S = (S_1, S_2)$, $S_1 \in AST(a, x), S_2 \in AST(y, b)$ avec $ax|b, a|yb \in \mathcal{R}$.*

On est maintenant en mesure de décrire l'algorithme :

Proposition 4.23. MAST *restreint aux collections d'arbres binaires est soluble en temps $O(kn^3)$.*

Démonstration. Soit $\mathcal{T} = \{T_1, ..., T_k\}$ une collection d'arbres binaires sur un ensemble d'étiquettes L. L'algorithme commence par calculer l'ensemble de triplets $\mathcal{R}$. Il calcule ensuite les valeurs $\text{MAST}(a, b)$ par les relations de récurrence suivantes :

$$\text{MAST}(a, a) = 1$$
$$\text{MAST}(a, b) = \max\{\text{MAST}(a, x) + \text{MAST}(y, b) : ax|b, a|yb \in \mathcal{R}\}$$

L'algorithme retourne alors le maximum des valeurs $\text{MAST}(a, b)$ pour $a, b \in L$.

Justifions la terminaison et la correction de l'algorithme. On définit une relation $<$ sur les paires (a, b) par : $(a, b) < (c, d)$ si et seulement si $\mathsf{lca}^{\mathcal{T}}(a, b) \prec_{\mathcal{T}} \mathsf{lca}^{\mathcal{T}}(c, d)$. On montre alors que :

(i) lors du calcul de $\textsc{Mast}(a,b)$, les arguments des appels récursifs sont tels que $(a,x),(y,b) < (a,b)$;
(ii) par induction sur (a,b) (ordonné selon $<$), on montre que $\textsc{Mast}(a,b)$ est la taille d'un plus grand arbre de $AST(a,b)$;
(iii) on montre que la valeur retournée par l'algorithme est effectivement $\textsc{Mast}(\mathcal{T})$.

Les points (i), (iii) ne présentent pas de difficulté, et le point (ii) résulte du Lemme 4.22.

Le temps d'exécution de l'algorithme est $O(kn^3)$. D'une part, la construction de $\mathcal{R}$ s'effectue en temps $O(kn^3)$: elle nécessite l'examen de chaque 3-ensemble $\{x,y,z\}$, qui sont en nombre $O(n^3)$; pour un 3-ensemble donné tester si $xy|z \in rt(T_i)$ s'effectue en temps constant par deux requêtes lca, et donc tester si $xy|z \in \mathcal{R}$ s'effectue en temps $O(k)$. D'autre part, le calcul des valeurs $\textsc{Mast}(a,b)$ par programmation dynamique s'effectue en temps $O(n^3)$: en effet, les paires (a,b) sont en nombre $O(n^2)$; pour une paire (a,b) donnée, on calcule $\textsc{Mast}(a,b)$ en temps $O(n)$. □

On décrit maintenant la généralisation de l'algorithme précédent aux arbres de degré $\leq d$. Etant donnée $\mathcal{T}$, on définit $\mathcal{R} = \cap_i rt(T_i)$ et $\mathcal{F} = \cap_i f(T_i)$. On définit $AST(a,b)$ comme précédemment. On montre alors :

Lemme 4.24. *Les points suivants sont équivalents :*

1. $S \in AST(a,b)$;
2. *il existe* $x,y,z_1,z'_1,...,z_p,z'_p$ *tels que :*
   (i) $ax|b \in \mathcal{R}, a|yb \in \mathcal{R}$,
   (ii) pour tout $i \in [p]$, $az_ib \in \mathcal{F}$ *et* $a|z_iz'_i \in \mathcal{R}$,
   (iii) pour tout $i,j \in [p]$ *distincts,* $az_iz_j \in \mathcal{F}$,
   (iv) $S = (S_0, S_1, ..., S_p, S_{p+1})$ *avec* $S_0 \in AST(a,x), S_{p+1} \in AST(b,y)$ *et pour tout* $i \in [p]$ $S_i \in AST(z_i, z'_i)$.

On montre maintenant :

Proposition 4.25. $\textsc{Mast}$ *restreint aux collections d'arbres de degré* $\leq d$ *est soluble en temps* $O(n^d + kn^3)$ *(si* $d \neq 3$*), en temps* $O(n^4 + kn^3)$ *(si* $d = 3$*).*

Démonstration. Soit $\mathcal{T} = \{T_1, ..., T_k\}$ une collection d'arbres de degré $\leq d$ sur un ensemble d'étiquettes L. L'algorithme commence par calculer l'ensemble de triplets $\mathcal{R} = \cap_{i\in[k]} rt(T_i)$ et l'ensemble de fans $\mathcal{F} = \cap_{i\in[k]} f(T_i)$. Il calcule ensuite les valeurs $\textsc{Mast}(a,b)$ par l'algorithme suivant :

– si $a = b$, renvoyer 1 ;
– si $a \neq b$, alors
  1. construire les ensembles

$$A = \{z \in L : az|b \in \mathcal{R}\}$$
$$B = \{z \in L : bz|a \in \mathcal{R}\}$$
$$C = \{z \in L : azb \in \mathcal{F}\}$$

2. choisir $x^* \in A$ tel que $MAST(a, x^*)$ soit maximum.
3. choisir $y^* \in B$ tel que $MAST(b, y^*)$ soit maximum.
4. pour tout $z \in C$, choisir $z^* \in \{z' \in C : zz'|a \in \mathcal{R}\} \cup \{z\}$, tel que $MAST(z, z^*)$ soit maximum.
5. construire un graphe pondéré G dont l'ensemble de sommets est C, et où deux sommets $z, z' \in C$ sont adjacents si et seulement si $azz' \in \mathcal{F}$. Chaque sommet z reçoit le poids $MAST(z, z^*)$.
6. choisir une clique S de G, de poids maximum.
7. renvoyer

$$MAST(a, x^*) + MAST(b, y^*) + \sum_{z \in S} MAST(z, z^*)$$

Finalement, l'algorithme retourne le maximum des valeurs $\textsc{Mast}(a, b)$ pour $a, b \in L$.

La correction et la terminaison de l'algorithme se montrent de manière analogue à la Proposition 4.23, et reposent sur le Lemme 4.24.

Justifions le temps d'exécution de l'algorithme. La construction de $\mathcal{R}$ et $\mathcal{F}$ s'effectuent en temps $O(kn^3)$. Analysons le temps d'exécution nécessaire au calcul d'une valeur $\textsc{Mast}(a, b)$. Observons tout d'abord que la construction des ensembles A, B, C et le choix de x^*, y^* nécessitent un temps $O(n)$. Donc les étapes (1-3) se font en temps $O(n)$. D'autre part, les étapes (4-5) prennent un temps $O(|C|^2) = O(n^2)$.

Observons que l'étape (6) prend un temps $O(n^{d-2})$. En effet, une clique maximum de G comporte au plus $d-2$ sommets : si $z_1, ..., z_p$ est une clique de G, alors on a (i) pour tout i, $az_ib \in \mathcal{F}$, (ii) pour tous i, j distincts $az_iz_j \in \mathcal{F}$. Ceci implique que pour tout $i \in [k]$, $T_i|\{a, z_1, ..., z_p, b\}$ est un arbre étoilé, et par suite $p \leq d-2$ puisque T_i est de degré $\leq d$. Dès lors, trouver une clique de poids maximum de G s'effectue par énumération des sous-ensembles de C de cardinal $\leq d-2$, et donc en temps $O(n^{d-2})$.

On conclut l'analyse du temps de calcul de $\textsc{Mast}(a, b)$ en distinguant les cas suivants. Si $d = 2$, alors $C = \emptyset$, donc les étapes (4-6) sont sans effet, et le temps total est $O(n)$. Si $d = 3$, alors le terme dominant du temps de calcul est $O(n^2)$ induit par les étapes (4-5). Si $d \geq 4$, alors le terme dominant est $O(n^{d-2})$ induit par l'étape (6).

Le temps d'exécution total de l'algorithme est donc : $O(kn^3)$ si $d = 2$, $O(n^4 + kn^3)$ si $d = 3$, $O(n^d + kn^3)$ si $d \geq 4$. □

Algorithmes FPT

Dans cette section, on décrit des algorithmes FPT pour MAST, pour les paires de paramètres (k, d) et (k, q).

Proposition 4.26. *1.* MAST *est soluble en temps* $O(2^{2kd}n^3)$ *;*

2. MAST *est soluble en temps* $O(2^{O(kq)}n^3)$.

Plutôt que de présenter séparément les deux algorithmes, on choisit d'adopter une présentation unifiée. Cette présentation unifiée repose sur la définition d'une variante *colorée* de MAST, appelée COLORED-MAST, consistant à rechercher un *arbre d'accord coloré*. Commençons par définir le problème.

Soit C un ensemble de couleurs. Un *arbre coloré* sur L est un arbre T sur L muni d'un coloriage $\chi : N(T) \to C$. Une *collection colorée* sur L est une famille $\mathcal{T} = \{T_1, ..., T_k\}$, où chaque T_i est un arbre coloré sur L.

Etant donné un arbre coloré T sur L, et un arbre S sur $L' \subseteq L$, un *plongement coloré* de S dans T est une application $\phi : N(S) \to N(T)$ telle que :

1. ϕ est un plongement de S dans T (vu comme arbre non coloré) ;
2. si u est un noeud de S de fils $u_1, ..., u_d$, alors les noeuds $\mathsf{child}_T(\phi(u), \phi(u_i))$ sont de couleurs distinctes dans T.

On dit que S est un *sous-arbre coloré* de T (noté $S \leq_c T$) si et seulement si il existe un plongement coloré de S dans T. Etant donnée une collection colorée $\mathcal{T} = \{T_1, ..., T_k\}$ sur L et un arbre S sur $L' \subseteq L$, on dit que S est un *arbre d'accord coloré* de $\mathcal{T}$ si et seulement si $S \leq_c T_i$ pour tout i.

On note $AST_c(\mathcal{T})$ l'ensemble des arbres d'accord coloré de $\mathcal{T}$, et $MAST_c(\mathcal{T})$ la taille maximum d'un arbre d'accord coloré de $\mathcal{T}$. Le problème COLORED-MAST consiste, étant donnée une collection colorée $\mathcal{T}$, à trouver un arbre d'accord coloré de taille maximum $MAST_c(\mathcal{T})$.

Ces définitions sont illustrées ci-dessous :

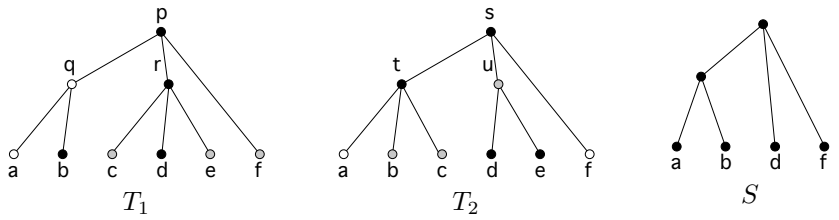


FIG. 4.5 – Une collection colorée $\mathcal{T} = \{T_1, T_2\}$ sur un ensemble d'étiquettes $L = \{a, b, c, d, e, f\}$ et sur un ensemble de trois couleurs (blanc, gris, noir). S est un arbre d'accord coloré de taille maximum.

On va voir que COLORED-MAST peut se résoudre en temps $O(2^{2kc}n^3)$, où $c = |C|$ est le nombre de couleurs (Proposition 4.27).

Avant d'établir ce résultat, on montre qu'il implique la Proposition 4.26.

Preuve de la Proposition 4.26. Point 1 : soit une collection $\mathcal{T} = \{T_1, ..., T_k\}$ sur L où chaque arbre est de degré maximum d. En utilisant d couleurs, on peut colorer chaque arbre T_i de façon à ce que : pour tout noeud interne u, les fils de u soient de couleurs distinctes. On obtient ainsi une collection colorée $\mathcal{T}' = \{T'_1, ..., T'_k\}$ sur L utilisant un ensemble de d couleurs. Il est alors clair qu'un arbre S sur

$L' \subseteq L$ est un arbre d'accord de $\mathcal{T}$ si et seulement si c'est un arbre d'accord coloré de $\mathcal{T}'$. Par conséquent, on peut calculer $MAST(\mathcal{T}) = MAST_c(\mathcal{T}')$ en temps $O(2^{2kd}n^3)$ comme annoncé.

Point 2 : soit une collection $\mathcal{T} = \{T_1, ..., T_k\}$ sur L dont on cherche un arbre d'accord de taille q. Considérons le processus aléatoire suivant : pour tout $i \in [k]$, générer un arbre coloré T'_i à partir de T_i, en colorant chaque noeud avec une couleur aléatoire uniforme dans $[q]$. On obtient une collection colorée $\mathcal{T}' = \{T'_1, ..., T'_k\}$ sur L utilisant un ensemble de q couleurs. On montre alors que :

1. si $MAST(\mathcal{T}) \geq q$, alors $MAST_c(\mathcal{T}') \geq q$ avec probabilité $\geq \frac{1}{e^{kq}}$;
2. si $MAST(\mathcal{T}) < q$, alors $MAST_c(\mathcal{T}') < q$ avec probabilité 1.

Dès lors, considérons l'algorithme suivant pour décider si $MAST(\mathcal{T}) \geq q$. Répéter e^{kq} fois l'expérience suivante : (i) générer une collection colorée $\mathcal{T}'$ par le processus ci-dessus, (ii) tester si $MAST_c(\mathcal{T}') \geq q$. L'algorithme répond positivement si le test a réussi à l'une des étapes, et répond négativement sinon.

Le temps d'exécution de l'algorithme est $O(e^{kq} \times 2^{2kq}n^3) = O(2^{O(kq)}n^3)$, et par l'observation précédente, l'algorithme répond « oui » avec probabilité $\geq \frac{1}{2}$ si $MAST(\mathcal{T}) \geq q$, répond « non » avec probabilité 1 sinon. □

Le reste de la section est consacré à la preuve du résultat suivant :

Proposition 4.27. COLORED-MAST *est soluble en temps* $O(2^{2kc}n^3)$.

Commençons par introduire quelques définitions. Soit un $\mathcal{T}$-tuple π. Un π-*filtre* est un tuple $\phi = (\phi[1], ..., \phi[k])$, où pour tout i, $\phi[i]$ est un sous-ensemble non vide de couleurs des fils de $\pi[i]$. On note $F(\pi)$ l'ensemble des π-filtres. Un π-filtre ϕ est *fin* si $|\phi[i]| = 1$ pour tout i, *épais* si $|\phi[i]| \geq 2$ pour tout i, *idéal* s'il est soit fin, soit épais. Etant donnés deux π-filtres ϕ, ϕ', on note $\phi \sqcup \phi'$ le π-filtre ϕ'' tel que $\phi''[i] = \phi[i] \cup \phi'[i]$ pour tout i ; on note $\phi \sqsubseteq \phi'$ si et seulement si $\phi[i] \subseteq \phi'[i]$ pour tout i.

Etant donné un ensemble $L' \subseteq L$, un arbre coloré T sur L, et un noeud u de T, on définit $\mathsf{cspan}_T(u, L')$ comme l'ensemble des couleurs des fils v de u tels que $L(v) \cap L' \neq \emptyset$. Etant donné un $\mathcal{T}$-tuple π, on définit le π-filtre $\mathsf{cspan}^{\mathcal{T}}(\pi, L') = (\mathsf{cspan}_{T_1}(\pi[1], L'), ..., \mathsf{cspan}_{T_k}(\pi[k], L'))$. On emploiera également les notations $\mathsf{cspan}_T(u, S)$ et $\mathsf{cspan}^{\mathcal{T}}(\pi, S)$ dans le cas où S est un arbre.

Ainsi, dans l'exemple de la Figure 4.5, si l'on note B, N, G les couleurs blanc, noir, gris on voit que :

- si l'on considère le $\mathcal{T}$-tuple $\pi = (p, s)$, alors $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = (\{B, N, G\}, \{B, N, G\})$;
- si l'on considère le $\mathcal{T}$-tuple $\pi = (q, t)$, alors $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = (\{B, N\}, \{B, G\})$.

Etant donné un arbre T, notons $CS(T)$ l'ensemble des sous-arbres de T issus de la racine. Etant donnés deux arbres S_1, S_2 d'ensembles d'étiquettes disjoints, on définit $S_1 \oplus_{p,q} S_2$ pour $p, q \in \{0, 1\}$:

– si $(p,q) = (0,0)$, c'est l'arbre S tel que $CS(S) = \{S_1, S_2\}$;
– si $(p,q) = (0,1)$, c'est l'arbre S tel que $CS(S) = \{S_1\} \cup CS(S_2)$;
– si $(p,q) = (1,0)$, c'est l'arbre S tel que $CS(S) = CS(S_1) \cup \{S_2\}$;
– si $(p,q) = (1,1)$, c'est l'arbre S tel que $CS(S) = CS(S_1) \cup CS(S_2)$.
Cette définition est illustrée ci-dessous :

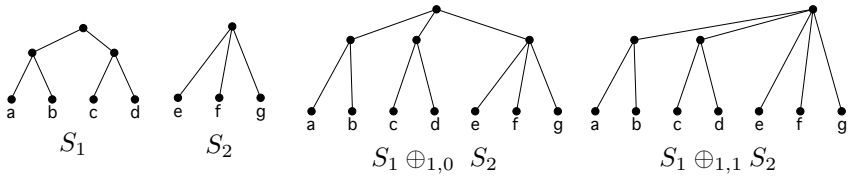


FIG. 4.6 – Deux arbres S_1, S_2 et les arbres $S_1 \oplus_{p,q} S_2$ pour $(p,q) = (1,0), (1,1)$

Etant donné un ensemble S, une (p,q)-*partition* de S est une paire d'ensembles non vides (S_1, S_2) tels que : (i) S_1, S_2 forment une partition de S, (ii) si $(p,q) = (0,0)$, alors $|S_1| = 1, |S_2| = 1$; si $(p,q) = (0,1)$ alors $|S_1| = 1, |S_2| \geq 2$; si $(p,q) = (1,0)$ alors $|S_1| \geq 2, |S_2| = 1$; si $(p,q) = (1,1)$ alors $|S_1| \geq 2, |S_2| \geq 2$. Etant donné un π-filtre ϕ, une (p,q)-*partition* de ϕ est une paire de π-filtres (ϕ_1, ϕ_2) tels que : pour tout $i \in [k]$, $(\phi_1[i], \phi_2[i])$ est une (p,q)-partition de $\phi[i]$.

Le résultat suivant fournit une caractérisation récursive des arbres d'accord colorés.

Lemme 4.28. *Supposons que $S = S_1 \oplus_{p,q} S_2$. Les points suivants sont équivalents :*

1. *$S \in AST_c(\mathcal{T})$;*
2. *$S_1 \in AST_c(\mathcal{T}), S_2 \in AST_c(\mathcal{T})$, et si l'on pose $\pi = \mathsf{lca}^{\mathcal{T}}(S)$, alors $(\mathsf{cspan}^{\mathcal{T}}(\pi, S_1), \mathsf{cspan}^{\mathcal{T}}(\pi, S_2))$ est une (p,q)-partition de $\mathsf{cspan}^{\mathcal{T}}(\pi, S)$.*

Etant donnés une paire $a, b \in L$ induisant un $\mathsf{lca}^{\mathcal{T}}$-tuple π, et un π-filtre ϕ, on définit deux ensembles d'arbres $AST_1(a,b,\phi)$ et $AST_2(a,b,\phi)$:
– $AST_1(a,b,\phi)$ (défini si ϕ est épais) est l'ensemble des arbres S tels que : (i) $S \in AST_c(\mathcal{T})$, (ii) $a, b \in L(S)$, (iii) $\mathsf{lca}^{\mathcal{T}}(S) = \pi$, (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$.
– $AST_2(a,b,\phi)$ (défini si $a \neq b$ et ϕ est idéal) est l'ensemble des arbres S tels que : (i) $S \in AST_c(\mathcal{T})$, (ii) $a \in L(S)$, (iii) $\mathsf{lca}^{\mathcal{T}}(S) \preceq_{\mathcal{T}} \pi$, (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$.
Notons que dans la définition de $AST_1(a,b,\phi)$, les conditions (ii) et (iii) reviennent à exiger que ab soit paire couvrante de S.

Le lemme suivant donne une caractérisation récursive des ensembles $AST_1(a,b,\phi)$.

Lemme 4.29. *Soit $\pi = \mathsf{lca}^{\mathcal{T}}(a,b)$, et soit ϕ un π-filtre épais. Les points suivants sont équivalents :*

1. *$S \in AST_1(a,b,\phi)$;*
2. *il existe $p, q \in \{0,1\}$, (ϕ_1, ϕ_2) (p,q)-partition de ϕ, $S_1 \in AST_2(a,b,\phi_1)$, $S_2 \in AST_2(b,a,\phi_2)$ tels que $S = S_1 \oplus_{p,q} S_2$.*

Démonstration. (1) $\Rightarrow$ (2) : supposons que $S \in AST_1(a,b,\phi)$. Comme a,b sont distincts et $a,b \in L(T)$, on a donc $|S| \geq 2$. On peut donc écrire $S = S_1 \oplus_{p,q} S_2$ avec $pq \in \{0,1\}$, $a \in L(S_1), b \in L(S_2)$. Définissons les π-filtres ϕ_1, ϕ_2 par : $\phi_i = \mathsf{cspan}^{\mathcal{T}}(\pi, S_i)$. Par le lemme 4.28, on a alors $S_1 \in AST_c(\mathcal{T}), S_2 \in AST_c(\mathcal{T})$, et (ϕ_1, ϕ_2) est une (p,q)-partition de ϕ. On montre alors sans difficulté que $S_1 \in AST_2(a,b,\phi_1)$, $S_2 \in AST_2(b,a,\phi_2)$, où les points (ii) résultent du fait que $a \in L(S_1), b \in L(S_2)$, les points (iv) résultent de la définition de ϕ_i, et les autres points sont immédiats.

(2) $\Rightarrow$ (1) : supposons que $S = S_1 \oplus_{p,q} S_2$, avec (ϕ_1, ϕ_2) (p,q)-partition de ϕ et S_1, S_2 vérifiant les hypothèses. Montrons que $S \in AST_1(a,b,\phi)$.

Les points (ii) $a,b \in L(S)$ et (iii) $\mathsf{lca}^{\mathcal{T}}(S) = \pi$ résultent du fait que $a \in L(S_1), b \in L(S_2)$, et que ab est une paire couvrante de S. Le point (iv) résulte du fait que $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \mathsf{cspan}^{\mathcal{T}}(\pi, S_1) \sqcup \mathsf{cspan}^{\mathcal{T}}(\pi, S_2) = \phi_1 \sqcup \phi_2 = \phi$; ici, la première égalité résulte de la définition de $\oplus_{p,q}$, la seconde du fait que $\mathsf{cspan}^{\mathcal{T}}(\pi, S_i) = \phi_i$, et la troisième du fait que (ϕ_1, ϕ_2) est une partition de ϕ.

Le point (i) $S \in AST_c(\mathcal{T})$ résulte du lemme 4.28, dont les hypothèses sont vérifiées :

- $S_1 \in AST_c(\mathcal{T})$ (car $S_1 \in AST_2(a,b,\phi_1)$) et de même $S_2 \in AST_c(\mathcal{T})$;
- $\pi = \mathsf{lca}^{\mathcal{T}}(S)$ (car ab est une paire couvrante de S) ;
- $\mathsf{cspan}^{\mathcal{T}}(\pi, S_i) = \phi_i$, $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$ (cf ci-dessus), et (ϕ_1, ϕ_2) est une (p,q)-partition de ϕ (par hypothèse).

On conclut que $S \in AST_1(a,b,\phi)$. □

Le lemme suivant donne une caractérisation récursive des ensembles $AST_2(a,b,\phi)$. Soit $\phi = (\{x_1\}, ..., \{x_k\})$ un π-filtre fin. On note $S(\pi,\phi)$ l'ensemble des $\mathcal{T}$-tuples π' tels que pour tout $i \in [k]$, $\pi'[i] <_{T_i} \pi[i]$ et $\mathsf{child}_{T_i}(\pi[i], \pi'[i])$ est de couleur x_i.

Lemme 4.30. *Soit $\pi = \mathsf{lca}^{\mathcal{T}}(a,b)$, et soit ϕ un π-filtre idéal. Les points suivants sont équivalents :*

1. *$S \in AST_2(a,b,\phi)$;*
2. *il existe $c \in L$ tel que a,c induise un $\mathsf{lca}^{\mathcal{T}}$-tuple π' tel que :*
   - *si ϕ est épais alors : (a) $\pi' = \pi$ et (b) $S \in AST_1(a,c,\phi)$;*
   - *si ϕ est fin alors : (a) $\pi' \in S(\pi,\phi)$, (b) il existe ϕ' π'-filtre tel que $S \in AST_1(a,c,\phi')$.*

Démonstration. (1) $\Rightarrow$ (2) : supposons que (1) $S \in AST_2(a,b,\phi)$. Alors S admet une paire couvrante ac (avec c éventuellement égal à a). Posons $\pi' = \mathsf{lca}^{\mathcal{T}}(a,c)$.

Premier cas : ϕ est épais. Alors $\pi' = \pi$, et donc (a) est vérifiée. Montrons que (b) l'est. Le point (i) $S \in AST_c(\mathcal{T})$ résulte de (1). Les points, (ii) $a,c \in L(S)$, (iii) $\mathsf{lca}^{\mathcal{T}}(S) = \pi'$ résultent de la définition de c. Le point (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi', S) = \phi$ résulte de (1) et du fait que $\pi = \pi'$. On conclut que $S \in AST_1(a,c,\phi)$.

Deuxième cas : ϕ est fin. Alors $\phi = (\{x_1\}, ..., \{x_k\})$. Par (1), on a $S \in AST_c(\mathcal{T})$ et $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$. Donc pour tout $i \in [k]$ on a $\pi'[i] <_{T_i} \pi[i]$ et $\mathsf{child}_{T_i}(\pi[i], \pi'[i])$ est de couleur x_i : on obtient que (a) $\pi' \in S(\pi,\phi)$ est vérifiée. Soit $\phi' = \mathsf{cspan}^{\mathcal{T}}(\pi', S)$, alors ϕ' est un π'-filtre pour lequel (b) est vérifiée.

Le point (i) $S \in AST_c(\mathcal{T})$ résulte de (1). Les points (ii) $a, c \in L(S)$ et (iii) $\mathsf{lca}^{\mathcal{T}}(S) = \pi'$ résultent de la définition de c. Le point (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi', S) = \phi'$ résulte de la définition de ϕ'. On conclut que $S \in AST_1(a, c, \phi')$.

(2) $\Rightarrow$ (1) : supposons qu'il existe $c \in L$ vérifiant les hypothèses.

Premier cas : ϕ est épais. Alors (a) $\pi' = \pi$ et (b) $S \in AST_1(a, c, \phi)$. Montrons que $S \in AST_2(a, b, \phi)$. Les points (i) $S \in AST_c(\mathcal{T})$, (ii) $a \in L(S)$ résultent de (b). Les points (iii) $\mathsf{lca}^{\mathcal{T}}(S) \preceq_{\mathcal{T}} \pi$ et (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$ résultent de (b) et du fait que $\pi = \pi'$.

Deuxième cas : ϕ est fin. Alors $\phi = (\{x_1\}, ..., \{x_k\})$, et (a) $\pi' \in S(\pi, \phi)$, (b) il existe ϕ' π'-filtre tel que $S \in AST_1(a, c, \phi')$. Montrons que $S \in AST_2(a, b, \phi)$. Les points (i) $S \in AST_c(\mathcal{T})$, (ii) $a \in L(S)$ résultent de (b). Le point (iii) $\mathsf{lca}^{\mathcal{T}}(S) \preceq_{\mathcal{T}} \pi$ résulte de (b) et du fait que $\pi' \preceq_{\mathcal{T}} \pi$. Le point (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$ résulte de (a) : pour chaque $i \in [k]$, on a $\mathsf{cspan}_{T_i}(\pi[i], S) = \{x_i\} = \phi[i]$. □

On est maintenant en mesure de montrer :

Preuve de la Proposition 4.27. Définissons $MAST_i(a, b, \phi)$ comme la taille d'un plus grand arbre de $AST_i(a, b, \phi)$. Soit $a, b \in L$, $\pi = \mathsf{lca}^{\mathcal{T}}(a, b)$ et ϕ un π-filtre. Alors les lemmes 4.29 et 4.30 fournissent les relations de récurrence suivantes :

$$\begin{cases} \text{si } a = b : & MAST_1(a, b, \phi) = 1 \\ \text{si } \ a \neq b : & MAST_1(a, b, \phi) = \\ & \max\{MAST_2(a, b, \phi_1) + MAST_2(b, a, \phi_2) : \exists p, q \in \{0, 1\} \text{ tq} \\ & (\phi_1, \phi_2)\ (p, q)\text{-partition de } \phi\} \end{cases} \tag{4.1}$$

$$\begin{cases} \phi \text{ épais} : & MAST_2(a, b, \phi) = \\ & \max\{MAST_1(a, c, \phi) : c \in L \text{ tq } \mathsf{lca}^{\mathcal{T}}(a, c) = \pi\} \\ \phi \text{ fin} : & MAST_2(a, b, \phi) = \\ & \max\{MAST_1(a, c, \phi') : c \in L \text{ tq si } \pi' = \mathsf{lca}^{\mathcal{T}}(a, c) \\ & \ \text{alors } \pi' \in S(\pi, \phi) \text{ et } \phi' \in F(\pi')\} \end{cases} \tag{4.2}$$

Ces équations fournissent un algorithme pour Colored-Mast, consistant à calculer les valeurs $MAST_1(a, b, \phi)$ et $MAST_2(a, b, \phi)$, et à obtenir $MAST_c(\mathcal{T})$ comme le maximum des valeurs $MAST_1(a, b, \phi)$ pour $a, b \in L$, ϕ $\mathsf{lca}^{\mathcal{T}}(a, b)$-filtre. Les valeurs sont calculées par programmation dynamique sur les tuples (a, b, ϕ), ordonnés par la relation suivante :

$$(a, b, \phi) < (a', b', \phi') \text{ si et seulement si } \mathsf{lca}^{\mathcal{T}}(a, b) \prec_{\mathcal{T}} \mathsf{lca}^{\mathcal{T}}(a', b') \text{ ou } (\mathsf{lca}^{\mathcal{T}}(a, b) = \mathsf{lca}^{\mathcal{T}}(a', b') \text{ et } \phi \sqsubset \phi')$$

On montre à présent que le temps d'exécution de l'algorithme est $O(2^{2kc} n^3)$.

Le calcul d'une valeur $MAST_1(a, b, \phi)$ par l'équation (4.1) nécessite d'examiner les paires (ϕ_1, ϕ_2) formant une partition de ϕ, donc ce calcul se fait

en temps $O(2^{kc})$. Le calcul d'une valeur $MAST_2(a, b, \phi)$ par l'équation (4.2) nécessite d'examiner chaque sommet $c \in L$, et pour c fixé d'examiner chaque $\mathsf{lca}^{\mathcal{T}}(a, c)$-filtre ϕ', donc ce calcul se fait en temps $O(2^{kc}n)$.

Comme le nombre de tuples (a, b, ϕ) à examiner lors de la programmation dynamique est $O(2^{kc}n^2)$, et comme pour chaque tuple les valeurs $MAST_1, MAST_2$ sont calculées en temps $O(2^{kc}n)$, on conclut que l'étape de programmation dynamique prend un temps total $O(2^{2kc}n^3)$. Le calcul de $MAST_c(\mathcal{T})$ à la fin de l'algorithme nécessite un temps $O(2^{kc}n^2)$. Au total, le temps d'exécution de l'algorithme est donc $O(2^{2kc}n^3)$. □

4.1.4 Résultats négatifs

Approximabilité

On a vu dans la section précédente que MAST est polynomial pour des arbres de degré borné. On donne maintenant des résultats d'inapproximabilité pour le problème lorsque les arbres sont de degré non borné. On montre que : (i) le problème MAST pour un nombre d'arbres non borné est aussi difficile à approximer que MAXIMUM INDEPENDENT SET (Proposition 4.31), (ii) le problème $3-$MAST est APX-dur (Proposition 4.32) et n'est pas approximable à un facteur constant (Proposition 4.33), (iii) le problème complémentaire $3-$CMAST est APX-dur (Proposition 4.34). Ces résultats sont dûs à [4, 10, 7].

Proposition 4.31 ([10]). MAST *n'est pas approximable à* $n^{1-\epsilon}$ *si* $\mathsf{P} \neq \mathsf{NP}$.

Démonstration. Rappelons que problème MIS n'est pas approximable à $n^{1-\epsilon}$ si $\mathsf{P} \neq \mathsf{NP}$: il s'agit d'un résultat établi par [28] sous l'hypothèse $\mathsf{NP} \not\subseteq \mathsf{ZPP}$, et dérandomisé par [38]. Il suffit donc de donner une réduction linéaire de MIS à MAST. Soit $G = (V, E)$ une instance de MIS, on construit la collection $\mathcal{T}$ sur l'ensemble d'étiquettes V comportant les arbres suivants :

- l'arbre S est un arbre en étoile sur V,
- pour chaque $e = \{x, y\} \in E$, l'arbre T_e dont la racine a les fils suivants : (i) pour chaque $z \in V \backslash e$, une feuille d'étiquette z, (ii) un noeud interne ayant pour fils deux feuilles d'étiquettes x, y.

La correction de la réduction résulte du fait que : pour tout $V' \subseteq V$ de cardinal ≥ 3, V' est un indépendant de G si et seulement si $\mathcal{T}|V'$ est isomorphe.

Soit $V' \subseteq V$ de cardinal ≥ 3 tel que V' soit un indépendant de G. Soit T un arbre en étoile sur l'ensemble d'étiquettes V', alors T est un arbre d'accord de $\mathcal{T}$: $T \leq S$ est clair, et pour chaque $e \in E$, $T \leq T_e$ résulte du fait qu'au plus une des extrémités de e est dans V', et donc $T_e|L(T)$ est un arbre en étoile. On conclut que $\mathcal{T}|V'$ est isomorphe.

Soit $V' \subseteq V$ de cardinal ≥ 3 tel que $\mathcal{T}|V'$ est isomorphe. Soit $e = \{x, y\} \in E$, on montre que V' contient au plus une extrémité de e : en effet, si l'on avait $x, y \in V'$, en considérant $z \in V'$ distinct de x, y on obtient que $xyz \in f(S), xy|z \in rt(T_e)$, contredisant le fait que $\mathcal{T}|V'$ est isomorphe. On conclut que V' est un indépendant de G. □

Proposition 4.32 ([4, 10]). $3-$MAST *est* APX*-dur.*

Démonstration. Rappelons la définition du problème THREE DIMENSIONAL MATCHING (3DM) : étant donnés des ensembles disjoints V_1, V_2, V_3, un ensemble de triplets $S \subseteq V_1 \times V_2 \times V_3$, chercher un ensemble $S' \subseteq S$ de cardinal maximal tels que deux triplets distincts de S' n'aient aucune composante égale. On considère la restriction du problème 3DM aux instances telles que : pour tout $i \in \{1,2,3\}$, chaque élément de V_i apparaisse dans au plus Δ triplets. On note ce problème $3\text{DM}-\Delta$.

On donne une L-réduction de $3\text{DM}-\Delta$ à $3-$MAST. Considérons une instance de $3\text{DM}-\Delta$, consistant en V_1, V_2, V_3 et un ensemble de n triplets $S \subseteq V_1 \times V_2 \times V_3$. On construit une collection $\mathcal{T} = \{T_1, T_2, T_3\}$ sur l'ensemble d'étiquettes $L = S \cup L'$, où $L' = \{x_1, ..., x_n\}$ est un ensemble de n nouvelles étiquettes. L'arbre T_i est défini comme suit :

- sa racine a pour fils : (i) les feuilles d'étiquettes $x_1, ..., x_n$, (ii) les noeuds u_x^i pour chaque $x \in V_i$;
- chaque noeud u_x^i a pour fils : les feuilles d'étiquette t pour chaque $t \in S$ tel que $t[i] = x$.

L'observation suivante met en relation les solutions de 3DM (pour l'instance S) et de $3-$MAST (pour l'instance $\mathcal{T}$).

Fait. (i) si $S' \subseteq S$ est une solution de 3DM, alors il existe T arbre d'accord de $\mathcal{T}$ tel que $|T| \geq |S'| + n$;
(ii) si T est un arbre d'accord de $\mathcal{T}$, alors il existe $S' \subseteq S$ solution de 3DM tel que $|S'| \geq |T| - n$.
(iii) $\mathsf{opt}' = \mathsf{opt} + n$.

Preuve. Point (i) : considérons l'arbre T dont la racine a pour fils : (i) les feuilles d'étiquettes $x_1, ..., x_n$, (ii) les feuilles d'étiquettes t pour chaque $t \in S'$. Il n'est pas difficile de voir que T est un arbre d'accord de $\mathcal{T}$ tel que $|T| \geq |S'| + n$. Point (ii) : Le résultat est clair si $|T| \leq n$. Supposons maintenant que $|T| > n$. Comme $|S| = n$, $L(T)$ contient alors l'une des étiquettes x_i. Soit $S' = L(T) \backslash \{x_1, ..., x_n\}$, alors $|S'| \geq |T| - n$. Montrons que S' est une solution de 3DM. Soient $t, t' \in S'$, supposons par contradiction que $t[p] = t'[p]$ pour un certain p. Comme $t \neq t'$, il existe $q \neq p$ tel que $t[q] \neq t'[q]$. On obtient que $x_i|tt' \in rt(T_p), x_i tt' \in f(T_q)$, contredisant le fait que T est un arbre d'accord de $\mathcal{T}$. Le point (iii) résulte des points (i) et (ii) de façon immédiate. □

Justifions maintenant que la réduction est une L-réduction :

- il existe α tel que $\mathsf{opt}' \leq \alpha \times \mathsf{opt}$: en effet, on a $\mathsf{opt} \geq \frac{n}{\Delta}$ du fait que chaque élément apparaît dans au plus Δ triplets. Comme $\mathsf{opt}' = n + \mathsf{opt}$, on obtient que $\mathsf{opt}' \leq (\Delta + 1) \times \mathsf{opt}$.
- il existe β tel que : à partir d'une solution T de MAST, on peut construire en temps polynomial une solution S' de 3DM telle que $|\mathsf{opt} - \mathsf{m}(S')| \leq \beta \times |\mathsf{opt}' - \mathsf{m}'(T)|$. En effet, soit T un arbre d'accord de $\mathcal{T}$, par l'observation ci-dessus on peut construire en temps polynomial S' solution de 3DM telle

que $|S'| \geq |T| - n$. On a donc : $\mathsf{opt}'(T) - \mathsf{m}'(T) = \mathsf{opt}' - |T| \geq \mathsf{opt} + n - |S'| - n = \mathsf{opt} - \mathsf{m}(S')$.

On a donc bien une L-réduction. On conclut en observant que 3DM − 3 est APX-dur [31]. □

Proposition 4.33 ([10]). *On a les résultats d'inapproximabilité suivants pour* 3 − MAST *: (i) inapproximabilité à un facteur constant sauf si* P = NP*, (ii) inapproximabilité à* $2^{(\log n)^\alpha}$ *pour tout* $\alpha < 1$*, sauf si* $\mathsf{NP} \subseteq DTIME(2^{\mathsf{polylog}(n)})$.

Démonstration. La preuve emploie la technique d'*auto-amélioration*, utilisée fréquemment [30, 2, 5, 3, 1, 27, 15] pour amplifier le seuil d'inapproximabilité d'un problème de maximisation.

Pour mettre en oeuvre cette technique, on définit le *produit* de deux arbres étiquetés. Etant donnés deux arbres étiquetés T, T', l'arbre $T \otimes T'$ est l'arbre étiqueté sur $L(T) \times L(T')$ défini comme suit :

- si l est une étiquette, alors $l \otimes T'$ est l'arbre sur $\{l\} \times L(T')$ obtenu en remplaçant chaque étiquette l' de T par l'étiquette (l, l') ;
- $T \otimes T'$ est l'arbre sur $L(T) \times L(T')$ obtenu à partir de T en substituant chaque feuille d'étiquette l par l'arbre $l \otimes T'$.

On définit également le produit de deux collections de même cardinal k. Etant données deux collections $\mathcal{T} = \{T_1, ..., T_k\}, \mathcal{T}' = \{T'_1, ..., T'_k\}$, on pose $\mathcal{T} \otimes \mathcal{T}' = \{T_1 \otimes T'_1, ..., T_k \otimes T'_k\}$. On a alors la propriété suivante :

Fait.

(i) $S \leq T \otimes T'$ si et seulement si il existe $x_1, ..., x_m \in L(T)$, un arbre $S_0[X_1, ..., X_m]$ et des arbres $S_1, ..., S_m$ tels que (i) $S_0[x_1, ..., x_m] \leq T$, (ii) pour tout $i \in [m]$, $S_i \leq T'$, (iii) $S = S_0[x_1 \otimes S_1, ..., x_m \otimes S_m]$.

(ii) si S est un arbre d'accord de $\mathcal{T}$ et si S' est un arbre d'accord de $\mathcal{T}'$, alors $S \times S'$ est un arbre d'accord de $\mathcal{T} \otimes \mathcal{T}'$;

(iii) si S'' est un arbre d'accord de $\mathcal{T} \otimes \mathcal{T}'$, alors on peut obtenir en temps polynomial S arbre d'accord de $\mathcal{T}$, S' arbre d'accord de $\mathcal{T}'$ tel que $|S''| \leq |S| \times |S'|$;

(iv) $MAST(\mathcal{T} \otimes \mathcal{T}') = MAST(\mathcal{T}) \times MAST(\mathcal{T}')$.

Preuve. Point (i) : clair.

Point (ii) : soit $i \in [k]$, alors $S \leq T_i, S' \leq T'_i$. On a $L(S) = \{x_1, ..., x_m\}$, et donc $S = C[x_1, ..., x_m]$. Alors $S \otimes S' = C[x_1 \otimes S', ..., x_m \otimes S']$ est tel que $S \otimes S' \leq T_i \otimes T'_i$ par le point (i). On conclut que $S \otimes S'$ est un arbre d'accord de $\mathcal{T}$.

Point (iii) : par le point (i), il existe un arbre $S_0[X_1, ..., X_m]$ et des arbres $S_1, ..., S_m$ tels que $S'' = S_0[x_1 \otimes S_1, ..., x_m \otimes S_m]$. Posons $S = S_0[x_1, ..., x_m]$, et choisissons S' égal à l'arbre S_i ($i \in [m]$) de taille maximale. Pour tout $i \in [k]$, on a alors $S \leq T_i, S' \leq T'_i$. On obtient que S est un arbre d'accord de $\mathcal{T}$, S' est un arbre d'accord de $\mathcal{T}'$, et $|S''| \leq |S| \times |S'|$.

Point (iv) : résulte des points (ii) et (iii). □

Justifions à présent comment cette construction permet d'établir les résultats d'inapproximabilité annoncés.

Point (i) : Supposons que le problème soit approximable à un facteur constant ρ en temps polynomial $O(n^c)$. Soit une constante $\delta > 1$ arbitraire, considérons l'algorithme de δ-approximation suivant pour $3-$MAST. Etant donnée une collection $\mathcal{T}$:

- choisir une constante k telle que $\rho^{1/k} \leq \delta$ (par exemple $k = \lceil \frac{\log \rho}{\log \delta} \rceil$) ;
- calculer une ρ-approximation S de $3-$MAST sur l'instance $\mathcal{T}^k$;
- obtenir un arbre d'accord S' de $\mathcal{T}$ tel que $|S'| \geq |S|^{1/k}$ (cf preuve du point iv).

L'algorithme calcule bien une δ-approximation, puisque l'arbre S' obtenu est tel que :

$$|S'| \geq |S|^{1/k} \geq (\frac{MAST(\mathcal{T}^k)}{\rho})^{1/k} \geq \frac{MAST(\mathcal{T})}{\delta}$$

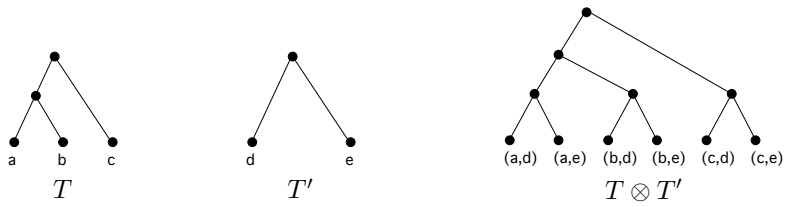
où la deuxième inégalité résulte du fait que S est une ρ-approximation, et la troisième inégalité résulte du point (iv) et du fait que $\rho^{1/k} \leq \delta$. On obtient donc un algorithme de δ-approximation pour $3-$MAST de complexité temporelle $O(n^{kc})$, donc polynomiale pour δ fixé. Mais l'APX-difficulté de $3-$MAST implique qu'il est NP-dur d'approximer le problème à une certaine constante $\delta > 1$. Donc $\mathsf{P} = \mathsf{NP}$.

Point (ii) : Supposons que le problème soit approximable en temps polynomial à un facteur $2^{(\log n)^\alpha}$, pour un certain $\alpha < 1$. Soit une constante $\delta > 1$ arbitraire, considérons l'algorithme de δ-approximation suivant pour $3-$MAST. Etant donnée une collection $\mathcal{T}$:

- choisir k tel que $2^{k^{\alpha-1}(\log n)^\alpha} < \delta$ (par exemple $k = \lceil (\frac{(\log n)^\alpha}{\log \delta})^{\frac{1}{1-\alpha}} \rceil$;
- procéder comme ci-dessus.

Par le même argument que précédemment, l'algorithme calcule une δ-approximation pour $3-$MAST, en temps $O(n^{kc}) = 2^{O((\log n)^{1+\frac{\alpha}{1-\alpha}})}$. En considérant δ pour lequel $3-$MAST est NP-dur à approximer à un facteur δ, on conclut que $\mathsf{P} \subseteq DTIME(2^{\mathsf{polylog}(n)})$. □

On a représenté ci-dessous un exemple de produit de deux arbres, au sens de la notation $\otimes$:



T T' $T \otimes T'$

FIG. 4.7 – Deux arbres T, T' et l'arbre $T \otimes T'$

On considère maintenant le problème complémentaire CMAST. Observons que la preuve de la Proposition 4.31 fournit également une réduction linéaire de MVC (problème complémentaire de MIS) à CMAST, et établit donc que CMAST est APX-dur. On montre à présent :

Proposition 4.34 ([7]). $3-$CMast *est* APX*-dur.*

Démonstration. On montre que la réduction utilisée dans la preuve de la Proposition 4.32 est une L-réduction de $3\text{DM}-\Delta$ à $3-$CMast. Notons qu'on a maintenant $\mathsf{m}'(T) = 2n - |T|$ si T est un arbre d'accord de $\mathcal{T}$, et $\mathsf{opt}' = 2n - (\mathsf{opt} + n) = n - \mathsf{opt}$. On montre que :

- il existe α tel que $\mathsf{opt}' \leq \alpha \times \mathsf{opt}$: en effet, on a $\mathsf{opt} \geq \frac{n}{\Delta}$. Comme $\mathsf{opt}' = n - \mathsf{opt}$, on obtient que $\mathsf{opt}' \leq (\Delta - 1) \times \mathsf{opt}$.
- il existe β tel que : à partir d'une solution T de CMast, on peut construire en temps polynomial une solution S' de 3DM telle que $|\mathsf{opt} - \mathsf{m}(S')| \leq \beta \times |\mathsf{opt}' - \mathsf{m}'(T)|$. Par le même argument que dans la preuve de la Proposition 4.32, si T est un arbre d'accord de $\mathcal{T}$ on peut construire en temps polynomial S' solution de 3DM telle que $|S'| \geq |T| - n$. On obtient : $\mathsf{m}'(T) - \mathsf{opt}' = 2n - |T| - \mathsf{opt}' \geq (n - |S|) - (n - \mathsf{opt}) = \mathsf{opt} - \mathsf{m}(S')$.

On a donc bien une L-réduction. □

Complexité paramétrique.

On considère maintenant le degré maximum d. On utilise les outils de la Section 2.3.3 en montrant que Mast$[d, q]$ est $\mathsf{W_I}[1]$-dur (Proposition 4.37). D'après la Proposition 2.18, cela implique que Mast n'est pas soluble en temps $\Phi(d)N^{o(d)}$ sous l'hypothèse ETH (où N est la taille de l'instance). Ceci suggère donc l'optimalité asymptotique de l'algorithme décrit en Proposition 4.25.

Pour les besoins de la réduction, on introduit une variante du problème Partitioned Independent Set (défini dans le Chapitre 3). Ce problème, nommé Restricted Partitioned Independent Set (RPis), est défini comme suit : étant donné un entier k et un graphe k-parti *sans sommets isolés* G, décider si G a un indépendant partitionné.

On montre dans un premier temps :

Lemme 4.35. Pis$[k]$ *est* $\mathsf{W_I}[1]$*-dur.*

Lemme 4.36. RPis$[k]$ *est* $\mathsf{W_I}[1]$*-dur.*

Démonstration. On donne une fpt-réduction linéaire depuis Pis$[k]$. Soit $I = (G, k)$ une instance de Pis$[k]$, où G est un graphe k-parti de partition $V_1, ..., V_k$. On construit l'instance $I' = (G', k)$ de RPis$[k]$, où G' est obtenu à partir de G en ajoutant un nouveau sommet w_i à chaque ensemble V_i, et pour chaque i des arêtes de w_i vers chaque élément de V_{i+1} (où $k + 1 = 1$ par convention). Le graphe G' ainsi obtenu est sans sommets isolés, et on vérifie que G a un indépendant partitionné si et seulement si G' a un indépendant partitionné.

Clairement, si $I = \{v_1, ..., v_k\}$ est un indépendant partitionné de G, alors I est également un indépendant partitionné de G'. Supposons maintenant que $I = \{v_1, ..., v_k\}$ est un indépendant partitionné de G'. Alors pour tout $i \in [k]$, v_i est distinct de w_i, car sinon G' contiendrait l'arête $v_i v_{i+1}$. On conclut que I est un indépendant partitionné de G. □

On décrit maintenant le résultat de difficulté annoncé :

Proposition 4.37. MAST$[d, q]$ *est* $\mathsf{W}[1]$*-dur.*

Démonstration. On donne une fpt-réduction linéaire depuis RPIS$[k]$. La réduction envoie chaque instance $I = (G, k)$ de RPIS$[k]$ sur une instance $I' = (\mathcal{T}, k', q)$ de MAST$[d]$ avec $k' = k + 1$ et $q = k$. Cette réduction est définie seulement pour $k \geq 3$ (puisque le cas $k < 3$ peut être vérifié en temps polynomial).

Construction. Soit $G = (V, E)$ un graphe k-parti, de partition $V_1, ..., V_k$. On construit une instance de MAST de la façon suivante. L'ensemble d'étiquettes est $L := V$. La collection est $\mathcal{T} := \{P\} \cup \{Q_e : e \in E\}$, où P est la *composante de contrôle* et les Q_e sont les *composantes de sélection*. Ces arbres sont construits comme suit. Pour chaque i, soit $<_{V_i}$ un ordre total sur V_i, et soit $R_i := rake(V_i, <_{V_i})$. Alors $P = (R_1, ..., R_k)$. Pour tout $e = xy \in E$, Q_e est obtenu à partir de P en supprimant x, y, et en ajoutant l'arbre $S_{x,y} := (x, y)$ comme fils de la racine.

Correction. Clairement, la construction se fait en temps polynomial, et les arbres de $\mathcal{T}$ sont de degré maximum $\leq k'$. Il reste à justifier que : G a un indépendant partitionné si et seulement si $MAST(\mathcal{T}) \geq k$.

Supposons que $I = \{v_1, ..., v_k\}$ est un indépendant partitionné de G, avec $v_i \in V_i$ pour tout i. Alors $T = (v_1, ..., v_k)$ est un arbre d'accord de $\mathcal{T}$. En effet, il est clair que $P|L = T$; d'autre part, pour chaque $e \in E$, comme I est un indépendant e contient au plus un élément v_i, et donc $Q_e|L = T$.

Supposons que T est un arbre d'accord de $\mathcal{T}$ de taille k. Observons que : Pour tout i, $|L(T) \cap V_i| \leq 1$. Par contradiction, supposons que $L(T) \cap V_i$ contienne deux éléments x, y. Comme $|L(T)| = k \geq 3$, on peut trouver $z \in L(T)$ distinct de x, y. De plus (comme G n'a pas de sommet isolé) on peut trouver une arête $e = xw \in E$ avec $w \neq y$. On a alors $xy|z \in rt(P)$, tandis que $xz|y \in rt(Q_e)$ ou $yz|x \in rt(Q_e)$ ou $xyz \in f(Q_e)$: donc x, y, z induisent un conflit entre $\mathcal{T}$, contredisant le fait que T est un arbre d'accord.

Par conséquent, comme $|L(T)| = k$, on doit avoir $|L(T) \cap V_i| = 1$ pour tout i. Notons v_i l'unique élément de $L(T) \cap V_i$, et considérons l'ensemble $I = \{v_1, ..., v_k\}$. Alors I est un indépendant partitionné de G. En effet, on a $v_i \in V_i$ pour tout i. De plus, G ne peut pas contenir d'arête $e = v_i v_j$: si c'était le cas, puisque $k \geq 3$ on pourrait trouver $v_p \neq v_i, v_j \in L(T)$, et on obtiendrait $x_i x_j x_p \in f(P)$, $x_i x_j | x_p \in rt(Q_e)$, contredisant le fait que T est un arbre d'accord. □

Le principe de la réduction est illustré ci-dessous :

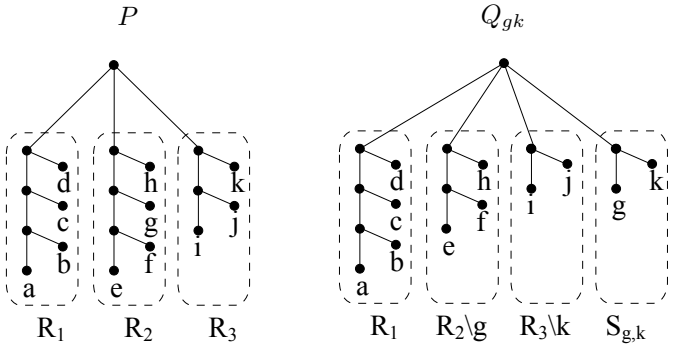


FIG. 4.8 – Etant donnée une instance de RPIS avec $k = 3$, $V_1 = \{a, b, c, d\}, V_2 = \{e, f, g, h\}, V_3 = \{i, j, k\}$ et contenant une arête gk, les arbres P et Q_{gk} construits par la réduction.

4.2 Problème Mct

On considère dans cette section le problème MCT. Le plan suivi est similaire à la section 4.1. On présente en Section 4.2.1 contient des définitions relatives au problème MCT, ainsi qu'un historique des résultats obtenus. On décrit en Section 4.2.2 un algorithme de 3-approximation en temps linéaire pour le problème complémentaire. On présente en Section 4.2.3 des algorithmes exacts (polynomiaux ou FPT) pour MCT. Enfin, on présente en Section 4.2.4 des résultats d'inapproximabilité et de complexité paramétrique pour le problème.

4.2.1 Préliminaires

Pré-sous-arbres et préplongements

On donne une définition équivalente en termes de *préplongement*. Un *préplongement* de S dans T est une application $\phi : N(S) \to N(T)$ telle que :

1. ϕ préserve feuilles et noeuds internes : (i) si l est une feuille de S d'étiquette x, alors $\phi(l)$ est une feuille de T d'étiquette x, (ii) si u est un noeud interne de S, alors $\phi(u)$ est un noeud interne de T ;
2. (i) si u, v sont deux noeuds de S tels que $u \leq_S v$, alors $\phi(u) \leq_T \phi(v)$; (ii) si u, v sont deux noeuds incomparables de S, alors $\phi(\mathsf{lca}_S(u, v)) = \mathsf{lca}_T(\phi(u), \phi(v))$.

On a alors : $S \trianglelefteq T$ si et seulement si il existe un préplongement de S dans T. Cette définition est illustrée ci-dessous.

Observons que la définition de préplongement diffère de la définition de plongement donnée en Section 4.1.1 par le point 2 (i) : dans le cas d'un préplongement, deux noeuds u, v tels que u soit descendant de v peuvent avoir la même image, ce qui n'était pas possible dans le cas d'un plongement. De ce fait, un préplongement n'est pas une injection en général.

On a les propriétés suivantes :

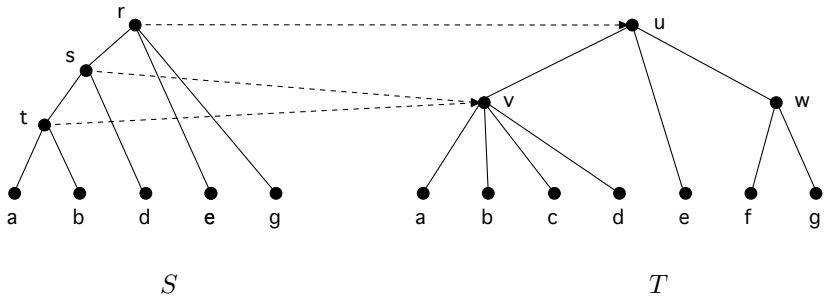


FIG. 4.9 – Deux arbres S, T tels que $S \trianglelefteq T$. Le préplongement ϕ de S dans T est tel que $\phi(r) = u, \phi(s) = \phi(t) = v$, comme indiqué par les flèches en pointillé.

Observation 4.38. *1. L'identité est un préplongement de T dans T ;*

2. Si ϕ est un préplongement de S dans T, et ϕ' est un plongement de T dans U, alors $\phi' \circ \phi$ est un préplongement de S dans U ;

3. Si ϕ est un préplongement de S dans T, alors $\phi|N(S|L')$ est un préplongement de $S|L'$ dans $T|L'$.

La relation $\trianglelefteq$ a donc les propriétés suivantes :

Lemme 4.39. *(i) $\trianglelefteq$ est une relation d'ordre ; (ii) si $S \trianglelefteq T$, alors $S|L' \trianglelefteq T|L'$.*

Arbres compatibles, problème Mct

Soit $\mathcal{T} = \{T_1, ..., T_k\}$ une collection d'arbres sur un même ensemble d'étiquettes L. Soit S un arbre sur $L' \subseteq L$. On dit que S est un *arbre compatible* de $\mathcal{T}$ si et seulement si $S \trianglelefteq T_i$ pour tout i. On dit que $\mathcal{T}$ est *compatible* si et seulement si elle admet un arbre compatible S tel que $L(S) = L$. Si $\mathcal{T}$ est compatible, un tel arbre S est appelé *raffinement commun* de $\mathcal{T}$, et on dit que S est un *raffinement commun minimal* de $\mathcal{T}$ si pour tout S' raffinement commun de $\mathcal{T}$, S' raffine S.

Ces notions ont les propriétés suivantes :

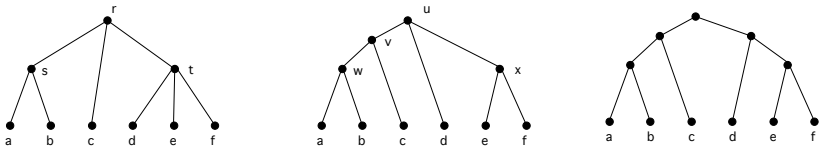
Lemme 4.40. *(i) Si T est un arbre compatible de $\mathcal{T}$ et si $S \trianglelefteq T$, alors S est un arbre compatible de $\mathcal{T}$;*

(ii) Si T est un arbre compatible de $\mathcal{T}$, alors $T|L'$ est un arbre compatible de $\mathcal{T}|L'$;

(iii) Si $\mathcal{T}$ est compatible, alors $\mathcal{T}|L'$ est compatible.

La notion d'arbre compatible est illustrée ci-dessous :

Le problème MCT consiste à chercher un arbre compatible de taille maximum d'une collection donnée en entrée. Comme dans le cas de MAST, on définit formellement le problème en terme d'ensemble d'étiquettes : étant donnée une collection d'arbres $\mathcal{T}$ sur L, trouver $L' \subseteq L$ de cardinal maximum tel que $\mathcal{T}|L'$ soit compatible. On définit également le problème complémentaire CMCT : étant donnée une collection d'arbres $\mathcal{T}$ sur L, trouver $L' \subseteq L$ de cardinal minimum tel



T_1 T_2 S

FIG. 4.10 – Une collection de deux arbres $\mathcal{T} = \{T_1, T_2\}$ sur l'ensemble d'étiquettes $L = \{a, b, c, d, e, f\}$, et un arbre compatible maximum S.

que $\mathcal{T} \backslash L'$ soit compatible. On note $MCT(\mathcal{T})$, resp. $CMCT(\mathcal{T})$, la taille d'une solution optimale pour MCT, resp. CMCT.

Les paramètres considérés pour l'étude du problème MCT sont les mêmes que dans le cas de MAST : le nombre d'étiquettes n, le nombre d'arbres k, le degré maximum d d'un arbre de $\mathcal{T}$. Etant donné $k \geq 2$, on note $k-$MCT la restriction de MCT aux collections de k arbres. On considère également des versions paramétrées de MCT dans lesquelles on fixe des bornes sur la taille d'une solution cherchée : (i) étant donnée une collection $\mathcal{T}$ et un entier q, décider si $MCT(\mathcal{T}) \geq q$, (ii) étant donnée une collection $\mathcal{T}$ et un entier p, décider si $CMCT(\mathcal{T}) \leq p$.

De manière analogue à la caractérisation des collections non isomorphes fournies par le Lemme 4.4, on a une caractérisation simple des collections non compatibles en termes de conflits. En reprenant la notion de h-conflit introduite en Section 4.1.1, on a :

Lemme 4.41 ([9, 22]). *(i) Deux arbres T_1, T_2 sont compatibles si et seulement si il n'existe pas de h-conflit entre T_1, T_2 ; (ii) Une collection $\mathcal{T}$ est compatible si et seulement si il n'existe pas de h-conflit entre $\mathcal{T}$.*

Historique et nouveaux résultats

Le problème MCT a été introduit dans [26]. [26] ont montré la NP-difficulté de $6-$MCT. [29] ont montré que $2-$MCT était NP-dur si l'un des arbres était de degré non borné, et était polynomial si les deux arbres étaient de degré borné.

Dans [7], nous avons étudié l'approximabilité du problème MCT. Nous avons notamment obtenu des résultats d'inapproximabilité semblables à ceux connus pour MAST, à savoir : (i) MCT est aussi difficile à approximer que MAXIMUM INDEPENDENT SET ; (ii) $2-$MCT est APX-dur, et n'est pas approximable à un facteur constant si P $\neq$ NP. L'approximabilité du problème complémentaire CMCT a été étudiée par [22, 7, 24]. En utilisant la caractérisation du Lemme 4.41, [22] présentent un algorithme de 3-approximation en temps $O(k^2n^2)$, et [6] présentent un algorithme de 3-approximation en temps $O(n^2 + kn)$. Dans [24], nous obtenons un algorithme de 3-approximation en temps linéaire $O(kn)$. Par ailleurs, nous montrons dans [7] que $2-$CMCT est APX-dur.

Des résultats ont été également obtenus sur la complexité paramétrique de MCT. En utilisant le Lemme 4.41, [8] obtiennent des algorithmes FPT par rap-

port à p, de complexité temporelle $O(3^p kn)$ et $O(n^3 + 2.27^p)$. [23] décrit un algorithme résolvant MCT en temps $O(2^{2kd}n^k)$. Dans [25], nous obtenons de nouveaux résultats sur la complexité paramétrique de MCT. D'une part, nous montrons que le problème est polynomial dans le cas d'arbres de degré borné, en présentant un algorithme de complexité temporelle $O(n^{2^{d+1}} + kn^3)$. Nous montrons également que le problème est W[1]-dur par rapport à la paire de paramètres (d, q) ; la réduction utilisée pour établir ce résultat implique en fait que MCT ne peut pas être résolu en temps $\Phi(d)N^{o(2^{d/2})}$ sous l'hypothèse ETH (où N est la taille de l'instance). Enfin, nous présentons des algorithmes FPT pour les paires de paramètres (k, d) et (k, q), de complexités respectives $O(2^{2kd}n^3)$ et $O(2^{O(kq)}n^3)$.

4.2.2 Algorithme d'approximation

On présente dans cette section un algorithme de 3-approximation en temps linéaire $O(kn)$ pour le problème CMCT.

Le principe de l'algorithme est analogue à l'algorithme décrit pour CMAST en Section 4.1.2. Introduisons les définitions suivantes.

Définition 4.42. *Soit T_1, T_2 deux arbres sur le même ensemble d'étiquettes L. Un* h-conciliateur *de T_1, T_2 est un ensemble $\mathcal{C}$ de h-conflits disjoints entre T_1, T_2 tels que $T_1 \backslash L(\mathcal{C}), T_2 \backslash L(\mathcal{C})$ aient un raffinement commun.*

Définition 4.43. *Soit $\mathcal{T}$ une collection sur un ensemble d'étiquettes L. Un* h-conciliateur *de $\mathcal{T}$ est un ensemble $\mathcal{C}$ de h-conflits disjoints entre $\mathcal{T}$ tels que $\mathcal{T} \backslash L(\mathcal{C})$ soit compatible.*

Le lemme 4.41 implique qu'un h-conciliateur fournit une 3-approximation du problème CMCT :

Lemme 4.44. *Si $\mathcal{C}$ est un h-conciliateur de $\mathcal{T}$, alors $L(\mathcal{C})$ est une 3-approximation de* CMCT *pour $\mathcal{T}$.*

De manière analogue à l'algorithme TROUVERWHSCONCILIATEUR décrit en Section 4.1.2, on va décrire un algorithme TROUVERHCONCILIATEUR qui, étant donnés deux arbres T_1, T_2 sur un même ensemble de n étiquettes, trouve un h-conciliateur de T_1, T_2 en temps $O(n + c)$ où c est le nombre de conflits trouvés (Proposition 4.46). Cet algorithme fournit un algorithme de 3-approximation en temps linéaire pour CMCT :

Proposition 4.45. CMCT *est 3-approximable en temps $O(kn)$.*

Démonstration. Etant donnée une collection $\mathcal{T} = \{T_1, ..., T_k\}$, on détermine un h-conciliateur de $\mathcal{T}$ de la manière suivante. On maintient un arbre S raffinement commun minimal de $T_1, ..., T_i$, initialisé à T_1. Pour chaque i de 2 à k, on identifie un h-conciliateur $\mathcal{C}_i$ de S, T_i, on supprime ses étiquettes de la collection, et on remplace S par le raffinement commun minimal de $S \backslash L(\mathcal{C}_i), T_i \backslash L(\mathcal{C}_i)$. Identifier un h-conciliateur de S, T_i se fait par un appel à TROUVERHCONCILIATEUR(S, T_i).

Cet algorithme construit bien un h-conciliateur de $\mathcal{T}$: en effet, notons $\mathcal{C}'_i$ l'ensemble des conflits identifiés à la fin de l'étape i, on montre par récurrence que $\mathcal{C}'_i$ est un h-conciliateur de $\{T_1, ..., T_i\}$. Ceci résulte du Lemme 5.24 et du fait que $\mathcal{C}'_i = \mathcal{C}'_{i-1} \cup \mathcal{C}_i$, où $\mathcal{C}'_{i-1}$ est un h-conciliateur de $\{T_1, ..., T_{i-1}\}$, et $\mathcal{C}_i$ est un h-conciliateur de S, T_i, avec S raffinement commun minimal de $T_1, ..., T_{i-1}$.

Comme l'algorithme fait $(k-1)$ appels à TrouverHConciliateur et que chaque appel prend un temps $O(n)$, le temps d'exécution est clairement $O(kn)$. □

On montre maintenant :

Proposition 4.46. *Il existe un algorithme* TrouverHConciliateur *qui détermine un h-conciliateur de T_1, T_2 en temps $O(n + c)$.*

L'algorithme TrouverHConciliateur est une adaptation de l'algorithme TrouverWhsConciliateur. Il maintient une liste $\mathcal{C}$ de h-conflits disjoints entre T_1, T_2, initialisée à $\emptyset$. Il effectue un parcours ascendant de T_1, en cherchant à apparier chaque noeud u de T_1 avec un noeud v de T_2, tel que $T_1(u)\backslash L(\mathcal{C})$ et $T_2(v)\backslash L(\mathcal{C})$ aient un raffinement commun.

Le pseudocode de la procédure TrouverHConciliateur est semblable à celui de la procédure TrouverWhsConciliateur, et est donné ci-dessous :

TrouveHConciliateur(T_1, T_2)
  {initialisation}
  InitialiserStructures()
  pour chaque feuille u de T_1 **faire**
    soit v la feuille de T_2 de même étiquette
    $match(u) \leftarrow v$
  fin pour
  pour chaque noeud interne u de T_1 examiné en ordre postfixe **faire**
    $v \leftarrow$ ApparierNoeud(u)
    si $v \neq \perp$ alors $match(u) \leftarrow v$
  fin pour

Décrivons maintenant le processus d'appariement. Lors d'un appel à la procédure ApparierNoeud(u), les fils $u_1, ..., u_d$ de u ont été appariés à des feuilles $v_1, ..., v_d$ de T_2, et on cherche à apparier u avec un noeud v de T_2. Dans T_2, colorons en noir les feuilles v_i, et en blanc les autres feuilles. Colorons chaque noeud v de T_2 : (i) en noir si $T_2(u)$ ne contient que des feuilles noires, (ii) en blanc si $T_2(u)$ ne contient que des feuilles blanches, (iii) en gris sinon. L'appariement sera possible si et seulement si :

Condition 4.47. *Soit $v = \mathsf{lca}_{T_2}(v_1, ..., v_d)$. Alors v n'a pas de fils gris.*

Autrement dit, chaque fils de v est noir ou blanc. Tant que cette condition n'est pas vérifiée, on a la propriété que le sous-arbre $T_2(v)$ contient deux feuilles noires x, y et une feuille blanche z telles que $x|yz \in rt(T_2)$: en effet, soit w un

fils gris de v, alors on peut choisir $y, z \in T_2(w)$ et x à l'extérieur de $T_2(w)$. Une procédure TROUVECONFLIT va alors trouver trois feuilles x, y, z de la sorte. En choisissant $l \in I(x), l' \in I(y), l'' \in I(z)$, on obtient un h-conflit $\{l, l', l''\}$ entre T_1, T_2. Comme en Section 4.1.2, on supprime alors les étiquettes l, l', l'' par trois appels à une procédure auxiliaire *supprimerEtiquette*.

Lorsque la condition est vérifiée, la procédure APPARIERNOEUD(u) renvoie un noeud de T_2 auquel u va alors être apparié dans la procédure appelante TROUVEHCONCILIATEUR. Si v n'a pas de fils blanc, alors u va être apparié avec v, qui est donc retourné par APPARIERNOEUD(u). Si v a au moins un fils blanc et a pour fils noirs $v'_1, ..., v'_q$, alors dans T_2 les noeuds v'_i sont supprimés et remplacés par un unique noeud v' tel que $I(v') = I(v'_1) \cup ... \cup I(v'_q)$; alors u va être apparié avec v', qui est donc retourné par APPARIERNOEUD(u).

Justifions maintenant que l'identification d'un conflit par la procédure TROUVECONFLIT peut se faire en temps amorti $O(1)$.

On suppose que chaque noeud v de T_2 comporte un champ $filsNoirs(v)$. L'algorithme va maintenir l'invariant suivant :

Invariant 4.48. *Au début de chaque appel à* TROUVECONFLIT*, (i) la liste $FeuillesNoires$ contient les feuilles noires de T_2, (ii) pour chaque noeud v de T_2, $filsNoirs(v)$ est la liste des fils noirs de v.*

Pour maintenir cet invariant, on utilise deux procédures : (i) la procédure $devientFeuilleNoire(v)$ est appelée lorsque v devient une feuille noire, elle ajoute v à la liste $FeuillesNoires$ et à la liste des fils noirs du noeud père ; (ii) la procédure $supprimerFeuille(v)$ qui supprime la feuille v de T_2, et si v était noire supprime v de la liste $FeuillesNoires$ et de la liste des fils noirs du noeud père. On suppose que l'ajout et la suppression d'un noeud v dans la liste $filsNoirs(w)$ est effectué par des procédures auxiliaires $ajouterFilsNoir(v, u)$, $supprimerFilsNoir(v, u)$.

Le pseudocode des procédures $devientFeuilleNoire$ et $supprimerFeuille$ est donné ci-dessous :

$devientFeuilleNoire(v)$
  $couleur(v) \leftarrow noir$
  $FeuillesNoires \leftarrow FeuillesNoires \cup \{v\}$
  $p \leftarrow pere(v)$
  si $p \neq \perp$ alors $ajouterFilsNoir(v, p)$

Les procédures $devientFeuilleNoire$ et $supprimerFeuille$ peuvent s'implémenter de façon à avoir un temps d'exécution constant. Pour ce faire, on représente la liste $FeuillesNoires$ et les listes $filsNoirs(v)$ par des listes doublement chaînées. En outre, chaque feuille noire v comporte deux champs auxiliaires : le premier maintient un pointeur vers la cellule correspondante de la liste $FeuillesNoires$, le deuxième maintient un pointeur vers la cellule correspondante de la liste $filsNoirs(p)$.

---

supprimerFeuille(v)
  {mettre à jour T_2}
  ...
  si *couleur*$(v) =$ *noir* **alors**
    $FeuillesNoires \leftarrow FeuillesNoires - \{v\}$
    $p \leftarrow pere(v)$
    si $p \neq \perp$ alors $supprimerFilsNoir(v, p)$
  fin si

On décrit maintenant un second invariant. Disons qu'un noeud interne de T_2 est *lourd* s'il a au moins deux fils qui sont des feuilles noires. On va maintenir l'invariant suivant :

Invariant 4.49. *Au début de chaque appel à* TrouveConflit*, T_2 ne comporte aucun noeud lourd.*

Pour maintenir l'invariant 4.49, on fait en sorte que : dès qu'un noeud interne de T_2 devient lourd, ses fils noirs sont remplacés par une unique feuille noire. Ceci est détecté par la procédure $ajouterFilsNoir$, qui appelle alors la procédure *devientLourd*. Le maintien de l'invariant est assuré par la procédure *devientLourd*(v) (appelée si v noeud de T_2 devient lourd) et par la procédure *commenceAppariement*(u) (appelée au début de l'appariement de u noeud de T_1).

Le pseudocode de ces deux procédures est donné ci-dessous :

---

devientLourd(v)
  soient $v_1, ..., v_d$ les fils noirs de v (éléments de la liste $filsNoirs(v)$)
  si tous les fils de v sont noirs **alors**
    $w \leftarrow v$
  sinon
    ajouter à v un nouveau fils w
  fin si
  $I(w) \leftarrow I(v_1) \cup ... \cup I(v_d)$
  supprimer $v_2, ..., v_d$ de $\mathcal{I}$, et remplacer v_1 par w
  pour chaque $i \in [d]$, $supprimerFeuille(v_i)$
  $devientFeuilleNoire(w)$

---

commenceAppariement(u)
  soient $u_1, ..., u_d$ les fils de u
  soit $v_1 = match(u_1), ..., v_d = match(u_d)$ les noeuds de T_2 associés
  pour chaque $i \in [d]$, $devientFeuilleNoire(v_i)$

Une analyse immédiate, analogue à celle menée en Section 4.1.2, permet de voir que le temps d'exécution propre d'un appel à *devientLourd*(v) est en

$O(n(v))$ où $n(v)$ est le nombre de fils noirs de v, et que le temps d'exécution propre d'un appel à la procédure *commenceAppariement*(u) est en $O(d(u))$, où $d(u)$ est le degré de u dans T_1.

Sans faire d'hypothèse sur la méthode de détection des conflits, on représente ci-dessous le processus d'appariement d'un noeud u :

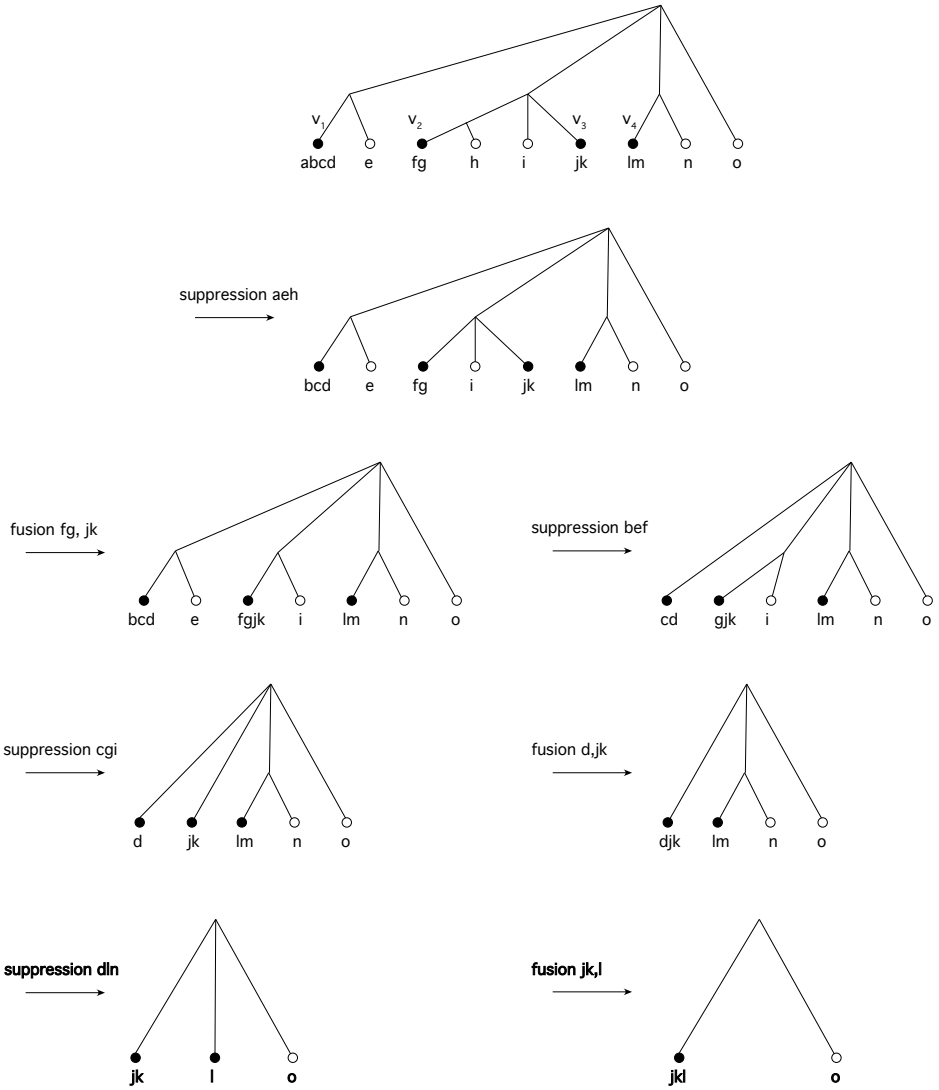


FIG. 4.11 – Le processus d'appariement d'un noeud u ayant quatre fils u_1, u_2, u_3, u_4 avec $I(u_1) = \{a, b, c, d\}, I(u_2) = \{f, g\}, I(u_3) = \{j, k\}, I(u_4) = \{l, m\}$.

On va maintenant adapter à MCT la stratégie d'identification des conflits

utilisant la notion de *feuilles bornes*, présentée en Section 4.1.2. Comme en Section 4.1.2, on maintient la liste des feuilles bornes :

Invariant 4.50. *Au début de chaque appel à* TrouveConflit*, la liste* $FeuillesBornes$ *contient les feuilles bornes de* T_2*.*

Le maintien de cet invariant s'effectue comme indiqué en Section 4.1.2. Les lemmes suivants montrent que les feuilles bornes permettent d'identifier un conflit tant que $|FeuillesNoires| \geq 2$.

Comme en Section 4.1.2, appelons *intervalle noir* de $\mathcal{I}$ un intervalle maximal de $\mathcal{I}$ constitué uniquement de feuilles noires.

Soit $\sigma = v_1...v_m$ une séquence de feuilles de T_2, et pour tout i soit $u_i = parent_{T_2}(v_i)$. On dit que σ est un *peigne ascendant* si et seulement si pour tout i, $u_{i+1} >_{T_2} u_i$. On dit que σ est un *peigne descendant* si et seulement si pour tout i, $u_{i+1} <_{T_2} u_i$.

Le lemme suivant caractérise les intervalles noirs :

Lemme 4.51. *Soit* $I = a_1...a_m$ *un intervalle noir de* $\mathcal{I}$*, alors :*

1. *l'une des extrémités de* I *est une borne ;*
2. *il existe* $i \leq m$ *tel que* $a_1, ..., a_i$ *est un peigne ascendant et* $a_{i+1}, ..., a_m$ *est un peigne descendant.*

Démonstration. Observons que l'invariant 4.49 implique que T_2 ne comporte aucun noeud plein, et donc le lemme 4.18 de la Section 4.1.2 s'applique.

Le point 1 résulte alors du point 1 du lemme 4.18. Le point 2 résulte du point 2 du lemme 4.18, et de l'invariant 4.49. □

Le lemme suivant montre que, tant que u est non apparié, on peut trouver soit deux feuilles bornes, soit une feuille borne vérifiant une certaine condition :

Lemme 4.52. *Au début de l'appel à* TrouveConflit*, on a : si* $|FeuillesNoires| \geq 2$*, alors*

- *soit* $|FeuillesBornes| \geq 2$*,*
- *soit* $FeuillesBornes = \{a\}$*, et pour tout* $b \in FeuillesNoires$ *distinct de* a*,* $parent_{T_2}(a) \not> _{T_2} parent_{T_2}(b)$*.*

Démonstration. Soit $I_1, ..., I_p$ une énumération des intervalles noirs de $\mathcal{I}$. Si $p \geq 2$, alors par le point 1 du lemme 4.51 on peut trouver des feuilles bornes $a_1 \in I_1, a_2 \in I_2$, et donc $|FeuillesBornes| \geq 2$.

Supposons maintenant que $p = 1$. Soit $I = I_1$ l'unique intervalle noir. On a alors $|I| \geq 2$ puisque $|FeuillesNoires| \geq 2$. Alors $I = a_1...a_m$ avec $m \geq 2$. Par le point 2 du lemme 4.51, il existe $i \leq m$ tel que $a_1, ..., a_i$ est un peigne ascendant et $a_{i+1}, ..., a_m$ est un peigne descendant. Définissons $u_1, ..., u_m$ par $u_i = parent_{T_2}(a_i)$, on a donc $u_1 <_{T_2} u_2 <_{T_2} ... <_{T_2} u_i$ et $u_{i+1} >_{T_2} ... >_{T_2} u_m$. Alors $FeuillesBornes = \{a\}$ avec :

- soit $a = a_1$ et pour tout $j > 1$, $parent_{T_2}(a_1) \not> _{T_2} parent_{T_2}(a_j)$: en effet, si $j \leq i$ alors $u_1 <_{T_2} u_j$, et si $j > i$ alors u_1, u_j sont incomparables par $<_{T_2}$.

– soit $a = a_m$ et pour tout $i < m$, $parent_{T_2}(a_m) \not> _{T_2} parent_{T_2}(a_i)$: raisonnement similaire au cas précédent.

On conclut en observant que $FeuillesNoires = I$. □

Le lemme suivant montre qu'on peut identifier un h-conflit tant que u est non apparié :

Lemme 4.53. *Soit a une feuille borne, et soit b une feuille noire distincte de a et telle que $parent_{T_2}(a) \not> _{T_2} parent_{T_2}(b)$. Soit $p = pred_{\mathcal{I}}(a)$ et $s = succ_{\mathcal{I}}(a)$. Alors :*

– *si a est une borne gauche : alors $\{a, b, p\}$ est un h-conflit entre T_1, T_2 ;*
– *si a est une borne droite : alors $\{a, b, s\}$ est un h-conflit entre T_1, T_2.*

Démonstration. Par symétrie, on peut supposer que a est une borne gauche. Alors p est blanche, et on a (i) soit $s = \perp$, (ii) soit $s \neq \perp$ et $sa|p \notin rt(T_2)$. Soit $u = parent_{T_2}(a)$ et $u' = parent_{T_2}(b)$.

Soit $v = \mathsf{lca}_{T_2}(a, b)$, alors $u <_{T_2} v$: en effet, on a $u \leq_{T_2} v$ et $u' \leq_{T_2} v$, et $u = v$ est impossible car cela impliquerait que $u' \leq_{T_2} u$ ($u = u'$ contredirait l'invariant 4.49, et $u' <_{T_2} u$ contredirait l'hypothèse du lemme). D'autre part, $u = \mathsf{lca}_{T_2}(a, p)$: c'est clair si $n = \perp$, et si $n \neq \perp$ cela résulte du fait que $sa|p \notin rt(T_2)$.

On a alors $u = \mathsf{lca}_{T_2}(a, p) <_{T_2} \mathsf{lca}_{T_2}(a, b) = v$, ce qui implique que $T_2|\{a, b, p\}$ est l'arbre $ap|b$. On conclut que $\{a, b, p\}$ est un h-conflit entre T_1, T_2. □

En utilisant la méthode d'identification des conflits du Lemme 4.53, on obtient l'algorithme d'appariement suivant :

TrouveConflit()
  si $|FeuillesBornes| \geq 2$ **alors**
    choisir $a, b \in FeuillesBornes$
    si $parent_{T_2}(a) >_{T_2} parent_{T_2}(b)$ **alors**
      $a \leftrightarrow b$
    fin si
  sinon
    soit a l'unique élément de $FeuillesBornes$
    soit $b \in FeuillesNoires \backslash \{a\}$
  fin si
  $p \leftarrow pred_{\mathcal{I}}(a), s \leftarrow succ_{\mathcal{I}}(a)$
  si a est une borne gauche **alors**
    renvoyer $\{a, b, p\}$
  sinon
    renvoyer $\{a, b, s\}$
  fin si

Observons qu'à la fin de la procédure ApparierNoeud, les listes $FeuillesNoires$, $FeuillesBornes$, et les listes $filsNoir(u)$ pour u noeud de T_2, sont réinitialisées à la liste vide. Par une analyse similaire à celle menée en Section 4.1.2, on peut

ApparierNoeud(u)
  commenceAppariement(u)
  tant que $|FeuillesNoires| \geq 2$ **faire**
    $\{x, y, z\} \leftarrow$ TrouveConflit()
    choisir $l \in I(x), l' \in I(y), l'' \in I(z)$
    supprimer les étiquettes l, l', l''
  fin tant que
  si $|FeuillesNoires| = 0$ **alors**
    {l'appariement a échoué}
    renvoyer $\perp$
  fin si
  si $|FeuillesNoires| = 1$ **alors**
    {l'appariement a réussi}
    soit v l'unique élément de $FeuillesNoires$
    $FeuillesNoires \leftarrow \emptyset$
    $FeuillesBornes \leftarrow \emptyset$
    $p \leftarrow pere(v)$, $filsNoirs(p) \leftarrow \emptyset$
    renvoyer v
  fin si

montrer que la procédure TrouveHConciliateur(T_1, T_2) s'exécute en temps $O(n + c)$.

4.2.3 Algorithme polynomial et algorithmes FPT

Algorithme polynomial

On présente un algorithme polynomial pour Mct sur des arbres de degré borné. On montre :

Proposition 4.54. Mct *est soluble en temps* $O(n^{2^{d+1}} + kn^3)$.

On suppose dans la suite que $\mathcal{T} = \{T_1, ..., T_k\}$ est une collection sur L d'arbres de degré $\leq d$. Notons $BCT(\mathcal{T})$ l'ensemble des arbres compatibles binaires S pour $\mathcal{T}$, alors $MCT(\mathcal{T})$ est la taille d'un plus grand arbre de $BCT(\mathcal{T})$. On va décrire des relations de récurrence permettant de calculer $MCT(\mathcal{T})$ par programmation dynamique en temps $O(n^{2^{d+1}} + kn^3)$.

On procède en trois étapes. Dans un premier temps, on définit les sous-problèmes du programme dynamique : étant donné un arbre U de hauteur $\leq p$, on définira un ensemble d'arbres $BCT_p(U)$, qui intuitivement est l'ensemble des arbres compatibles obtenus à partir de U en substituant chaque noeud de U de profondeur p par un sous-arbre. Les sous-problèmes du programme dynamique consisteront à calculer pour chaque arbre U, la taillle $MCT_p(U)$ d'un plus grand arbre de $BCT_p(U)$.

Dans un second temps, on décrit une caractérisation récursive des ensembles $BCT(\mathcal{T})$ faisant intervenir des relations binaires $\prec_{\mathcal{T}}$ et $\perp_{\mathcal{T}}$ sur $BCT(\mathcal{T})$. On

montre dans un troisième temps que cette caractérisation récursive implique une description récursive des ensembles $BCT_d(U)$, dans laquelle un ensemble $BCT_d(U)$ est mis en relation avec des ensembles $BCT_d(U')$ pour $U' \prec_{\mathcal{T}} U$. Au moment de prouver la Proposition 4.54, on verra que cette description récursive fournit une relation de récurrence pour calculer les valeurs $MCT_d(U)$, et ainsi obtenir $MCT(\mathcal{T})$.

1er point. On introduit la notion de *tuteur*, ainsi que les ensembles BCT_p et $BCT_p(U)$.

Soit un arbre S. La *profondeur* d'un noeud u de S est la longueur du chemin joignant u à la racine ; par convention, la profondeur de la racine est 0. La *hauteur* de S est la profondeur du noeud le plus profond.

Soit un entier p supérieur ou égal à la hauteur de S, et soit un arbre T. On dit que S est un *p-tuteur* de T si et seulement si il existe une famille d'arbres $\{T_x : x \in L(S)\}$ telle que : (i) T est obtenu à partir de S en substituant chaque feuille $x \in L(S)$ par T_x, (ii) si x est à profondeur $< p$ alors T_x est une feuille de même étiquette que x ; si x est à profondeur p alors T_x contient l'étiquette de x.

Cette définition est illustrée ci-dessous :

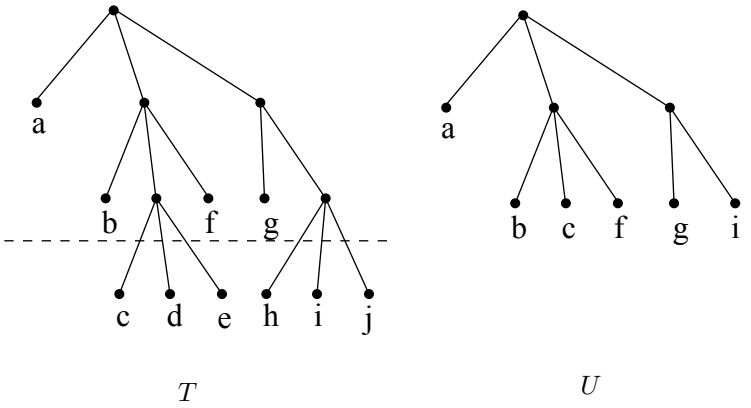


FIG. 4.12 – Un arbre T, et un arbre U étant un 2-tuteur de T.

Les deux lemmes suivants, donnés sans démonstration, décrivent des propriétés simples des tuteurs :

Lemme 4.55. *Soit un entier $p \geq 1$. Soit $U = (U_1, U_2)$, $S = (S_1, S_2)$. Les points suivants sont équivalents :*

- *U est un p-tuteur de S ;*
- *pour $i \in \{1, 2\}$, U_i est un $(p-1)$-tuteur de S_i.*

Lemme 4.56. *Soit des entiers $p, q \geq 1$ tels que $q \leq p$.*

1. *Si V est un p-tuteur de T, et si U est un q-tuteur de V, alors U est un q-tuteur de T ;*

2. *Si U est un q-tuteur de T, il existe V p-tuteur de T tel que V soit également un p-tuteur de U.*

Soit p un entier. On note BCT_p l'ensemble des arbres de $BCT(\mathcal{T})$ de hauteur $\leq p$. Si $U \in BCT_p$, on note $BCT_p(U)$ l'ensemble des arbres $S \in BCT(\mathcal{T})$ dont U est un p-tuteur. Observons que :

Lemme 4.57. $|BCT_p| = O(n^{2^p})$.

Démonstration. En effet, soit N_p le nombre d'arbres binaires S de hauteur $\leq p$ tels que $L(S) \subseteq L$. Alors $|BCT_p| \leq N_p$. Pour tout $i \leq 2^p$, notons $n_{i,p}$ le nombre d'arbres binaires non étiquetés de hauteur $\leq p$ et comportant i feuilles. Alors :

$$N_p = \sum_{i=1}^{2^p} n_{i,p} \ \frac{n!}{(n-i)!}$$

Comme $\frac{n!}{(n-i)!} = O(n^i)$, on conclut que $N_p = O(n^{2^p})$. □

Lemme 4.58. $BCT(\mathcal{T}) = \cup_{U \in BCT_p} BCT_p(U)$.

Démonstration. ($\supseteq$) est clair. ($\subseteq$) résulte du fait que si $S \in BCT(\mathcal{T})$, alors en considérant U un p-tuteur arbitraire de S, on obtient que $U \in BCT_p$ et $S \in BCT_p(U)$. □

2e point. Le lemme suivant fournit une caractérisation récursive des arbres compatibles. Etant donné $L' \subseteq L$, un arbre T sur L et un noeud u de T, on définit $\mathsf{span}_T(u, L')$ comme l'ensemble des fils v de u tels que $L(v) \cap L' \neq \emptyset$. Etant donné un $\mathcal{T}$-tuple π, on définit $\mathsf{span}^{\mathcal{T}}(\pi, L') = (\mathsf{span}_{T_1}(\pi[1], L'), ..., \mathsf{span}_{T_k}(\pi[k], L'))$.

Lemme 4.59. *Soit S un arbre binaire sur $L' \subseteq L$ tel que $|S| \geq 2$. Les points suivants sont équivalents :*

1. *$S \in BCT(\mathcal{T})$;*
2. *$S = (S_1, S_2)$ avec $S_1 \in BCT(\mathcal{T})$, $S_2 \in BCT(\mathcal{T})$, et si l'on pose $\pi = \mathsf{lca}^{\mathcal{T}}(S)$, alors $(\mathsf{span}^{\mathcal{T}}(\pi, S_1), \mathsf{span}^{\mathcal{T}}(\pi, S_2))$ est une partition de $\mathsf{span}^{\mathcal{T}}(\pi, S)$.*

Pour $i \in [k]$ on définit les relations binaires $\prec_{T_i}$, $\equiv_{T_i}$ et $\perp_{T_i}$ sur $BCT(\mathcal{T})$ par :
- $S \prec_{T_i} S'$ si et seulement si (i) $\mathsf{lca}_{T_i}(S) <_{T_i} \mathsf{lca}_{T_i}(S')$ ou (ii) $\mathsf{lca}_{T_i}(S) = \mathsf{lca}_{T_i}(S')$ (égal à un noeud u) et $\mathsf{span}_{T_i}(u, S) \subset \mathsf{span}_{T_i}(u, S')$.
- $S \equiv_{T_i} S'$ si et seulement si (i) $\mathsf{lca}_{T_i}(S) = \mathsf{lca}_{T_i}(S')$ (égal à un noeud u) et (ii) $\mathsf{span}_{T_i}(u, S) = \mathsf{span}_{T_i}(u, S')$;
- $S \perp_{T_i} S'$ si et seulement si en posant $u = \mathsf{lca}_{T_i}(L(S) \cup L(S'))$, on a : $\mathsf{span}_{T_i}(u, S), \mathsf{span}_{T_i}(u, S')$ sont disjoints.

On définit les relations binaires $\prec_{\mathcal{T}}$, $\equiv_{\mathcal{T}}$ et $\perp_{\mathcal{T}}$ sur $BCT(\mathcal{T})$ par :
- $S \prec_{\mathcal{T}} S'$ si et seulement si $S \prec_{T_i} S'$ pour tout $i \in [k]$;
- $S \equiv_{\mathcal{T}} S'$ si et seulement si $S \equiv_{T_i} S'$ pour tout $i \in [k]$;
- $S \perp_{\mathcal{T}} S'$ si et seulement si $S \perp_{T_i} S'$ pour tout $i \in [k]$.

La relation $\prec_{\mathcal{T}}$ est un ordre strict qui guidera l'étape de programmation dynamique utilisée dans l'algorithme. D'autre part, la relation $\perp_{\mathcal{T}}$ permet de réénoncer le lemme 4.59 de la façon suivante, en reformulant la condition sur les span à l'aide de cette relation :

Lemme 4.60. *Soit S un arbre binaire sur $L' \subseteq L$ tel que $|S| \geq 2$. Les points suivants sont équivalents :*

1. *$S \in BCT(\mathcal{T})$;*
2. *$S = (S_1, S_2)$ avec $S_1, S_2 \in BCT(\mathcal{T})$ et $S_1 \perp_{\mathcal{T}} S_2$.*

Démonstration. Il suffit d'observer que : si l'on pose $\pi = \mathsf{lca}^{\mathcal{T}}(S)$, alors il y a équivalence entre : (i) $S_1 \perp_{\mathcal{T}} S_2$, (ii) $(\mathsf{span}^{\mathcal{T}}(\pi, S_1), \mathsf{span}^{\mathcal{T}}(\pi, S_2))$ est une partition de $\mathsf{span}^{\mathcal{T}}(\pi, S)$. □

De plus, notons que :

Lemme 4.61. *Si $S = (S_1, S_2)$ avec $S_1 \perp_{\mathcal{T}} S_2$, alors $S_1 \prec_{\mathcal{T}} S$ et $S_2 \prec_{\mathcal{T}} S$.*

Démonstration. Montrons que $S_1 \prec_{\mathcal{T}} S$, l'autre cas étant symétrique. On montre en fait que $S_1 \prec_{T_i} S$ pour tout $i \in [k]$. Fixons $i \in [k]$, et soit $u = \mathsf{lca}_{T_i}(L(S_1))$ et $v = \mathsf{lca}_{T_i}(L(S))$. Clairement $u \leq_{T_i} v$. Si $u <_{T_i} v$, on conclut que $S_1 \prec_{T_i} S$. Supposons maintenant que $u = v$. Comme $S_1 \perp_{T_i} S_2$, on a que : $\mathsf{span}_{T_i}(v, S_1), \mathsf{span}_{T_i}(v, S_2)$ sont disjoints. Comme ces ensembles sont non vides et comme leur union est égale à $\mathsf{span}_{T_i}(v, S)$, on obtient que $\mathsf{span}_{T_i}(v, S_1) \subset \mathsf{span}_{T_i}(v, S)$, et on conclut que $S_1 \prec_{T_i} S$. □

3e point. On va décrire une caractérisation récursive des ensembles $BCT_d(U)$ (Proposition 4.64). On montre d'abord :

Lemme 4.62. *Soit un entier $p \geq 1$. Soient U, S deux arbres tels que U est un p-tuteur de S. Soit T un arbre de degré $d \leq p+1$ tel que $S \trianglelefteq T$. Alors : (i) $\mathsf{lca}_T(U) = \mathsf{lca}_T(S)$, (ii) si $u \geq_T \mathsf{lca}_T(U)$, alors $\mathsf{span}_T(u, U) = \mathsf{span}_T(u, S)$.*

Démonstration. Montrons le point (i). Soit ab une paire couvrante de U, alors ab est également une paire couvrante de S, et donc $\mathsf{lca}_T(U) = \mathsf{lca}_T(S) = \mathsf{lca}_T(a, b)$.

Montrons le point (ii). Soit v un noeud de S, posons $\phi(v) = \mathsf{span}_T(u, S(v))$. On montre dans un premier temps que : si v est un noeud de S de profondeur h, alors $|\phi(v)| \leq \max(d - h, 1)$. On raisonne par récurrence sur h. Le résultat est clair si $h = 0$. Supposons maintenant $h > 0$, alors dans S le noeud v a pour père w et pour frère v', et w est de profondeur $h - 1$. Si $u >_T \mathsf{lca}_T(S(v))$, alors $\mathsf{span}_T(u, S(v))$ est un singleton, et donc $|\phi(v)| = 1$. Supposons maintenant que $u = \mathsf{lca}_T(S(v))$. Comme $u \geq_T \mathsf{lca}_T(U) = \mathsf{lca}_T(S)$, on a donc $u = \mathsf{lca}_T(S(v)) = \mathsf{lca}_T(S(w)) = \mathsf{lca}_T(S)$. Par le Lemme 4.59, $\mathsf{span}_T(u, S(w))$ est l'union disjointe de $\mathsf{span}_T(u, S(v)), \mathsf{span}_T(u, S(v'))$, et comme chaque ensemble est non vide on obtient que $|\phi(v)| < |\phi(w)|$. Mais on a par hypothèse de récurrence $|\phi(w)| \leq d - (h - 1)$, et on conclut que $|\phi(v)| \leq d - h$.

On conclut la preuve de la manière suivante. Partitionnons $L(U)$ en L_1 (les étiquettes de U à profondeur $< p$) et L_2 (les étiquettes de U à profondeur p).

Comme U est un p-tuteur de S, il existe une famille d'arbres $\{S_x : x \in L(U)\}$ tels que (i) chaque arbre S_x contient l'étiquette x, (ii) S est obtenu à partir de U en substituant chaque feuille $x \in L(U)$ par S_x. Dès lors, si $x \in L_2$ alors $\mathsf{span}_T(u, S_x)$ est de cardinal 1 par l'observation ci-dessus, et comme $x \in L(S_x)$ on conclut que $\mathsf{span}_T(u, S_x) = \mathsf{span}_T(u, x)$. On obtient donc :

$$\begin{aligned}\mathsf{span}_T(u, S) &= (\cup_{x\in L_1}\mathsf{span}_T(u, x)) \cup (\cup_{x\in L_2}\mathsf{span}_T(u, S_x)) \\ &= (\cup_{x\in L_1}\mathsf{span}_T(u, x)) \cup (\cup_{x\in L_2}\ \mathsf{span}_T(u, x)) \\ &= \mathsf{span}_T(u, U)\end{aligned}$$

ce qui est le résultat attendu. □

Lemme 4.63. *Soit $p \geq d-1$. Soient $U_1, U_2 \in BCT_p$ tels que $U_1 \perp_{\mathcal{T}} U_2$. Soient S_1, S_2 tels que $S_i \in BCT_p(U_i)$. Alors $S_1 \perp_{\mathcal{T}} S_2$.*

Démonstration. Il suffit d'établir le résultat pour $\mathcal{T} = \{T\}$, le résultat pour le cas général s'en déduisant de manière immédiate.

Soit T de degré d. Considérons deux arbres U_1, U_2 de hauteur $\leq p$ tels que $U_1 \trianglelefteq T, U_2 \trianglelefteq T, U_1 \perp_T U_2$. Considérons S_1, S_2 tels que : (i) U_i est p-tuteur de S_i, (ii) $S_i \trianglelefteq T$. Montrons que $S_1 \perp_T S_2$.

Posons $u = \mathsf{lca}_T(L(S_1) \cup L(S_2))$, on a alors $u = \mathsf{lca}_T(L(U_1) \cup L(U_2))$ par le point (i) du Lemme 4.62. Mais alors, pour $i \in \{1, 2\}$, on a $u \geq_T \mathsf{lca}_T(U_i)$, ce qui implique que $\mathsf{span}_T(u, U_i) = \mathsf{span}_T(u, S_i)$ par le point (ii) du Lemme 4.62. Comme $U_1 \perp_T U_2$, on obtient que $\mathsf{span}_T(u, U_1), \mathsf{span}_T(u, U_2)$ sont disjoints, et on conclut que $\mathsf{span}_T(u, S_1), \mathsf{span}_T(u, S_2)$ sont disjoints. □

Soit $S \in BCT_p$, et soit $S' \in BCT_{p+1}$. On dit que S' *étend* S si et seulement si S est un p-tuteur de S'. On décrit une caractérisation récursive des ensembles $BCT_p(U)$.

Proposition 4.64. *Soit $p \geq d$. Soit $U = (U_1, U_2) \in BCT_p$. Soit S un arbre sur $L' \subseteq L$. Les points suivants sont équivalents :*

- *$S \in BCT_p(U)$;*
- *$S = (S_1, S_2)$ et il existe V_1, V_2, V_i étendant U_i, tels que $S_i \in BCT_p(V_i)$.*

Démonstration. ($\Rightarrow$) : comme $S \in BCT_p(U)$, on a $S = (S_1, S_2)$ avec $S_i \in BCT_{p-1}(U_i)$. Par le point 2 du Lemme 4.56, on peut trouver V_i étendant U_i qui soit un p-tuteur de S_i. Alors $S_i \in BCT_p(V_i)$.

($\Leftarrow$) : supposons qu'il existe V_1, V_2, V_i étendant U_i, tel que $S_i \in BCT_p(V_i)$. Alors $S_i \in BCT(\mathcal{T})$, et V_i est un p-tuteur de S_i. Par le point 1 du Lemme 4.56, U_i est un $p-1$-tuteur de S_i. Il résulte par le Lemme 4.55 que U est un p-tuteur de S. Comme $U \in BCT(\mathcal{T})$, par le Lemme 4.60 on a $U_1 \perp_{\mathcal{T}} U_2$. Comme $S_i \in BCT_{p-1}(U_i)$ et comme $p - 1 \geq d - 1$, par le Lemme 4.63 on a $S_1 \perp_{\mathcal{T}} S_2$. On déduit que $S \in BCT(\mathcal{T})$ par le Lemme 4.60.

On a donc montré : (i) $S \in BCT(\mathcal{T})$, (ii) U est un p-tuteur de S. On conclut que $S \in BCT_p(U)$. □

On est maintenant en mesure de prouver la Proposition 4.54.

Preuve de la Proposition 4.54. La Proposition 4.64 fournit les relations de récurrence suivantes permettant de calculer les valeurs $MCT_d(U)$ pour $U \in BCT_d$:

$$\begin{cases} MCT_d(U) & = 1 \text{ si } |U| = 1 \\ MCT_d(U) & = \max\{MCT_d(V_1) + MCT_d(V_2) : \\ & \quad V_1, V_2 \in BCT_d, V_i \text{ étendant } U_i\} \text{ si } U = (U_1, U_2) \end{cases} \tag{4.3}$$

L'algorithme pour MCT procède en quatre étapes :

1. on calcule l'ensemble $R = \cup_{T_i \in \mathcal{T}} rt(T_i)$;
2. on calcule l'ensemble BCT_d ;
3. on calcule les valeurs $MCT_d(U)$ pour chaque $U \in BCT_d$, en utilisant 4.3 ;
4. on obtient $MCT(\mathcal{T}) = \max_{U \in BCT_d} MCT_d(U)$.

A l'étape 2, on énumère les arbres T de hauteur $\leq d$ tels que $L(T) \subseteq L$, et on retient les arbres compatibles. On teste si T est un arbre compatible de $\mathcal{T}$ en vérifiant que pour tous $x, y, z \in L(T)$ distincts, si $xy|z \in rt(T)$ alors $xz|y \notin R, yz|x \notin R$.

A l'étape 3, les valeurs $MCT_d(U)$ sont calculées par programmation dynamique selon la relation $\prec_{\mathcal{T}}$. Ceci est permis par le fait que si $U = (U_1, U_2)$ et si $V_i \in BCT_d$ étend U_i, alors $V_i \prec_{\mathcal{T}} U$. En effet, le Lemme 4.62 implique que $V_i \equiv_{\mathcal{T}} U_i$, le Lemme 4.63 implique que $U_i \prec_{\mathcal{T}} U$, et on conclut que $V_i \prec_{\mathcal{T}} U$.

Justifions que l'algorithme s'exécute en temps $O(n^{2^{d+1}} + kn^3)$. L'étape 1 s'effectue en temps $O(kn^3)$, en tabulant l'ensemble R. L'étape 2 s'effectue en temps $O(n^{2^d})$: on énumère les arbres T de hauteur $\leq d$ tels que $L(T) \subseteq L$, pour chaque arbre on teste si c'est un arbre compatible de $\mathcal{T}$ en temps $O(|T|^3) = O(1)$. L'étape 3 s'effectue en temps $O(n^{2^{d+1}})$: en effet, pour un arbre U donné, calculer $MCT_d(U)$ par l'équation 4.3 nécessite de calculer pour chaque $i \in \{1, 2\}$ le maximum des valeurs $MCT_d(V_i)$ pour V_i étendant U_i, ce qui prend un temps $O(|BCT_d|)$; calculer toutes les valeurs $MCT_d(U)$ prend donc un temps $O(|BCT_d|^2) = O(n^{2^{d+1}})$. Enfin, l'étape 4 s'effectue en temps $O(|BCT_d|) = O(n^{2^d})$. □

Algorithmes FPT

On décrit des algorithmes FPT pour MCT, pour les paires de paramètres (k, d) et (k, q).

Proposition 4.65. *1.* MCT *est soluble en temps* $O(2^{2kd} n^3)$ *;*

2. MCT *est soluble en temps* $O(2^{O(kq)} n^3)$.

Le principe est semblable aux algorithmes FPT pour MAST décrits en Section 4.1.3 : on se ramène à une version colorée de MCT, appelée COLORED-MCT, qui consiste à rechercher un *arbre compatible coloré*.

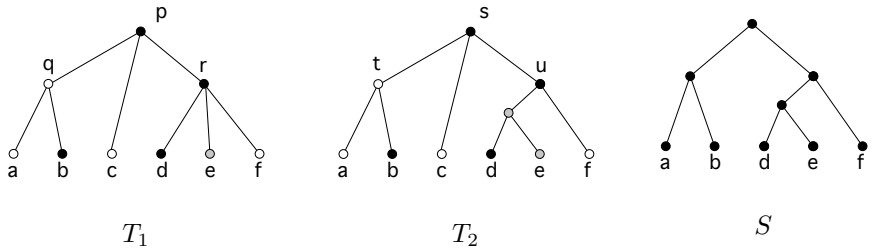
Soit un arbre coloré T sur L, et un arbre binaire S sur $L' \subseteq L$, un *préplongement coloré* de S dans T est une application $\phi : N(S) \to N(T)$ telle que :

1. ϕ est un préplongement de S dans T (vu comme arbre non coloré) ;
2. si u, v, v' sont trois noeuds de S tels que $\phi(v), \phi(v') <_T \phi(u)$, alors les noeuds $\mathsf{child}_T(\phi(u), \phi(v))$ et $\mathsf{child}_T(\phi(u), \phi(v'))$ sont égaux ou de couleurs distinctes.

On dit que S est un *pré-sous-arbre coloré* de T (noté $S \trianglelefteq_c T$) si et seulement si il existe un préplongement coloré de S dans T. Etant donnée une collection colorée $\mathcal{T} = \{T_1, ..., T_k\}$ sur L et un arbre binaire S sur $L' \subseteq L$, on dit que S est un *arbre compatible coloré* de $\mathcal{T}$ si et seulement si $S \trianglelefteq_c T_i$ pour tout i.

On note $BCT_c(\mathcal{T})$ l'ensemble des arbres compatibles colorés de $\mathcal{T}$, et $MCT_c(\mathcal{T})$ la taille maximum d'un arbre compatible coloré de $\mathcal{T}$. Le problème Colored-Mct consiste, étant donnée une collection colorée $\mathcal{T}$, à trouver un arbre d'accord coloré de taille maximum $MCT_c(\mathcal{T})$.

La figure suivante illustre ces définitions :



T_1 T_2 S

Fig. 4.13 – Une collection colorée $\mathcal{T} = \{T_1, T_2\}$ sur un ensemble d'étiquettes $L = \{a, b, c, d, e, f\}$ et sur un ensemble de trois couleurs (blanc, gris, noir). S est un arbre compatible coloré de taille maximum.

On va montrer que Colored-Mct peut se résoudre en temps $O(2^{2kc}n^3)$ où c est le nombre de couleurs. Ceci implique la Proposition 4.65, la preuve étant analogue à celle de la Proposition 4.26 en Section 4.1.3.

Proposition 4.66. Colored-Mct *est soluble en temps* $O(2^{2kc}n^3)$.

L'algorithme utilise des idées similaires à l'algorithme pour Colored-Mast. Il utilise la programmation dynamique sur les tuples (a, b, ϕ) où ϕ est un filtre associé au $\mathsf{lca}^{\mathcal{T}}$-tuple induit par a, b.

Notons $BCT_c(\mathcal{T})$ l'ensemble des arbres compatibles colorés pour $\mathcal{T}$. On a alors la caractérisation récursive suivante des arbres compatibles colorés, analogue à la caractérisation du Lemme 4.59 pour les arbres compatibles non colorés.

Lemme 4.67. *Supposons que* $S = (S_1, S_2)$. *Les points suivants sont équivalents :*

1. $S \in BCT_c(\mathcal{T})$;
2. $S_1 \in BCT_c(\mathcal{T}), S_2 \in BCT_c(\mathcal{T})$, *et si l'on pose* $\pi = \mathsf{lca}^{\mathcal{T}}(S)$, *alors* $(\mathsf{cspan}^{\mathcal{T}}(\pi, S_1), \mathsf{cspan}^{\mathcal{T}}(\pi, S_2))$ *est une partition de* $\mathsf{cspan}^{\mathcal{T}}(\pi, S)$.

Etant donnés une paire $a, b \in L$ induisant un $\mathsf{lca}^{\mathcal{T}}$-tuple π, et un π-filtre ϕ, on définit deux ensembles d'arbres $BCT_1(a, b, \phi)$ et $BCT_2(a, b, \phi)$:

- $BCT_1(a, b, \phi)$ (défini si ϕ est épais) est l'ensemble des arbres S tels que (i) $S \in BCT_c(\mathcal{T})$, (ii) $a, b \in L(S)$, (iii) $\mathsf{lca}^{\mathcal{T}}(S) = \pi$, (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$.
- $BCT_2(a, b, \phi)$ (défini si $a \neq b$) est l'ensemble des arbres S tels que (i) $S \in BCT_c(\mathcal{T})$, (ii) $a \in L(T)$, (ii) $\mathsf{lca}^{\mathcal{T}}(S) \preceq_{\mathcal{T}} \pi$, (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi, S) = \phi$.

Le lemme suivant donne une caractérisation récursive des ensembles $BCT_1(a, b, \phi)$.

Lemme 4.68. *Soit $\pi = \mathsf{lca}^{\mathcal{T}}(a, b)$, et soit ϕ un π-filtre épais. Les points suivants sont équivalents :*

1. *$S \in BCT_1(a, b, \phi)$;*
2. *il existe (ϕ_1, ϕ_2) partition de ϕ, $S_1 \in AST_2(a, b, \phi_1)$, $S_2 \in AST_2(b, a, \phi_2)$ tels que $S = (S_1, S_2)$.*

Démonstration. Analogue à la preuve du Lemme 4.29, en utilisant cette fois le Lemme 4.67 au lieu du Lemme 4.28. □

Le lemme suivant donne une caractérisation récursive des ensembles $BCT_2(a, b, \phi)$. Soit ϕ un π-filtre. On note $S(\pi, \phi)$ l'ensemble des $\mathsf{lca}^{\mathcal{T}}$-tuples π' tels que : (i) pour chaque i tel que $\phi[i]$ est un singleton $\{x_i\}$, $\pi'[i] \leq_{T_i} x_i$, (ii) pour chaque i tel que $|\phi[i]| \geq 2$, $\pi'[i] = \pi[i]$. Soit $\pi' \in S(\pi, \phi)$, on note $SF(\pi, \phi, \pi')$ l'ensemble des π'-filtres ϕ' qui coïncident avec ϕ pour les indices i tels que $|\phi[i]| \geq 2$.

Lemme 4.69. *Soit $\pi = \mathsf{lca}^{\mathcal{T}}(a, b)$, et soit ϕ un π-filtre. Les points suivants sont équivalents :*

1. *$S \in BCT_2(a, b, \phi)$;*
2. *il existe $c \in L$ tel que a, c induise un $\mathsf{lca}^{\mathcal{T}}$-tuple π', et il existe un π'-filtre ϕ', tels que (a) $\pi' \in S(\pi, \phi)$, (b) $\phi' \in SF(\pi, \phi, \pi')$, (c) $S \in BCT_1(a, c, \phi')$.*

Démonstration. (1) $\Rightarrow$ (2) : supposons que (1) $S \in BCT_2(a, b, \phi)$. Alors S admet une paire couvrante a, c (avec c éventuellement égal à a). Posons $\pi' = \mathsf{lca}^{\mathcal{T}}(a, c)$ $(= \mathsf{lca}^{\mathcal{T}}(S))$ et $\phi' = \mathsf{cspan}^{\mathcal{T}}(\pi', S)$.

Montrons que (a) est vérifié. Soit $i \in [k]$ tel que $|\phi[i]| \geq 2$, on montre que $\pi'[i] = \pi[i]$. Comme $\mathsf{lca}_{T_i}(S) \leq_{T_i} \pi[i]$ et $\mathsf{cspan}_{T_i}(\pi[i], S) = \phi[i]$ est de cardinal ≥ 2, on conclut que $\mathsf{lca}_{T_i}(S) = \pi[i]$, et donc $\pi[i] = \pi'[i]$.

Montrons que (b) est vérifié. Soit $i \in [k]$ tel que $|\phi[i]| \geq 2$, on montre que $\phi'[i] = \phi[i]$. En effet, on a alors $\pi'[i] = \pi[i]$, et $\phi'[i] = \mathsf{cspan}_{T_i}(\pi'[i], S) = \mathsf{cspan}_{T_i}(\pi[i], S) = \phi[i]$.

Montrons que (c) est vérifié. Le point (i) $S \in BCT_c(\mathcal{T})$ résulte de (1). Les points (ii) $a, c \in L(S)$ et (iii) $\mathsf{lca}^{\mathcal{T}}(S) = \pi'$ résultent de la définition de c. Le point (iv) $\mathsf{cspan}^{\mathcal{T}}(\pi', S) = \phi'$ résulte de la définition de ϕ'.

(2) $\Rightarrow$ (1) : supposons qu'il existe $c \in L$ vérifiant les hypothèses. Alors (a) $\pi' \in S(\pi, \phi)$, (b) $\phi' \in SF(\pi, \phi, \pi')$, (c) $S \in BCT_1(a, c, \phi')$. Montrons que $S \in BCT_2(a, b, \phi)$. En effet, les points (i) $S \in BCT_c(\mathcal{T})$, (ii) $a \in L(S)$ résultent de (c). Le point (iii) $\mathsf{lca}^{\mathcal{T}}(S) \preceq_{\mathcal{T}} \pi$ résulte de (c) et du fait que $\pi' \preceq_{\mathcal{T}} \pi$. Justifions le point (iv). Soit $i \in [k]$, montrons que $\mathsf{cspan}_{T_i}(\pi[i], S) = \phi[i]$. On sait que $\mathsf{cspan}_{T_i}(\pi'[i], S) = \phi'[i]$ par (c). On a les cas suivants :

– si $|\phi[i]| \geq 2$: résulte de $\pi'[i] = \pi[i]$ (par (a)) et $\phi'[i] = \phi[i]$ (par (b)) ;
– si $|\phi[i]| = 1$: alors $\phi'[i]$ est un singleton $\{x_i\}$, et on a $\pi'[i] \leq_{T_i} x_i$ (par (a)), donc $\mathsf{cspan}_{T_i}(\pi[i], S) = \{x_i\} = \phi[i]$.

On conclut que $S \in BCT_2(a, b, \phi)$ comme annoncé. □

On est maintenant en mesure de montrer :

Preuve de la Proposition 4.66. Définissons $MCT_i(a, b, \phi)$ comme la taille d'un plus grand arbre de $BCT_i(a, b, \phi)$. Alors les lemmes 4.68 et 4.60 fournissent les relations de récurrence suivantes :

$$\begin{cases} \text{si } a = b : & MCT_1(a, b, \phi) = 1 \\ \text{si } a \neq b : & MCT_1(a, b, \phi) = \\ & \max\{MCT_2(a, b, \phi') + MCT_2(b, a, \phi'') : \\ & (\phi', \phi'') \text{ partition de } \phi\} \end{cases} \tag{4.4}$$

$$\begin{cases} \text{si } a \neq b : & MCT_2(a, b, \phi) = \\ & \max\{MCT_1(a, c, \phi') : \\ & c \in L, a, c \text{ induisent un } \mathsf{lca}^{\mathcal{T}}\text{-tuple } \pi' \\ & \text{tel que } \pi' \in S(\pi, \phi), \phi' \in SF(\pi, \phi, \pi')\} \end{cases} \tag{4.5}$$

On obtient $MCT_c(\mathcal{T})$ comme le maximum des valeurs $MCT_1(a, b, \phi)$ pour $a, b \in L$, ϕ $\mathsf{lca}^{\mathcal{T}}(a, b)$-filtre. On obtient ainsi un algorithme pour MCT, de complexité temporelle $O(2^{2kd} n^3)$: l'analyse est la même que dans le cas de l'algorithme pour COLORED-MAST. □

4.2.4 Résultats négatifs

Approximabilité.

On donne des résultats d'inapproximabilité pour le problème sur des arbres de degré non borné. On montre que : (i) le problème MCT pour un nombre d'arbres non borné est aussi difficile à approximer que MAXIMUM INDEPENDENT SET (Proposition 4.71), (ii) le problème $2 -$ MCT est APX-dur (Proposition 4.73) et n'est pas approximable à un facteur constant (Proposition 4.74), (ii) le problème complémentaire $2 -$ CMCT est APX-dur (Proposition 4.75). Ces résultats apparaissent dans [7].

On montre dans un premier temps que MCT est aussi difficile à approximer que MAXIMUM INDEPENDENT SET (Proposition 4.71). Pour les besoins de la preuve, on introduit le problème intermédiaire MAXIMUM INDUCED DEGREE ONE SUBGRAPH (MIDS) : étant donné un graphe $G = (V, E)$, chercher $V' \subseteq V$ de cardinal maximum tel que $G[V']$ soit de degré maximum ≤ 1. On montre :

Lemme 4.70. *Pour tout $\epsilon < 1$,* MIDS *n'est pas approximable à $n^{1-\epsilon}$ si* $\mathsf{P} \neq \mathsf{NP}$.

Démonstration. On donne une réduction de MIS à MIDS. Soit $G = (V, E)$ une instance de MIS, on construit le graphe G' en créant 2 copies v^1, v^2 de chaque $v \in V$ reliées par une arête. Formellement, $G' = (V', E')$ où

$$V' = \{v^1, v^2 : v \in V\}$$
$$E' = E'_1 \cup E'_2, E'_1 = \{u^i v^j : uv \in E, i, j \in \{1, 2\}\}, E'_2 = \{v^1 v^2 : v \in V\}$$

Montrons que : $\text{MIDS}(G') = 2\text{MIS}(G)$.

Supposons que I est un indépendant de G. Alors $I' = \{v^1, v^2 : v \in V\}$ induit un sous-graphe de G' de degré maximum ≤ 1. Donc $\text{MIDS}(G') \geq 2\text{MIS}(G)$.

Supposons que $V' \subseteq V$ induit un sous-graphe de G' de degré maximum ≤ 1. Soient $C_1, ..., C_m$ les composantes connexes de $G'[V']$. Pour chaque $i \in [m]$, choisissons un sommet $v_i \in V$ tel que $C_i \cap \{v_i^1, v_i^2\} \neq \emptyset$. Observons que si $i \neq j$, alors : (i) $v_i \neq v_j$ (en raison des arêtes de E'_2), (ii) $v_i v_j \notin E$ (en raison des arêtes de E'_1). Soit $I = \{v_1, ..., v_m\}$, alors I est un indépendant de G de cardinal $m \geq \frac{|V'|}{2}$. Donc $\text{MIDS}(G') \leq 2\text{MIS}(G)$.

On conclut en observant que : si MIDS était approximable à $n^{1-\epsilon}$ pour un certain $\epsilon < 1$, alors MIS serait approximable à $\frac{(2n)^{1-\epsilon}}{2}$ et donc à $n^{1-\epsilon}$, ce qui impliquerait que $\mathsf{P} = \mathsf{NP}$ par le résultat de [38]. □

Proposition 4.71. *Pour tout $\epsilon < 1$,* Mct *n'est pas approximable à $n^{1-\epsilon}$ si $\mathsf{P} \neq \mathsf{NP}$.*

Démonstration. On donne une réduction linéaire de MIDS à Mct. Soit $G = (V, E)$ une instance de MIDS, on construit la collection $\mathcal{T}$ sur l'ensemble d'étiquettes V comportant, pour chaque $e = \{x, y\} \in E$, l'arbre T_e défini comme suit : sa racine a pour fils (i) pour chaque $z \in V \backslash e$, une feuille d'étiquette z, (ii) un noeud interne ayant pour fils deux feuilles d'étiquettes x, y.

La correction de la réduction résulte du fait que : pour tout $V' \subseteq V$, $G[V']$ est de degré maximum ≤ 1 si et seulement si $\mathcal{T}|V'$ est compatible.

Soit $V' \subseteq V$ tel que $G[V']$ soit de degré maximum ≤ 1. Alors $G[V']$ a des composantes connexes de cardinal 2, $\{x_1, y_1\}, ..., \{x_p, y_p\}$, et des composantes connexes de cardinal 1, $z_1, ..., z_q$. Soit $T = ((x_1, y_1), ..., (x_p, y_p), z_1, ..., z_q)$, alors T est un arbre compatible de $\mathcal{T}$. En effet, pour chaque $e \in E$, $T \trianglelefteq T_e$ résulte du fait que (i) soit V' contient au plus une extrémité de e, (i) soit $e = x_i y_i$ pour un certain $i \in [p]$. On conclut que $\mathcal{T}|V'$ est compatible.

Soit $V' \subseteq V$ tel que $\mathcal{T}|V'$ soit compatible. Soit $x \in V'$, on montre que $G[V']$ contient au plus une arête incidente à x. Supposons par contradiction que $G[V']$ contienne deux arêtes $e = xy, f = xz$ avec $y, z \in V'$. On a alors $xy|z \in rt(T_e), xz|y \in rt(T_f)$, contredisant le fait que $\mathcal{T}|V'$ est compatible. On conclut que $G[V']$ est de degré maximum ≤ 1. □

On va maintenant établir l'APX-difficulté de $2 -$ Mct (Proposition 4.73). On introduit le problème intermédiaire Maximum Star Forest (MSF) défini comme suit. Un graphe G est une *forêt étoilée* si et seulement si chaque composante connexe de G est une étoile ; de manière équivalente, G est une forêt étoilée si et seulement si il ne contient pas de chemin de longueur 3. Le problème

MSF consiste, étant donné un graphe G, à chercher une forêt étoilée couvrante de G comportant le maximum d'arêtes. On note MSFB $-\ \Delta$ la restriction de MSF aux graphes bipartis de degré maximum $\leq \Delta$.

Le problème MSF et ses variantes ont été considérés dans [19, 16, 35]. On montre :

Lemme 4.72. MSFB $-$ 3 *est* APX*-dur.*

Démonstration. Rappelons la définition du problème Minimum Dominating Set (MDS). Etant donné un graphe $G = (V, E)$, un *ensemble dominateur* de G est un ensemble $D \subseteq V$ tel que tout $x \in V \backslash D$ a un voisin dans D. Le problème MDS consiste, étant donné G, à chercher un ensemble dominateur de cardinal minimum. On note MDSB $-\ \Delta$ la restriction de MDS aux graphes bipartis de degré maximum $\leq \Delta$.

Soit $G = (V, E)$ un graphe, soit $n = |V|$. On remarque le lien suivant entre les ensembles dominateurs et les forêts étoilées de G :

Fait :

(i) à partir de F forêt étoilée couvrante de G, on peut construire D ensemble dominateur de G tel que $|D| = n - |F|$;

(ii) à partir de D ensemble dominateur de G, on peut construire F forêt étoilée couvrante de G telle que $|F| = n - |D|$.

(iii) $\mathsf{opt}' = n - \mathsf{opt}$.

Preuve : Point (i) : soit F une forêt étoilée couvrante de G. Soient $C_1, ..., C_m$ les composantes connexes de F. Chaque C_i est une étoile : on choisit $v_i \in C_i$ centre de l'étoile (c'est l'unique sommet de degré ≥ 2 si $|C_i| \geq 3$, c'est un sommet arbitraire de C_i sinon). Alors $D = \{v_1, ..., v_m\}$ est un ensemble dominateur de G, tel que $|D| = n - |F|$.

Point (ii) : soit D un ensemble dominateur de G. On construit une forêt étoilée par l'algorithme suivant : (i) démarrer avec $F = \emptyset$, (ii) pour chaque $x \in V \backslash D$, choisir $y \in D$ qui domine x, et ajouter l'arête xy à F. Clairement, l'ensemble d'arêtes ainsi obtenu forme une forêt étoilée couvrante de G, telle que $|F| = n - |D|$.

Le point (iii) est une conséquence immédiate des points (i) et (ii). □

On montre alors que l'identité est une L-réduction de MDSB $-\ \Delta$ à MSFB $-\ \Delta$:

- il existe α tel que $\mathsf{opt}' \leq \alpha \times \mathsf{opt}$: en effet, on a $\mathsf{opt} \geq \frac{n}{\Delta+1}$ puisque G est de degré maximum $\leq \Delta$. Comme $\mathsf{opt}' = n - \mathsf{opt}$, on obtient que $\mathsf{opt}' \leq \Delta \times \mathsf{opt}$.
- il existe β tel que : à partir d'une forêt étoilée F de G, on peut construire en temps polynomial un ensemble dominateur D de G telle que $|\mathsf{opt} - \mathsf{m}(D)| \leq \beta \times |\mathsf{opt}' - \mathsf{m}'(F)|$. En effet, étant donnée F forêt étoilée de G, par le point (i) ci-dessus on peut construire en temps polynomial D ensemble dominateur de G tel que $|D| = n - |F|$. Alors $\mathsf{m}(D) - \mathsf{opt} = |D| - \mathsf{opt} = (n - |F|) - (n - \mathsf{opt}') = \mathsf{opt}' - \mathsf{m}'(F)$.

On a donc bien une L-réduction. On conclut en utilisant le fait que MDSB − 3 est APX-dur [13]. □

Proposition 4.73. $2-$ MCT *est* APX*-dur.*

Démonstration. On donne une réduction linéaire de MSFB $-\Delta$ à $2-$MCT. Soit un graphe biparti $G=(V,E)$, de bipartition V_1, V_2, et de degré maximum $\leq \Delta$. On construit une collection $\mathcal{T}=\{T_1,T_2\}$ sur l'ensemble d'étiquettes $L=E$. L'arbre T_i est défini comme suit :

- sa racine a pour fils les noeuds u_x^i $(x \in V_i)$;
- chaque noeud u_x^i a les fils suivants : pour chaque arête $e \in E$ incidente à x, une feuille d'étiquette e.

On montre que : pour tout $F \subseteq E$, les arêtes de F induisent une forêt étoilée si et seulement si $\mathcal{T}|F$ est compatible. On raisonne par contraposée, en montrant que : $\mathcal{T}|F$ est incompatible si et seulement si les arêtes de F contiennent un chemin de longueur 3.

Supposons que $\mathcal{T}|F$ est incompatible : il existe alors $e,f,g \in F$ tels que $ef|g \in rt(T_1), e|fg \in rt(T_2)$. Comme $ef|g \in rt(T_1)$, dans T_1 les feuilles d'étiquettes e,f sont filles d'un sommet u_x^1, et la feuille d'étiquette g est fille d'un sommet u_y^1 avec $y \neq x$. De même, comme $e|fg \in rt(T_2)$, dans T_2 les feuilles d'étiquettes f,g sont filles d'un sommet u_z^2, et la feuille d'étiquette e est fille d'un sommet u_w^2 avec $w \neq z$. On conclut que $e=wx, f=xz, g=zy$ forment un chemin de longueur 3.

Réciproquement, supposons que F contient un chemin de longueur 3. Ce chemin est formé de trois arêtes $e=wx, f=xz, g=zy$ avec $x,y \in V_1, w,z \in V_2$. Dès lors, dans T_1 les feuilles d'étiquettes e,f sont filles de u_x^1, et la feuille d'étiquette g est fille de u_y^1, donc $ef|g \in rt(T_1)$. De même, dans T_2, les feuilles d'étiquettes f,g sont filles de u_z^2, et la feuille d'étiquette e est fille de u_w^2, donc $e|fg \in rt(T_2)$. On obtient que $ef|g \in rt(T_1), e|fg \in rt(T_2)$, et donc $\mathcal{T}|F$ est incompatible. □

Par un raisonnement analogue à la preuve de la Proposition 4.33 pour $3-$ MAST, on montre que $2-$MCT n'est pas approximable à un facteur constant.

Proposition 4.74. *On a les résultats d'inapproximabilité suivants pour* $2-$ MCT *: (i) inapproximabilité à un facteur constant sauf si* $\mathsf{P}=\mathsf{NP}$*, (ii) inapproximabilité à* $2^{(\log n)^\alpha}$ *pour tout* $\alpha<1$*, sauf si* $\mathsf{NP} \subseteq DTIME(2^{\mathsf{polylog}(n)})$.

Démonstration. On utilise la technique d'auto-amélioration de manière similaire à la preuve de la Proposition 4.33. En définissant le produit de deux collections $\mathcal{T},\mathcal{T}'$ comme en Proposition 4.33, on a les propriétés suivantes, dont la preuve est analogue à celle donnée dans la Proposition 4.33 :

Fait.

(i) $S \trianglelefteq T \otimes T'$ si et seulement si il existe $x_1,...,x_m \in L(T)$, un arbre $S_0[X_1,...,X_m]$ et des arbres $S_1,...,S_m$ tels que (i) $S_0[x_1,...,x_m] \trianglelefteq T$, (ii) pour tout $i \in [m]$, $S_i \trianglelefteq T'$, (iii) $S=S_0[x_1 \otimes S_1,...,x_m \otimes S_m]$.

(ii) si S est un arbre compatible de $\mathcal{T}$ et si S' est un arbre compatible de $\mathcal{T}'$, alors $S \times S'$ est un arbre compatible de $\mathcal{T} \otimes \mathcal{T}'$;
(iii) si S'' est un arbre compatible de $\mathcal{T} \otimes \mathcal{T}'$, alors on peut obtenir en temps polynomial S arbre compatible de $\mathcal{T}$, S' arbre compatible de $\mathcal{T}'$ tel que $|S''| \leq |S| \times |S'|$;
(iv) $MCT(\mathcal{T} \otimes \mathcal{T}') = MCT(\mathcal{T}) \times MCT(\mathcal{T}')$.

Les résultats d'inapproximabilité s'obtiennent alors par le même raisonnement qu'en Proposition 4.33. □

On considère à présent le problème complémentaire CMCT. On montre qu'il est APX-dur, même pour deux arbres.

Proposition 4.75. $2 - \text{CMCT}$ *est* APX*-dur.*

Démonstration. On montre que la réduction utilisée dans la preuve de la Proposition 4.73 est une L-réduction de $\text{MSFB} - \Delta$ à $2 - \text{CMCT}$. Soit $m = |E|$, alors $\mathsf{m}'(T) = m - |T|$ si T est un arbre compatible de $\mathcal{T}$, et $\mathsf{opt}' = m - \mathsf{opt}$.

Observons que $\mathsf{opt} \geq \frac{m}{\Delta}$. En effet, considérons la forêt étoilée F construite par l'algorithme suivant : (i) démarrer avec $F = \emptyset$, (ii) pour chaque sommet $x \in V_1$, ajouter à F les arêtes $xy \in E$ non adjacentes à une arête déjà présente dans F. Montrons que $|F| \geq \frac{m}{\Delta}$. Pour chaque arête $e \in F$ notons E_e l'ensemble des arêtes $e' \in E$ adjacentes au même sommet de V_2, alors

- $\cup_{e \in F} E_e = E$: en effet, considérons une arête $e' = xy \in E$. Si $e' \in F$, alors $e' \in E_e$ avec $e = e'$. Supposons que $e' \notin F$, on montre qu'il existe $e \in F$ tel que $e' \in E_e$. Si ce n'était pas le cas, alors F ne contiendrait aucune arête incidente à y, mais alors lors de l'examen de x par l'algorithme ci-dessus e' serait ajouté à F, contradiction.
- chaque ensemble E_e est de cardinal $\leq \Delta$.

On conclut que $|E| \leq \Delta \times |F|$, et donc $|F| \geq \frac{m}{\Delta}$. Ceci implique que $\mathsf{opt} \geq \frac{m}{\Delta}$ comme annoncé.

Justifions maintenant que la réduction est une L-réduction :

- il existe α tel que $\mathsf{opt}' \leq \alpha \times \mathsf{opt}$: en effet, comme $\mathsf{opt}' = m - \mathsf{opt}$ et $\mathsf{opt} \geq \frac{m}{\Delta}$, on obtient que $\mathsf{opt}' \leq (\Delta - 1) \times \mathsf{opt}$.
- il existe β tel que : à partir d'une solution T de CMCT, on peut construire en temps polynomial une solution F de MSFB telle que $|\mathsf{opt} - \mathsf{m}(F)| \leq \beta \times |\mathsf{opt}' - \mathsf{m}'(F)|$. En effet, on a vu dans la preuve de la Proposition 4.73 que si T est un arbre compatible de $\mathcal{T}$, alors $L(T)$ est un ensemble d'arêtes formant une forêt étoilée de G. A partir de T arbre compatible, on obtient donc F forêt étoilée telle que $|F| = |T|$, et par suite : $\mathsf{opt} - \mathsf{m}(F) = \mathsf{opt} - |F| = \mathsf{opt} - |T| = (m - \mathsf{opt}') - (m - \mathsf{m}'(T)) = \mathsf{m}'(T) - \mathsf{opt}$.

On a donc bien une L-réduction. □

Complexité paramétrique.

On utilise les outils de la Section 2.3.3 pour montrer que $\text{MCT}[2^{\lceil d/2 \rceil}, q]$ est $\mathsf{W_I}[1]$-dur (Proposition 4.77). Par la Proposition 2.18, cela implique que MCT

n'est pas soluble en temps $\Phi(d)N^{o(2^{d/2})}$ sous l'hypothèse ETH (où N est la taille de l'instance). Ceci suggère donc l'optimalité de l'algorithme décrit en Proposition 4.54.

Pour les besoins de la réduction, on introduit un problème intermédiaire. Etant donné un graphe k-parti $G = (V, E)$, un *indépendant 2-partitionné* de G est un indépendant de G qui contient exactement deux éléments dans chaque classe. Le problème 2-Partitioned-Independent-Set (Pis2) demande : étant donné un entier k, et un graphe k-parti G, décider si G a un indépendant 2-partitionné.

On montre dans un premier temps :

Lemme 4.76. Pis2$[k]$ *est* $\mathsf{W_I}[1]$*-dur.*

Démonstration. On donne une fpt-réduction linéaire depuis Pis$[k]$. Soit $I = (G, k)$ une instance de Pis$[k]$, où G est un graphe k-parti avec les classes $V_1, ..., V_k$. On construit une instance $I' = (G', k)$ de Pis2$[k]$ comme suit. G' est obtenu à partir de G en ajoutant un nouveau sommet v_i à chaque classe V_i, obtenant ainsi une nouvelle classe V'_i. On vérifie que : G a un indépendant partitionné si et seulement si G' a un indépendant 2-partitionné.

Clairement, si $I = \{x_1, ..., x_k\}$ est un indépendant partitionné de G, alors $I' = \{x_1, v_1, ..., x_k, v_k\}$ est un indépendant 2-partitionné de G'. Supposons maintenant que $I' = \{x_1, y_1, ..., x_k, y_k\}$ est un indépendant 2-partitionné de G', avec $x_i, y_i \in V'_i$ pour tout i. Sans perte de généralité, on peut supposer que $x_i \neq v_i$ pour tout i, et donc $I = \{x_1, ..., x_k\}$ est un indépendant partitionné de G. □

On montre maintenant :

Proposition 4.77. Mct$[2^{\lceil d/2 \rceil}, q]$ *est* $\mathsf{W_I}[1]$*-dur.*

Démonstration. On donne une réduction qui envoie chaque instance $I = (G, k)$ de Pis2$[k]$ sur une instance $I' = (\mathcal{T}, k', q)$ de Mct$[2^{d/2}, q]$ telle que $\mathcal{T}$ soit de degré max $\leq d = 2\lceil \log_2 k \rceil + 1$; on pose $k' = 2^{\lceil d/2 \rceil}$ et $q = 2k$.

Construction. Soit $G = (V, E)$ un graphe k-parti, avec les classes $V_1, ..., V_k$. On construit l'instance suivante de Mct. L'ensemble d'étiquettes est $L := V$. La collection est $\mathcal{T} := \{P, P'\} \cup \{Q_e : e \in E\}$, où P, P' sont les *composantes de contrôle*, et les Q_e sont les *composantes de sélection*.

Ces arbres sont construits comme suit. Soit B un arbre binaire équilibré à k feuilles étiquetées $1, ..., k$, de hauteur $h \leq \lceil \log_2 k \rceil$. Introduisons la notation suivante : si $T_1, ..., T_k$ sont k arbres, alors $B[T_1, ..., T_k]$ désigne l'arbre obtenu en substituant chaque feuille i par T_i dans B. Pour chaque i, soit $<_{V_i}$ un ordre total sur V_i, et soit $R_i := rake(V_i, <_{V_i})$ et $R'_i := rake(V_i, >_{V_i})$. Pour chaque x, y notons $S_{x,y} = (x, y)$. On pose alors $P := B[R_1, ..., R_k]$ et $P' := B[R'_1, ..., R'_k]$. Maintenant, pour chaque $e = \{x, y\} \in E$ on définit Q_e comme suit. Supposons que $x \in V_i, y \in V_j$, alors $i \neq j$. Soit $p_{i,j}$ le chemin joignant les sommets i, j (racines de R_i, R_j) dans P. Notons e_i, e_j les arêtes de $p_{i,j}$ incidentes à i, j. Soit $P_{i,j}$ obtenu à partir de P en contractant les arêtes de $p_{i,j}$, à l'exception de e_i, e_j.

Dans $P_{i,j}$, les arêtes e_i, e_j sont incidentes à un même sommet $\lambda_{i,j}$. Alors Q_e est l'arbre obtenu à partir de $P_{i,j}$ en supprimant x, y, et en ajoutant $S_{x,y}$ comme sous-arbre fils de $\lambda_{i,j}$.

Correction. La construction se fait clairement en temps polynomial. De plus, observons que : P, P' sont de degré 2, et que pour chaque e l'arbre Q_e est de degré $\leq 2h + 1 \leq d$.

Pour établir que la réduction est correcte, on montre que :

Fait. G a un indépendant 2-partitionné si et seulement si $\text{MCT}(\mathcal{T}) \geq 2k$.

($\Rightarrow$) : supposons que $I = \{x_1, y_1, ..., x_k, y_k\}$ est un indépendant 2-partitionné de G, avec $x_i, y_i \in V_i$ pour tout i. Soit $T = B[(x_1, y_1), ..., (x_k, y_k)]$. Alors T est un arbre compatible de $\mathcal{T}$.

En effet, clairement $T \trianglelefteq P$ et $T \trianglelefteq P'$. De plus, soit $e = \{x, y\} \in E$, avec $x \in V_i, y \in V_j$, montrons que $T \trianglelefteq Q_e$. Comme on ne peut pas avoir $x, y \in I$, on est dans l'un des cas suivants :

- si $x \in I, y \notin I$: $Q_e|I$ est obtenu à partir de T en contractant les arêtes joignant i à j, sauf e_j.
- si $x \notin I, y \in I$: $Q_e|I$ est obtenu à partir de T en contractant les arêtes joignant i à j, sauf e_i.
- si $x \notin I, y \notin I$: $Q_e|I$ est obtenu à partir de T en contractant les arêtes joignant i à j, sauf e_i et e_j.

Cela prouve que T raffine $Q_e|I$, et donc $T \trianglelefteq Q_e$.

($\Leftarrow$) : supposons que T est un arbre compatible de $\mathcal{T}$ de taille $2k$. Faisons l'observation suivante :

Pour tout i, $|L(T) \cap V_i| \leq 2$. En effet, si $L(T) \cap V_i$ contenait trois éléments, ils induiraient un h-conflit entre $\{P, P'\}$, et donc entre $\mathcal{T}$.

Donc, puisque $|L(T)| = 2k$, on doit avoir $|L(T) \cap V_i| = 2$ pour tout i. Soit $X_i = L(T) \cap V_i$, et soit $I = X_1 \cup ... \cup X_k$, alors I est un indépendant 2-partitionné de G. En effet, on a $|X_i| = 2$ et $X_i \subseteq V_i$ pour tout i. De plus, on ne peut pas avoir une arête $e = \{u, v\} \in E$ telle que $e \subseteq I$: si c'était le cas, on aurait $u \in V_i, v \in V_j$ avec $i \neq j$; alors, en considérant l'élément $w \in X_i - \{u\}$, on obtiendrait $uv|w \in rt(P)$ et $uv|w \in rt(Q_e)$, contradiction. □

Le principe de la réduction est illustré ci-dessous :

4.3 Bibliographie

[1] Alekhnovich (M.), Buss (S.), Moran (S.) et Pitassi (T.). – Minimum Propositional Proof Length is NP-Hard to Linearly Approximate. *Journal of Symbolic Logic*, vol. 66, n° 1, 2001, pp. 171–191.

[2] Amaldi (E.) et Kann (V.). – The complexity and approximability of finding maximum feasible subsystems of linear relations. *Theoretical Computer Science*, vol. 147, 1995, pp. 181–210.

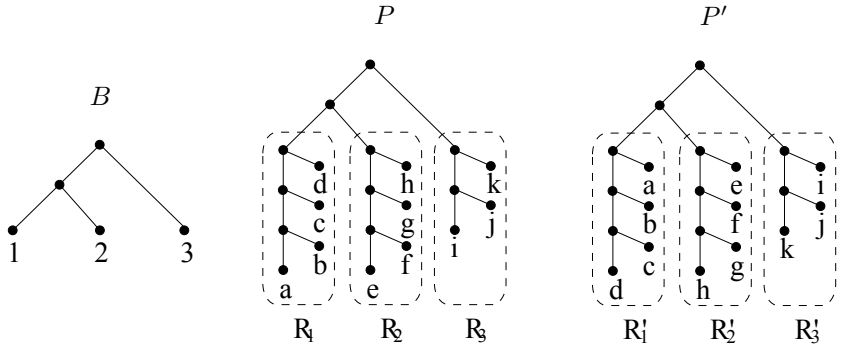


FIG. 4.14 – Etant donnée une instance de PIS2 avec $k = 3$, $V_1 = \{a, b, c, d\}, V_2 = \{e, f, g, h\}, V_3 = \{i, j, k\}$ et comportant les arêtes gk et bf, les arbres B, P, P' construits par la réduction.

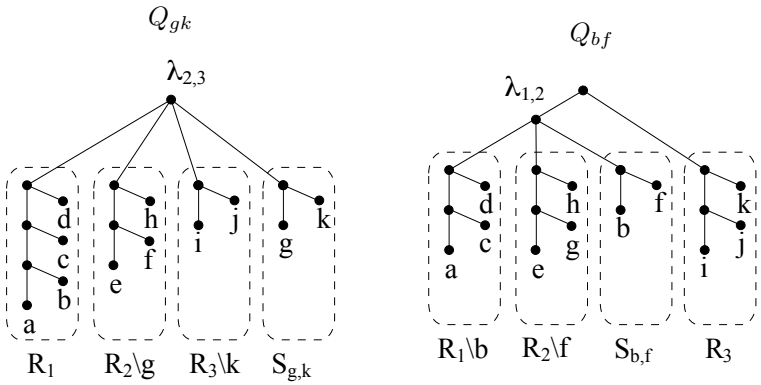


FIG. 4.15 – Les arbres Q_{gk} et Q_{bf} construits par la réduction.

[3] Amaldi (E.) et Kann (V.). – On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems. *Theoretical Computer Science*, vol. 209, 1998, pp. 237–260.

[4] Amir (A.) et Keselman (D.). – Maximum agreement subtree in a set of evolutionary trees : metrics and efficient algorithm. *SIAM Journal on Computing*, vol. 26, n° 6, 1997, pp. 1656–1669.

[5] Arora (S.), Babai (L.), Stern (J.) et Sweedyk (Z.). – The hardness of approximate optima in lattices, codes, and systems of linear equations. *Journal of Computer and System Sciences*, vol. 54, 1997, pp. 317–31.

[6] Berry (V.), Guillemot (S.), Nicolas (F.) et Paul (C.). – Linear time 3-approximation for the MAST problem. 2007.

[7] Berry (V.), Guillemot (S.), Nicolas (F.) et Paul (C.). – On the approximability of the MAST and MCT problems. 2007. – Accepté pour publication dans Discrete Applied Mathematics.

[8] Berry (V.) et Nicolas (F.). – Improved Parameterized Complexity of the Maximum Agreement Subtree and Maximum Compatible Tree Problems. *IEEE/ACM Transactions In Computational Biology and Bioinformatics*, vol. 3, n° 3, 2006, pp. 289–302.

[9] Berry (V.) et Nicolas (F.). – Maximum agreement and compatible supertrees. *Journal of Discrete Algorithms*, vol. 5, n° 3, 2007, pp. 564–591.

[10] Bonizonni (P.), Vedova (G. Della) et Mauri (G.). – Approximating the maximum isomorphic agreement subtree is hard. *International Journal of Foundations of Computer Science*, vol. 11, n° 4, 2000, pp. 579–590.

[11] Bryant (D.). – *Building trees, hunting for trees and comparing trees : theory and method in phylogenetic analysis*. – Thèse de doctorat, University of Canterbury, Department of Mathemathics, 1997.

[12] Bryant (D.), Fellows (M. R.), Raman (V.) et Stege (U.). – On the parametrized complexity of MAST and 3-Hitting Set. – 1998. manuscript.

[13] Chlebík (M.) et Chlebíková (J.). – Approximation Hardness of Dominating Set Problems. *In : Proceedings of ESA 2004*. pp. 192–203. – Springer Verlag.

[14] Cole (R.), Farach-Colton (M.), Hariharan (R.), Przytycka (T.M.) et Thorup (M.). – An $O(n \log n)$ Algorithm for the Maximum Agreement Subtree Problem for Binary Trees. *SIAM Journal on Computing*, vol. 30, n° 5, 2000, pp. 1385–1404.

[15] Coudert (D.), Datta (P.), Perennes (S.), Rivano (H.) et Voge (M-E.). – Shared Risk Resource Group : Complexity and Approximability Issues. *Parallel Processing Letters*, vol. 17, n° 2, 2007, pp. 169–184.

[16] et al. (A. Agra). – A spanning star forest model for the diversity problem in automobile industry. *In : ECCO XVIII, Minsk, Belarus, May 26-28*.

[17] Farach (M.), Przytycka (T.M.) et Thorup (M.). – On the Agreement of Many Trees. *Information Processing Letters*, vol. 55, n° 6, 1995, pp. 297–301.

[18] Farach (M.) et Thorup (M.). – Fast Comparison of Evolutionary Trees. *Information and Computation*, vol. 123, n° 1, 1995, pp. 29–37.

[19] Ferneyhough (S.), Haas (R.), Hanson (D.) et MacGillivray (G.). – Star forests, dominating sets and Ramsey-type problems. *Discrete Mathematics*, vol. 245, 2002, pp. 255–262.

[20] Finden (C.R.) et Gordon (A.D.). – Obtaining common pruned trees. *Journal of Classification*, vol. 2, 1985, pp. 255–276.

[21] Gabow (H.N.) et Tarjan (R.E.). – Faster Scaling Algorithms for Network problems. *SIAM Journal on Computing*, vol. 18, n° 5, 1989, pp. 1013–1036.

[22] Ganapathy (G.) et Warnow (T.J.). – Approximating the complement of the maximum compatible subset of leaves of k trees. *In : Proceedings of APPROX 2002*, pp. 122–134.

[23] Ganapathysaravanabavan (G.) et Warnow (T.). – Finding a maximum compatible tree for a bounded number of trees with bounded degree is solvable in polynomial time. *In : Proceedings of WABI 2001*, pp. 156–163.

[24] Guillemot (S.). – Simpler algorithms for approximating the MAST and MCT problems. 2008. – En préparation.

[25] Guillemot (S.) et Nicolas (F.). – Parameterized complexity of the MAST and MCT problems. 2008. – En préparation.

[26] Hamel (A.M.) et Steel (M.A.). – Finding a maximum compatible tree is np-hard for sequences and trees. *Applied Mathematics Letters*, vol. 9, n° 2, 1996, pp. 55–59.

[27] Hassin (R.), Monnot (J.) et Segev (D.). – Approximation Algorithms and Hardness Results for Labeled Connectivity Problems. *In : Proceedings of MFCS 2006*, pp. 480–491.

[28] Hastad (J.). – Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, vol. 182, 1999, pp. 105–142.

[29] Hein (J.), Jiang (T.), Wang (L.) et Zhang (K.). – On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, vol. 71, n° 1–3, 1996, pp. 153–169.

[30] Jiang (T.) et Li (M.). – On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, vol. 24, n° 5, 1995, pp. 1122–1139.

[31] Kann (V.). – Maximum bounded 3-dimensional matching is MAX SNP-complete. *Information Processing Letters*, vol. 37, 1991, pp. 27–35.

[32] Kao (M.-Y.), Lam (T.W.), Sung (W.-K.) et Ting (H.-F.). – A decomposition theorem for maximum weight bipartite matchings with applications to evolutionary trees. *In : Proceedings of ESA 1999*, pp. 438–449.

[33] Kao (M.-Y.), Lam (T.W.), Sung (W.-K.) et Ting (H.-F.). – An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. *Journal of Discrete Algorithms*, vol. 40, n° 2, 2001, pp. 212–233.

[34] Kubicka (E.), Kubicki (G.) et McMorris (F.R.). – An algorithm to find agreement subtrees. *Journal of Classification*, vol. 12, 1995, pp. 91–99.

[35] Nguyen (C.T.), Shen (J.), Hou (M.), Sheng (L.), Miller (W.) et Zhang (L.). – Approximating the Spanning Star Forest Problem and Its Applications to Genomic Sequence Alignment. *In : Proceedings of SODA 2007*, pp. 645–654.

[36] Przytycka (T.). – Sparse dynamic programming for maximum agreement subtree problem. *In : Mathematical Hierarchies and Biology*, pp. 249–264.

[37] Steel (M.A.) et Warnow (T.J.). – Kaikoura tree theorems : Computing the maximum agreement subtree. *Information Processing Letters*, vol. 48, n° 2, 1993, pp. 77–82.

[38] Zuckerman (D.). – Linear degree extractors and the inapproximability of max clique and chromatic number. *In : Proceedings of STOC 2006*, pp. 681–690.

Chapitre 5

Les problèmes Slcs et Smast

On considère dans ce chapitre des collections d'objets étiquetés (séquences, arbres) ayant des ensembles d'étiquettes distincts mais qui se chevauchent partiellement. Une collection est dite compatible si les objets peuvent être combinés sans conflit en un objet unique. En présence d'une collection non compatible, on cherche à la rendre compatible par suppression d'un plus petit nombre d'étiquettes. On formule ainsi deux problèmes, Slcs portant sur les séquences, et Smast portant sur les arbres. Ces problèmes rencontrent des applications en bioinformatique. Ainsi, le problème Slcs permet de définir une mesure de similarité entre ordres de gènes ; le fait que chaque séquence ne comporte qu'un sous-ensemble des étiquettes permet de prendre en compte les pertes de gènes. Pour sa part, le problème Smast a une application en phylogénie pour la construction de superarbres, avec deux motivations principales. La première motivation consiste à déduire des relations entre taxons qui ne sont présentes dans aucun des arbres sources. La seconde motivation concerne le cas où l'ensemble des taxons est trop large pour appliquer une méthode coûteuse, telle que le maximum de vraisemblance ou le maximum de parcimonie : on adopte alors une approche diviser pour régner où l'on infère des arbres de petite taille par une méthode coûteuse en temps de calcul, pour ensuite les combiner en un arbre unique par une méthode de superarbre.

5.1 Problème Slcs

Cette section est consacrée au problème Slcs, et est organisée selon le plan suivant. La section 5.1.1 contient des définitions préliminaires relatives au problème Slcs. En section 5.1.2, on présente des résultats algorithmiques pour Slcs. En section 5.1.3, on présente des résultats de difficulté paramétrique pour Slcs : on montre en particulier que le problème Slcs paramétré par le nombre de séquences k est complet pour la classe de complexité WNL. En section 5.1.4, on présente les implications de ces résultats pour la complexité de Lcs et d'autres problèmes. On formule quelques remarques conclusives en section

5.1.5.

5.1.1 Préliminaires

Définitions

On introduit des définitions relatives aux *séquences*. Après [9], on appelle *p-séquence* sur L un mot sur L où chaque lettre apparaît au plus une fois. Les lettres sont appelées *étiquettes*, et l'ensemble des lettres apparaissant dans s est appelé l'*ensemble d'étiquettes* de s, noté $L(s)$. La $i$ème lettre de s est notée $s[i]$, et la longueur de s est notée $|s|$. Etant donnés deux éléments $x, y \in L(s)$, $x <_s y$ signifie que x précède y dans s. Etant donné $x \in L(s)$, on note $pred_s(x)$, resp. $succ_s(x)$, le prédécesseur, resp. successeur, de x dans s, ou $\perp$ s'il n'existe pas de telle étiquette. Etant donné un ensemble $L' \subseteq L$, la *restriction* de s à L', notée $s|L'$, est la sous-séquence de s formée des étiquettes de L'. Etant données deux p-séquences s, s', s *s'accorde avec* s' si et seulement si $s|L(s') = s'|L(s)$. De façon équivalente, elles doivent vérifier : pour chaque $x, y \in L(s) \cap L(s')$, $x <_s y$ si et seulement si $x <_{s'} y$.

Exemple 5.1. *Considérons la p-séquence* $s = edbcfa$, *alors* $L(s) = \{a, b, c, d, e, f\}$, *et la restriction de* s *à* $\{a, b, c, d\}$ *est* $s|\{a, b, c, d\} = dbca$. *De plus, si* $s' = hdbicag$, *alors* s *s'accorde avec* s' *puisqu'elles ont la même restriction à leur ensemble d'étiquettes communes* $\{a, b, c, d\}$.

On introduit maintenant des définitions relatives aux *collections* et à la *compatibilité*. Etant donné un ensemble L de n éléments, une *collection* sur L est une famille $\mathcal{C} = \{s_1, ..., s_k\}$ de p-séquences sur L. On supposera que pour les collections considérées dans la suite, chaque étiquette de L apparaît dans au moins une p-séquence, i.e. $L = \cup_{i \in [k]} L(s_i)$. Etant donné un ensemble $L' \subseteq L$, la *restriction* de $\mathcal{C}$ à L' est la collection $\mathcal{C}|L'$ sur L' définie par : $\mathcal{C}|L' = \{s_1|L', ..., s_k|L'\}$. Une *p-séquence compatible* pour $\mathcal{C}$ est une p-séquence s sur L telle que pour tout $i \in [k]$, s s'accorde avec s_i. Une *p-séquence compatible totale* pour $\mathcal{C}$ est une p-séquence compatible pour $\mathcal{C}$ dont l'ensemble d'étiquettes est L. On dit que $\mathcal{C}$ est *compatible* si et seulement si elle admet une p-séquence compatible totale. Un *conflit* dans $\mathcal{C}$ est un ensemble $L' \subseteq L$ tel que $\mathcal{C}|L'$ est non compatible.

Exemple 5.2. *Considérons la collection* $\mathcal{C} = \{abce, aef, fbd\}$ *sur* $L = \{a, b, c, d, e, f\}$. *Alors* $s = acefd$ *est une p-séquence compatible pour* $\mathcal{C}$. *Cependant,* $\mathcal{C}$ *est non compatible, car elle contient par exemple le conflit* $\{b, e, f\}$.

On introduit maintenant des définitions relatives au problème SLCS. Le problème SLCS demande : étant donnée une collection $\mathcal{C}$ sur L, trouver un ensemble $L' \subseteq L$ de cardinal maximum tel que $\mathcal{C}|L'$ soit compatible ; observons que c'est équivalent à chercher une p-séquence compatible pour $\mathcal{C}$ de longueur maximum. Le problème complémentaire CSLCS demande : étant donnée une collection $\mathcal{C}$ sur L, trouver un ensemble $L' \subseteq L$ de cardinalité minimum tel que $\mathcal{C}|(L \backslash L')$ soit compatible. On note SLCS($\mathcal{C}$), resp. CSLCS($\mathcal{C}$), la taille d'une solution optimale pour SLCS, resp. CSLCS.

Exemple 5.3. *Pour la collection définie ci-dessus, on a* $\text{SLCS}(\mathcal{C}) = 5$ *et* $\text{CSLCS}(\mathcal{C}) = 1$.

Les définitions suivantes seront utiles en Section 5.1.2. Si s est une p-séquence sur L, on pose $L^\top(s) = L(s) \cup \{\top\}$, et on étend la relation $<_s$ à $L^\top(s)$ en posant $x <_s \top$ pour tout $x \in L(s)$. Considérons une collection $\mathcal{C} = \{s_1, ..., s_k\}$ sur L. Une *position* dans $\mathcal{C}$ est un tuple $\pi = (x_1, ..., x_k)$ tel que $x_i \in L^\top(s_i)$ pour tout $i \in [k]$. La *position initiale* est $\pi_\perp = (s_1[1], ..., s_k[1])$. La *position finale* est $\pi_\top = (\top, ..., \top)$. L'*ensemble d'indices* d'une position π est $I(\pi) = \{i \in [k] : \pi[i] \neq \top\}$. L'*ensemble d'étiquettes* d'une position π est $S(\pi) = \{\pi[i] : i \in I(\pi)\}$.

On définit une relation d'ordre $\leq_\mathcal{C}$ sur les positions dans $\mathcal{C}$ comme suit : étant données π, π' positions dans $\mathcal{C}$, $\pi \leq_\mathcal{C} \pi'$ si et seulement si $\pi[i] \leq_{s_i} \pi'[i]$ pour tout $i \in [k]$. On définit également deux notions de *positions successeurs*. Soit π une position dans $\mathcal{C}$. Etant donné $i \in I(\pi)$, on définit $succ_i(\pi)$ comme la position π' telle que (i) $\pi'[i] = succ_{s_i}(\pi[i])$, (ii) $\pi'[j] = \pi[j]$ pour tout $j \neq i$. Etant donné $a \in S(\pi)$, on définit $succ_a(\pi)$ comme la position π' telle que (i) $\pi'[i] = succ_{s_i}(\pi[i])$ si $\pi[i] = a$, (ii) $\pi'[i] = \pi[i]$ sinon.

Exemple 5.4. *Pour la collection définie ci-dessus, considérons les positions* $\pi = \pi_\perp = (a, a, f)$ *et* $\pi' = (b, f, f)$. *Alors* $\pi \leq_\mathcal{C} \pi'$. *De plus, on a* $succ_1(\pi') = (c, f, f)$ *et* $succ_f(\pi') = (b, \perp, b)$.

Résultats

Nous obtenons les résultats suivants concernant l'approximabilité et la complexité paramétrique de SLCS. Nous montrons d'une part que CSLCS admet un algorithme de k-approximation en temps $O(kn)$, et un algorithme FPT en temps $O(k^p \times kn)$. Nous montrons d'autre part que SLCS admet un algorithme de 2^k-approximation en temps $O(kn^2)$, et est soluble en temps $O(kn^k)$ par programmation dynamique. Nous obtenons par ailleurs des résultats de difficulté paramétrique pour différents paramétrages du problème ; notamment, le problème paramétré par le nombre de séquences k est prouvé complet pour la classe WNL introduite au Chapitre 3.

Ce dernier résultat s'avère avoir des implications importantes. Il nous permet en effet d'établir la WNL-difficulté du problème LCS paramétré par le nombre de séquences, ainsi que d'autres problèmes auparavant classés comme « difficiles pour la W-hiérarchie » [6, 2, 3, 18]. Plus généralement, il fournit un cadre pour étudier une large classe de problèmes paramétrés : il s'agit de problèmes solubles par programmation dynamique k-dimensionnelle, appelés « problèmes de largeur bornée » dans [5]. [5, 6] ont proposé la notion de « difficulté pour la W-hiérarchie » pour caractériser la complexité de ces problèmes, que nous proposons de remplacer par la notion de WNL-difficulté.

Les résultats relatifs à la complexité paramétrique de SLCS sont résumés dans le tableau suivant :

On rappelle que q est une borne sur le nombre d'étiquettes à conserver, et que p est une borne sur le nombre d'étiquettes à supprimer, suivant la convention formulée en section 2.3.

Paramètres	Complexité de SLCS	Source
q	W[1]-complet	Prop 5.12
q, k	W[1]-complet	Prop 5.13
p	FPT par réduction à DFVS	Prop 5.6
k, p	FPT par un algorithme en $O(k^p \times kn)$	Prop 5.8
k	WNL-complet soluble en temps $O(kn^k)$	Prop 5.15 Prop 5.11

FIG. 5.1 – Complexité paramétrique de SLCS

5.1.2 Algorithmes

Algorithmes pour CSlcs

On décrit en premier lieu des algorithmes FPT et des algorithmes d'approximation pour le problème CSLCS.

La proposition suivante met en relation SLCS avec des problèmes d'optimisation sur les graphes. Etant donnée une collection $\mathcal{C} = \{s_1, ..., s_k\}$ sur L, on définit le digraphe $G(\mathcal{C})$ comme suit : (i) son ensemble de sommets est L, (ii) il contient un arc (x, y) si et seulement si il existe $i \in [k]$ tel que $x, y \in L(s_i)$ et $x <_{s_i} y$.

Le problème MAXIMUM ACYCLIC SUBGRAPH (MAXASG) consiste, étant donné un digraphe $G = (V, A)$, à trouver un ensemble $V' \subseteq V$ de cardinal maximum tel que $G[V']$ soit acyclique. Le problème MINIMUM DIRECTED FEEDBACK VERTEX SET (MINDFVS) consiste, étant donné un digraphe $G = (V, A)$, à trouver un ensemble $V' \subseteq V$ de cardinal minimum tel que $G[V \backslash V']$ soit acyclique.

Proposition 5.5. *(i) $\mathcal{C}$ est compatible si et seulement si $G(\mathcal{C})$ est acyclique ; (ii)* $\text{SLCS}(\mathcal{C}) = \text{MAXASG}(G(\mathcal{C}))$ *et* $\text{CSLCS}(\mathcal{C}) = \text{MINDFVS}(G(\mathcal{C}))$.

Démonstration. Le point (i) résulte du fait qu'une p-séquence compatible totale de $\mathcal{C}$ correspond à un tri topologique de $G(\mathcal{C})$. Le point (ii) est une conséquence immédiate du point (i). □

On a donc une réduction de SLCS au problème DFVS. Les résultats algorithmiques connus pour DFVS [16, 8, 7] impliquent les résultats suivants pour CSLCS.

Proposition 5.6. *(i)* CSLCS *peut être résolu en temps* $2^{O(p \log p)} kn^4$*, (ii)* CSLCS *peut être approximé à un facteur* $O(\log n \log \log n)$ *en temps polynomial.*

On donne à présent des algorithmes FPT et des algorithmes d'approximation, plus efficaces que ceux de la proposition précédente lorsque k est borné. Ils reposent sur le résultat suivant :

Proposition 5.7. *Etant donnée une collection $\mathcal{C}$, en temps $O(kn)$ on peut décider si $\mathcal{C}$ est compatible, renvoyer une p-séquence compatible en cas de résultat positif, ou renvoyer un conflit entre $\mathcal{C}$ de cardinal $\leq k$ en cas de résultat négatif.*

Démonstration. Avant de décrire l'algorithme, introduisons les notations suivantes. Supposons que $\mathcal{C} = \{s_1, ..., s_k\}$ est une collection sur L. Etant donnée π position dans $\mathcal{C}$, et $x \in L$, notons $n_\pi(x)$ le nombre d'indices $i \in [k]$ tels que $(x \in L(s_i)$ et $\pi[i] <_{s_i} x)$.

L'algorithme maintient une position π dans $\mathcal{C}$, et pour chaque $x \in L$ un compteur n_x égal à $n_\pi(x)$. En outre, il maintient une p-séquence s, préfixe d'une hypothétique p-séquence compatible pour $\mathcal{C}$. Il démarre avec $s = \epsilon, \pi = \pi_\perp$, et chaque n_x initialisé au nombre d'indices $i \in [k]$ tels que x soit présent dans s_i et différent de la première étiquette. Tant que $\pi \neq \pi_\top$, l'algorithme recherche $x \in S(\pi)$ tel que $n_x = 0$. S'il n'existe pas de tel élément, l'algorithme répond négativement en renvoyant $S(\pi)$. Sinon, l'algorithme choisit un tel élément x, et fait les mises à jour suivantes :

(i) pour chaque $i \in [k]$ tel que $\pi[i] = x$, soit $y = succ_{s_i}(x)$, si $y \neq \perp$ alors $n_y \leftarrow n_y - 1$;
(ii) faire $\pi \leftarrow succ_x(\pi)$ et $s \leftarrow sx$.

Lorsque $\pi = \pi_\top$ est atteinte, l'algorithme répond positivement et renvoie s.

La correction de l'algorithme résulte du fait que : (i) s'il répond négativement, alors l'ensemble $S(\pi)$ renvoyé est un conflit ; (ii) s'il répond positivement, alors la p-séquence s renvoyée est une p-séquence compatible totale pour $\mathcal{C}$. Le temps d'exécution de l'algorithme est $O(kn)$, puisque l'initialisation s'effectue en temps $O(kn)$, et puisqu'à chaque itération trouver $x \in S(\pi)$ tel que $n_x = 0$ et mettre à jour les compteurs prend un temps $O(k)$. □

Comme conséquence de la Proposition 5.7, on obtient :

Proposition 5.8. *(i)* CSLCS *peut être résolu en temps $O(k^p \times kn)$, (ii)* CSLCS *peut être approximé à un facteur k en temps $O(kn)$.*

Démonstration. Le point (i) résulte de la Proposition 5.7 en utilisant la recherche bornée. Le point (ii) s'obtient en adaptant l'algorithme de la Proposition 5.7 de manière à éliminer à la volée les conflits identifiés par l'algorithme. La modification à apporter à l'algorithme est la suivante. L'algorithme maintient un ensemble P d'étiquettes impliquées dans des conflits disjoints. Il démarre avec $P = \emptyset$. Si lors de l'examen d'une position π l'algorithme ne trouve pas $x \in S(\pi)$ tel que $n_x = 0$, il fait alors les mises à jour suivantes :

(i) il met à jour P en ajoutant l'ensemble des étiquettes de π : $P \leftarrow P \cup S(\pi)$;
(ii) il met à jour π et les compteurs n_x : pour chaque $i \in I(\pi)$, soit $y = succ_{s_i}(\pi[i])$, faire $n_y \leftarrow n_y - 1$ (si $y \neq \perp$), et faire $\pi[i] \leftarrow y$.

Lorsque $\pi = \pi_\top$ est atteinte, l'algorithme renvoie P. Il est clair que l'algorithme produit bien une k-approximation de CSLCS, en temps $O(kn)$. □

Algorithmes pour Slcs

On décrit maintenant des algorithmes polynomiaux et un algorithme d'approximation pour le problème SLCS.

On présente d'abord un cas où le problème SLCS peut se résoudre en temps polynomial. On dit qu'une collection $\mathcal{C} = \{s_1, ..., s_k\}$ sur L est *complète* si et seulement si chaque étiquette de L apparaît dans chaque p-séquence, *précomplète* si et seulement si chaque étiquette de L apparaît dans 1 ou k p-séquences. On montre que SLCS peut être résolu de manière efficace pour de telles collections.

Proposition 5.9. SLCS *peut être résolu en temps* $O(kn^2)$ *si l'instance est une collection complète, ou une collection précomplète.*

Démonstration. Si $\mathcal{C} = \{s_1, ..., s_k\}$ est une collection complète sur L, considérons le digraphe G dont l'ensemble de sommets est L, et qui contient un arc (x, y) si et seulement si $x <_{s_i} y$ pour tout $i \in [k]$. G est acyclique, et une p-séquence compatible pour $\mathcal{C}$ correspond à un chemin de G. Par conséquent, SLCS($\mathcal{C}$) peut être calculé comme la longueur d'un plus long chemin de G, et donc en temps $O(kn^2)$.

Si $\mathcal{C}$ est une collection précomplète sur L, soit L' l'ensemble des étiquettes qui apparaissent dans chaque p-séquence, et soit L'' les autres étiquettes. On a alors SLCS($\mathcal{C}$) = SLCS($\mathcal{C}|L'$) + $|L''|$: en effet, si s est une p-séquence compatible pour $\mathcal{C}|L'$, alors pour chaque $i \in [k]$ on peut insérer dans s les étiquettes de $L'' \cap L(s_i)$, en respectant leur ordre relatif dans s_i, ainsi que leur ordre par rapport aux éléments de L' qui apparaissent dans s et s_i ; on obtient alors une p-séquence compatible pour $\mathcal{C}$ de longueur $|s| + |L''|$. Finalement, comme $\mathcal{C}|L'$ est complète, SLCS($\mathcal{C}|L'$) peut être calculé en temps $O(kn^2)$, donc SLCS($\mathcal{C}$) peut être calculé avec la même complexité en temps. □

Le résultat précédent fournit un algorithme d'approximation pour le problème SLCS général :

Proposition 5.10. SLCS *peut être approximé à un facteur* 2^k *en temps* $O(kn^2)$.

Démonstration. On utilise la technique d'approximation par partitionnement [11]. Soit $\mathcal{C} = \{s_1, ..., s_k\}$ une collection sur L. Pour chaque $X \subseteq [k]$ soit L_X l'ensemble des étiquettes de L qui apparaissent exactement dans les p-séquences s_i pour $i \in X$. Clairement, les ensembles L_X ($X \subseteq [k]$) forment une partition de L. Définissons $n_X = |L_X|$ et $\mathcal{C}_X = \mathcal{C}|L_X$. Alors $\mathcal{C}_X$ est une collection complète, et par la Proposition 5.9 chaque valeur SLCS($\mathcal{C}_X$) peut être calculé en temps $O(|X|n_X^2)$. Considérons l'algorithme qui calcule SLCS($\mathcal{C}_X$) pour chaque $X \subseteq [k]$, et renvoie le maximum des valeurs calculées. Il est facile de voir que cet algorithme calcule une 2^k-approximation, en temps $\sum_{X \subseteq [k]} O(|X|n_X^2) = O(kn^2)$. □

On montre finalement que SLCS peut être résolu en temps polynomial pour k fixé, en utilisant la programmation dynamique :

Proposition 5.11. SLCS *peut être résolu en temps* $O(kn^k)$.

Démonstration. Soit π position dans $\mathcal{C}$. Soit une étiquette $x \in L$, on dit que x est *autorisée* en π si et seulement si pour tout $i \in [k]$ tel que $x \in L(s_i)$, on a : $\pi[i] \leq_{s_i} x$. On note $A(\pi)$ l'ensemble des étiquettes autorisées en π. Notons $\textsc{Slcs}(\pi)$ la taille d'une plus longue p-séquence compatible de $\mathcal{C}|A(\pi)$.

Notons $F(\pi)$ l'ensemble des éléments *pleins* de $S(\pi)$, i.e. l'ensemble des éléments $a \in S(\pi)$ tels que pour tout $i \in [k]$, $a \in L(s_i) \Rightarrow \pi[i] = a$. Etant données deux positions π, π' et $a \in S(\pi)$, $\pi \rightarrow_a \pi'$ si et seulement si $a \in F(\pi)$ et $\pi' \geq_{\mathcal{C}} succ_a(\pi)$. Une π-*chaîne* est une chaîne $\pi_1 \rightarrow_{a_1} \pi_2 \rightarrow_{a_2} ... \rightarrow_{a_m} \pi_{m+1}$, avec $\pi_1 \geq_{\mathcal{C}} \pi$ et $\pi_{m+1} = \pi_\top$. La *longueur* de la π-chaîne est m.

On montre :

Fait. $\textsc{Slcs}(\pi)$ est la longueur d'une plus longue π-chaîne.
Preuve. Considérons une π-chaîne $\pi_1 \rightarrow_{a_1} ... \rightarrow_{a_m} \pi_{m+1}$ de longueur m. Observons d'abord que chaque a_j est dans $A(\pi)$: supposons que $a_j \in L(s_i)$, on a alors $a_j \in F(\pi_j)$, ce qui implique que $a_j = \pi_j[i]$ et donc $a_j \geq_{s_i} \pi[i]$ (puisque $\pi_j \geq_{\mathcal{C}} \pi$). Observons ensuite que si $j < j'$ et $a_j, a_{j'} \in L(s_i)$ alors $a_j <_{s_i} a_{j'}$: en effet, on a alors $a_j = \pi_j[i]$ (puisque $a_j \in F(\pi_j)$) et $a_{j'} = \pi_{j'}[i]$ (puisque $a_{j'} \in F(\pi_{j'})$) ; comme $\pi_j \leq_{\mathcal{C}} \pi_{j'}$, on a donc $\pi_j[i] \leq_{s_i} \pi_{j'}[i]$, et donc $a_j <_{s_i} a_{j'}$. Il résulte que $s = a_1...a_m$ est une p-séquence compatible pour $\mathcal{C}|A(\pi)$.

Réciproquement, si $s = a_1...a_m$ est une p-séquence compatible pour $\mathcal{C}|A(\pi)$ de longueur m, alors considérons une exécution de l'algorithme de la Proposition 5.7 sur la collection $\mathcal{C}|L(s)$. Comme $\mathcal{C}|L(s)$ est compatible, l'algorithme répond positivement, et examine une suite de positions $\pi_1, ..., \pi_{m+1}$ dans $\mathcal{C}|L(s)$ telles que : pour tout $i \leq m$, $a_i \in F(\pi_i)$ et $\pi_{i+1} = succ_{a_i}(\pi_i)$ (dans $\mathcal{C}|L(s)$). En considérant les π_j comme des positions dans $\mathcal{C}$, on obtient que : (i) $\pi_1 \geq_{\mathcal{C}} \pi$, (ii) pour chaque $j \leq m$, $a_j \in F(\pi_j)$ et $\pi_{j+1} \geq_{\mathcal{C}} succ_{a_j}(\pi_j)$. On conclut que $\pi_1 \rightarrow_{a_1} ... \rightarrow_{a_m} \pi_{m+1}$ est une π-chaîne de longueur m. □

Le résultat ci-dessus fournit un algorithme de programmation dynamique pour résoudre Slcs en temps $O(kn^k)$. L'algorithme calcule $\textsc{Slcs}(\pi)$ pour chaque position π, par les relations de récurrence suivantes.

- si $\pi = \pi_\bot$, alors $\textsc{Slcs}(\pi) = 0$;
- si $\pi \neq \pi_\bot$, alors $\textsc{Slcs}(\pi) = \max(\textsc{Slcs}_1(\pi), \textsc{Slcs}_2(\pi))$, où :

$$\textsc{Slcs}_1(\pi) = \max\{\textsc{Slcs}(succ_i(\pi)) : i \in I(\pi)\}$$
$$\textsc{Slcs}_2(\pi) = \max\{1 + \textsc{Slcs}(succ_a(\pi)) : a \in F(\pi)\}$$

A la fin de l'algorithme, $\textsc{Slcs}(\mathcal{C})$ est obtenu comme $\textsc{Slcs}(\pi_\bot)$. Comme le nombre de positions π est $O(n^k)$, et comme chaque valeur $\textsc{Slcs}(\pi)$ peut être calculée en temps $O(k)$ à partir des valeurs $\textsc{Slcs}(\pi')$ ($\pi' >_{\mathcal{C}} \pi$), l'algorithme a la complexité en temps et en espace annoncée. □

5.1.3 Résultats négatifs

On décrit maintenant des résultats de complexité paramétrique pour le problème Slcs. Notre premier résultat concerne la complexité paramétrique du problème par rapport au paramètre q.

Proposition 5.12. SLCS[q] *est* W[1]*-complet.*

Démonstration. Observons que SLCS[q] avec $l = 2$ est équivalent au problème ACYCLIC SUBGRAPH (ASG) : étant donné un digraphe $G = (V, E)$ et un paramètre q, trouver $V' \subseteq V$ de cardinal q tel que $G[V']$ est acyclique. Il suffit donc de montrer que ASG est W[1]-complet.

L'appartenance de ASG à W[1] se montre par réduction à NTMC[q]. Etant donnée $I = (G, q)$ instance de ASG, on construit une MTN qui effectue les opérations suivantes : (i) deviner une suite de sommets $v_1, ..., v_q$, (ii) vérifier que pour tout $(v_i, v_j) \in A$ on a $i < j$. La machine effectue $O(q^2)$ pas, et accepte le mot vide si et seulement si G a un sous-graphe induit acyclique d'ordre q.

La W[1]-difficulté de ASG se montre par réduction depuis CLIQUE. Soit $I = (G, q)$ une instance de CLIQUE avec $G = (V, E)$ graphe non orienté, on construit une instance $I' = (G', q)$ de ASG comme suit. On fixe une énumération $v_1...v_n$ de G. Alors G' est un digraphe dont l'ensemble des sommets est V, et qui contient : (i) un arc (v_i, v_j) pour chaque $i, j \in [n]$, $i < j$; (ii) un arc (v_j, v_i) pour chaque $i, j \in [n]$, $i < j$ et $\{v_i, v_j\} \notin E$. Alors : pour chaque $X \subseteq V$, X est une clique de G si et seulement si $G'[X]$ est acyclique. □

On considère maintenant la complexité paramétrique de SLCS par rapport à la paire de paramètres (q, k) (Proposition 5.13). On utilisera les notations suivantes. Si $s = a_1...a_m$ est une p-séquence, son image miroir est $\tilde{s} = a_m...a_1$. Si $(V, <_V)$ est un ordre total et $\{s_x : x \in V\}$ est une famille de p-séquences ayant des ensembles d'étiquettes disjoints, on note $\prod_{x \in (V, <_V)} s_x$ leur concaténation selon l'ordre $<_V$. Si V est l'intervalle d'entiers $[p, q]$ et $<_V$ est l'ordre naturel sur $\mathbb{N}$, alors $\prod_{x \in (V, <_V)} s_x$ sera abrégé en $\prod_{i=p}^{q} s_i$.

On montre :

Proposition 5.13. SLCS[q, k] *est* W[1]*-complet.*

Démonstration. L'appartenance à W[1] résulte de la Proposition 5.12. La W[1]-difficulté se montre par réduction depuis PCLIQUE. Soit $I = (G, k)$ une instance de PCLIQUE, où $G = (V, E)$ est un graphe k-parti de partition $V_1, ..., V_k$. On construit une instance $I' = (\mathcal{C}, q', k')$ de SLCS[q, k] de la manière suivante.

Construction. On pose $k' = 2k+2, q' = 2k+k(k-1)/2$. On définit la collection $\mathcal{C}$ comme suit :

- Etiquettes : on introduit des étiquettes $a[v], b[v]$ pour chaque $v \in V$, et une étiquette $c[e]$ pour chaque $e \in E$.
- Séquences : on introduit deux p-séquences s, s' et $2k$ p-séquences $t_1, t'_1, ..., t_k, t'_k$. Soit $<_V$ un ordre total arbitraire sur V, et soit $<_E$ un ordre total arbitraire sur E. Pour $i, j \in [k]$ $(i < j)$, notons $E_{i,j}$ l'ensemble des arêtes de G ayant une extrémité dans V_i et l'autre dans V_j. On définit d'abord les p-séquences suivantes :

$$\forall i \in [k], \quad A[i] = \prod_{v \in (V_i, <_V)} a[v], \quad B[i] = \prod_{v \in (V_i, <_V)} b[v],$$

$$\forall i, j \in [k](i < j), \quad C[i, j] = \prod_{e \in (E_{i,j}, <_E)} c[e]$$

On définit alors s, s' comme suit :

$$s = \left(\prod_{i=1}^{k} A[i]\right)\left(\prod_{i=1}^{k}\prod_{j=i+1}^{k} C[i,j]\right)\left(\prod_{i=1}^{k} B[i]\right)$$

$$s' = \left(\prod_{i=1}^{k} \widetilde{A[i]}\right)\left(\prod_{i=1}^{k}\prod_{j=i+1}^{k} \widetilde{C[i,j]}\right)\left(\prod_{i=1}^{k} \widetilde{B[i]}\right)$$

Supposons que $i, j \in [k], i < j$. Etant donné $v \in V_i \cup V_j$, notons $E_{i,j}(v)$ l'ensemble des arêtes de $E_{i,j}$ incidentes à v. Alors pour chaque $i \in [k]$, on définit t_i, t'_i comme suit :

$$t_i = \prod_{v \in (V_i, <_V)} a[v]\left(\prod_{j=i+1}^{k}\prod_{e \in (E_{i,j}(v), <_E)} c[e]\right) b[v]$$

$$t'_i = \prod_{v \in (V_i, >_V)} a[v]\left(\prod_{j=1}^{i-1}\prod_{e \in (E_{i,j}(v), <_E)} c[e]\right) b[v]$$

Correction. La réduction est clairement calculable en temps polynomial. Pour établir que la réduction est correcte, on montre que :

Fait. G a une clique partitionnée si et seulement si $\mathcal{C}$ a une p-séquence compatible de longueur q'.

Preuve. ($\Rightarrow$) : supposons que G a une clique partitionnée $C = \{v_1, ..., v_q\}$, avec $v_i \in V_i$ pour tout i. Considérons la p-séquence suivante :

$$r = \left(\prod_{i=1}^{k} a[v_i]\right)\left(\prod_{i=1}^{k}\prod_{j=i+1}^{k} c[\{v_i, v_j\}]\right)\left(\prod_{i=1}^{k} b[v_i]\right)$$

Observons que r est bien définie, puisque $v_i \in V_i$ pour tout i, et $\{v_i, v_j\} \in E_{i,j}$ pour tout i, j. Alors r est une p-séquence compatible pour $\mathcal{C}$, de longueur q'. En effet, clairement r s'accorde avec s et avec s'. De plus, pour tout $i \in [k]$, r s'accorde avec t_i, puisque :

$$r|L(t_i) = t_i|L(r) = a[v_i]\left(\prod_{j=i+1}^{k} c[\{v_i, v_j\}]\right) b[v_i]$$

De même, r s'accorde avec t'_i.

($\Leftarrow$) : supposons que $\mathcal{C}$ a une p-séquence compatible r de longueur q'. On fait les observations suivantes.

Observation 1. il existe $v_i, v'_i \in V_i$ (pour tout i), $e_{i,j} \in E_{i,j}$ (pour tous i, j) tels que

$$s = \left(\prod_{i=1}^{k} a[v_i]\right)\left(\prod_{i=1}^{k}\prod_{j=i+1}^{k} c[e_{i,j}]\right)\left(\prod_{i=1}^{k} b[v'_i]\right)$$

Preuve. Puisque s s'accorde avec t et t', et puisque t et t' sont complètes, il résulte que s doit être sous-séquence de t et de t'. □

Observation 2. on a : (i) pour tout i, $v_i = v'_i$; (ii) pour tous i, j, $e_{i,j} = \{v_i, v_j\}$.

Preuve. Montrons (i). Supposons par contradiction que $v_i \neq v'_i$. Si $v_i <_V v'_i$, alors $a[v_i] <_s b[v'_i]$, mais $b[v'_i] <_{s'_i} a[v_i]$, impossible. Si $v_i >_V v'_i$, alors $a[v_i] <_s b[v'_i]$, mais $b[v'_i] <_{s_i} a[v_i]$, impossible. On doit donc avoir $v_i = v'_i$.

Montrons (ii). Supposons par contradiction qu'il existe i, j tels que $e_{i,j} = \{x, y\}$ avec $x \in V_i, y \in V_j$ et ($x \neq v_i$ ou $y \neq v_j$).

Supposons d'abord que $x \neq v_i$. Si $x <_V v_i$, alors $c[e_{i,j}] <_{s_i} a[v_i]$ mais $a[v_i] <_s c[e_{i,j}]$. Si $x >_V v_i$, alors $b[v_i] <_{s_i} c[e_{i,j}]$ mais $c[e_{i,j}] <_s b[v_i]$. On obtient dans chaque cas une impossibilité.

Supposons maintenant que $x = v_i$ mais $y \neq v_j$. Si $y <_V v_j$, alors $b[v_j] <_{s'_j} c[e_{i,j}]$ mais $c[e_{i,j}] <_s b[v_j]$. Si $y >_V v_j$, alors $c[e_{i,j}] <_{s'_j} a[v_j]$ mais $a[v_j] <_s c[e_{i,j}]$. Là encore, chaque cas conduit à une impossibilité. □

On conclut que l'ensemble $C = \{v_1, ..., v_k\}$ est une clique partitionnée de G. □

On considère à présent la complexité paramétrique de SLCS par rapport au seul paramètre k (Proposition 5.15). On montre un résultat de complétude pour la classe WNL introduite au Chapitre 2.

Pour établir la WNL-difficulté, on pourrait utiliser le problème GRID LABELING introduit au Chapitre 3. La réduction sera toutefois plus simple si l'on utilise la variante suivante du problème GRID LABELING.

Nom : GRID LABELING-2 (GL-2)
Instance : une $k \times q$-grille $G = (V, A)$, un ensemble S, une partition $\{S_v : v \in V\}$ de S, pour chaque $a = (u, v) \in A$ une fonction $f_a : S_u \cup S_v \to \mathbb{N}^*$.
Question : existe t-il un étiquetage préadmissible de G ? Un étiquetage préadmissible est une assignation à chaque sommet $v \in V$ d'un élément $l_v \in S_v$ tel que pour tout $a = (u, v) \in A$, $f_a(l_u) \leq f_a(l_v)$.

Pour souligner l'analogie de GL-2 avec le problème GRID LABELING (GL), notons que GL peut être reformulé de la façon suivante :

Nom : GRID LABELING (GL)
Instance : une $k \times q$-grille $G = (V, A)$, un ensemble S, une partition $\{S_v : v \in V\}$ de S, pour chaque $a = (u, v) \in A$ une fonction $f_a : S_u \cup S_v \to \mathbb{N}^*$.
Question : existe t-il un étiquetage admissible de G ? Un étiquetage admissible est une assignation à chaque sommet $v \in V$ d'un élément $l_v \in S_v$ tel que pour tout $a = (u, v) \in A$, $f_a(l_u) = f_a(l_v)$.

Pour faire le lien avec la formulation du Chapitre 3, observons que la relation R_e associée à chaque arête peut être encodée par une fonction f_a associée à (l'arc a correspondant à) e, qui associe à chaque élément de $S_u \cup S_v$ le numéro de sa classe d'équivalence dans R_e. Vérifier que deux éléments $x, y \in S_u \cup S_v$ sont équivalents revient alors à vérifier que $f_a(x) = f_a(y)$.

Proposition 5.14. GL-2[k] *est* WNL-*complet.*

Démonstration. L'appartenance à WNL se montre par le même raisonnement que le Point 2 de la Proposition 3.3. La WNL-difficulté de GL-2[k] se montre par réduction depuis GL[k]. Etant donnée une instance I de GL[k] impliquant une $q \times k$-grille G, on construit une instance I' de GL-2[k] impliquant une $2q \times 2k$-grille G', de la façon suivante.

Pour chaque sommet v de G, on introduit le gadget G_v représenté en Figure 5.2 (A). Il consiste en quatre sommets v_1, v_2, v_3, v_4 et quatre arcs $a_1 = (v_1, v_2), a_2 = (v_1, v_3), a_3 = (v_2, v_4), a_4 = (v_3, v_4)$. Etant donné l'ensemble S_v associé à v dans I, on associe à v_1, v_2, v_3, v_4 des copies disjointes $S_1, ..., S_4$ de ces ensembles, où $S_i = \{x_i : x \in S_v\}$. Soit $s = |S_v|$, et soit ϕ une bijection de S_v dans $[s]$. Pour i impair, on pose $f_{a_i}(x_j) = \phi(x)$; pour i pair, on pose $f_{a_i}(x_j) = s + 1 - \phi(x)$. On vérifie que dans un étiquetage préadmissible de G_v, les quatre sommets doivent être étiquetés par les quatre copies d'un même élément.

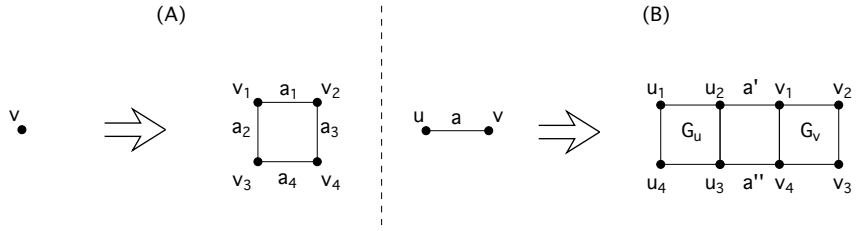


FIG. 5.2 – Figure (A) : le gadget G_v associé à un sommet v de G. Figure (B) : l'interconnexion de deux gadgets G_u, G_v correspondant à un arc horizontal $a = (u, v)$.

On associe à chaque sommet v de G un gadget G_v, et on interconnecte ces gadgets par des arcs de façon à former une grille G'. Pour chaque arc horizontal $a = (u, v)$ de G, on crée deux arcs horizontaux $a' = (u_2, v_1)$ et $a'' = (u_3, v_4)$, comme représenté en Figure 5.2 (B). Si l'image de f_a est $[n_a]$, on pose $f_{a'}(x_j) = f_a(x)$ et $f_{a''}(x_j) = n_a + 1 - f_a(x)$. On procède de même pour les arcs verticaux. Alors G' est une $2q \times 2k$-grille, et on peut vérifier que G a un étiquetage admissible si et seulement si G' a un étiquetage préadmissible. □

Proposition 5.15. SLCS[k] *est* WNL-*complet.*

Démonstration. L'appartenance à WNL se montre par réduction à NTMC[k]. Considérons une instance $I = (\mathcal{C}, q, k)$ de SLCS[k]. On construit une NTM M qui procède comme suit. Les k premières cellules du ruban représentent une position π dans $\mathcal{C}$. M démarre en choisissant de façon non-déterministe une position π dans $\mathcal{C}$. Elle effectue q étapes, chaque étape consistant en $O(k)$ pas. A chaque étape, elle choisit de façon non-déterministe une étiquette $a \in F(\pi)$, une position π' telle que $\pi \rightarrow_a \pi'$, et remplace π par π' sur son ruban. En utilisant le résultat établi dans la preuve de la Proposition 5.11, on peut alors montrer qu'il y a équivalence entre : (i) M accepte le mot vide en $q' = O(kq)$ pas en utilisant un espace $\leq 2k$, (ii) $\text{SLCS}(\mathcal{C}) \geq q$.

La WNL-difficulté se montre par réduction depuis GL-2[k]. Soit I une instance de GL-2[k], consistant en une $q \times k$-grille $G = (V, A)$, en un ensemble S, en une partition $\{S_v : v \in V\}$ de S, et pour chaque $a = (u, v) \in A$ en une fonction $f_a : S_u \cup S_v \rightarrow \mathbb{N}^*$. Pour chaque $a \in A$, supposons que l'image de f_a est incluse dans $[m]$ pour un certain m. Pour chaque $i \in [q], j \in [k]$, soit $v_{i,j}$ le sommet de G en ligne i, colonne j. Pour chaque $1 \leq i \leq q, 1 \leq j \leq k-1$, soit $a_{i,j}$ l'arc horizontal $(v_{i,j}, v_{i,j+1})$. Pour chaque $1 \leq i \leq q-1, 1 \leq j \leq k$, soit $a'_{i,j}$ l'arc vertical $(v_{i,j}, v_{i+1,j})$.

Construction. On construit l'instance suivante $I' = (\mathcal{C}, q', k')$ de SLCS[k]. On pose $q' = 3kq - k - q$ et $k' = 2k + 4$. On définit la collection $\mathcal{C}$ comme suit.

- Etiquettes : on introduit des étiquettes $a[v, x]$ pour chaque $v \in V, x \in S_v$, et des étiquettes $b[a, i]$ pour chaque $a \in A, i \in [m]$.
- Séquences : on définit deux p-séquences s, s', deux p-séquences t, t', et $2k$ p-séquences $u_1, u'_1, ..., u_k, u'_k$. Soit $<_S$ un ordre total sur S. On définit d'abord les p-séquences suivantes :

$$\forall v \in V, A[v] = \prod_{x \in (S_v, <_S)} a[v, x], \forall a \in A, B[a] = \prod_{i=1}^{m} b[a, i]$$

$$\forall a = (u, v) \in A, i \in [m], C[a, i] = \prod_{x \in S_u : f_a(x) = i} a[u, x], C'[a, i] = \prod_{x \in S_v : f_a(x) = i} a[v, x]$$

Les p-séquences de $\mathcal{C}$ sont définies ci-dessous.
Les p-séquences s, s' sont des séquences de contrôle : elles contraignent la forme d'une p-séquence compatible pour $\mathcal{C}$.

$$s_i = \left(\prod_{j=1}^{k-1} A[v_{i,j}] B[a_{i,j}] \right) A[v_{i,k}], s'_i = \left(\prod_{j=1}^{k-1} \widetilde{A[v_{i,j}]} \widetilde{B[a_{i,j}]} \right) \widetilde{A[v_{i,k}]}$$

$$s = \left(\prod_{i=1}^{q-1} s_i \left(\prod_{j=1}^{k} B[a'_{i,j}] \right) \right) s_q, s' = \left(\prod_{i=1}^{q-1} s'_i \left(\prod_{j=1}^{k} \widetilde{B[a'_{i,j}]} \right) \right) s'_q$$

Les p-séquences t, t' sont des séquences de sélection : elles assurent la

satisfaction des contraintes associées aux arcs horizontaux $a_{i,j}$.

$$t = \prod_{i=1}^{q} \prod_{j=1}^{k-1} \prod_{p=1}^{m} C[a_{i,j}, p] b[a_{i,j}, p], \quad t' = \prod_{i=1}^{q} \prod_{j=1}^{k-1} \prod_{p=1}^{m} b[a_{i,j}, p] C'[a_{i,j}, p]$$

Pour chaque $j \in [k]$, les p-séquences u_j, u'_j sont des séquences de sélection, qui assurent la satisfaction des contraintes associées aux arcs verticaux $a'_{i,j}$.

$$u_j = \prod_{i=1}^{q-1} \prod_{p=1}^{m} C[a'_{i,j}, p] b[a'_{i,j}, p], \quad u'_j = \prod_{i=1}^{q-1} \prod_{p=1}^{m} b[a'_{i,j}, p] C'[a'_{i,j}, p]$$

Correction. La réduction est clairement calculable en temps polynomial. Pour établir que la réduction est correcte, on montre :

Fait. G a un étiquetage préadmissible si et seulement si $\mathcal{C}$ a une p-séquence compatible de longueur q'.

Preuve. ($\Rightarrow$) : supposons que G a un étiquetage préadmissible. Introduisons les notations suivantes :

- pour $1 \leq i \leq q, 1 \leq j \leq k$, soit $x_{i,j}$ la valeur associée au sommet $v_{i,j}$;
- pour $1 \leq i \leq q, 1 \leq j < k$, soit $y_{i,j} = f_{a_{i,j}}(x_{i,j})$;
- pour $1 \leq i < q, 1 \leq j \leq k$, soit $z_{i,j} = f_{a'_{i,j}}(x_{i,j})$.

Considérons la p-séquence r définie comme suit :

$$r = \left(\prod_{i=1}^{q-1} r_i \left(\prod_{j=1}^{k} b[a'_{i,j}, z_{i,j}] \right) \right) r_q \tag{5.1}$$

$$\text{où } r_i = \left(\prod_{j=1}^{k-1} a[v_{i,j}, x_{i,j}] b[a_{i,j}, y_{i,j}] \right) a[v_{i,k}, x_{i,k}] \tag{5.2}$$

Alors r est de longueur $kq + k(q-1) + (k-1)q = q'$. On montre que r est une p-séquence compatible pour $\mathcal{C}$. Clairement, r s'accorde avec s et avec s'. De plus :

- r s'accorde avec t : on a en effet

$$r|L(t) = t|L(r) = \prod_{i=1}^{q} \prod_{j=1}^{k-1} a[v_{i,j}, x_{i,j}] b[a_{i,j}, y_{i,j}]$$

Ceci s'obtient en écrivant $t = \prod_{i=1}^{q} \prod_{j=1}^{k-1} t_{i,j}$, et en observant que r contient exactement deux étiquettes de $t_{i,j}$, qui sont $a[v_{i,j}, x_{i,j}]$ et $b[a_{i,j}, y_{i,j}]$. Ces étiquettes apparaissent dans cet ordre dans r, et elles apparaissent dans le même ordre dans $t_{i,j}$, puisque $f_{a_{i,j}}(x_{i,j}) = y_{i,j}$.

– r s'accorde avec t' : on a en effet

$$r|L(t') = t'|L(r) = \prod_{i=1}^{q} \prod_{j=1}^{k-1} b[a_{i,j}, y_{i,j}]a[v_{i,j+1}, x_{i,j+1}]$$

Ceci s'obtient en écrivant $t' = \prod_{i=1}^{q} \prod_{j=1}^{k-1} t'_{i,j}$, et en observant que r contient exactement deux étiquettes de $t'_{i,j}$, qui sont $b[a_{i,j}, y_{i,j}]$ et $a[v_{i,j+1}, x_{i,j+1}]$. Ces étiquettes apparaissent dans cet ordre dans r, et elles apparaissent dans le même ordre dans $t'_{i,j}$, puisque $f_{a_{i,j}}(x_{i,j+1}) \geq f_{a_{i,j}}(x_{i,j}) = y_{i,j}$.

– r s'accorde avec u_j et avec u'_j : par un argument analogue.

($\Leftarrow$) : supposons que $\mathcal{C}$ a une p-séquence compatible r de longueur q'. On fait les observations suivantes.

Observation 1 : r a la forme décrite dans l'Equation (5.1).

Preuve. Comme s, s' sont des p-séquences complètes, la séquence r doit être une sous-séquence de s et de s'. Par définition de s, s', elle peut contenir au plus une lettre dans chaque bloc $A[v_{i,j}], B[a_{i,j}], B[a'_{i,j}]$. Comme ces blocs sont en nombre q', r contient en fait exactement une lettre de chaque bloc, d'où le résultat. □

Dans l'observation suivante, les points 1 et 2 expriment respectivement la satisfaction des contraintes associées aux arcs horizontaux, resp. verticaux.

Observation 2 : On a les propriétés suivantes :

1. pour chaque $1 \leq i \leq q, 1 \leq j < k$, si $a = a_{i,j}, x = x_{i,j}, x' = x_{i,j+1}, y = y_{i,j}$, alors $f_a(x) \leq y \leq f_a(x')$.
2. pour chaque $1 \leq i < q, 1 \leq j \leq k$, si $a = a'_{i,j}, x = x_{i,j}, x' = x_{i+1,j}, y = z_{i,j}$, alors $f_a(x) \leq y \leq f_a(x')$.

Preuve. Considérons le Point 1. Soit $1 \leq i \leq q, 1 \leq j \leq k$. Observons que $a[v_{i,j}, x_{i,j}] <_r b[a_{i,j}, y_{i,j}] <_r a[v_{i,j+1}, x_{i,j+1}]$. Soit $a = a_{i,j}, x = x_{i,j}, x' = x_{i,j+1}, y = y_{i,j}$. Comme r s'accorde avec t, on a $a[v_{i,j}, x_{i,j}] <_t b[a_{i,j}, y_{i,j}]$, ce qui implique $f_a(x) \leq y$. Comme r s'accorde avec t', on a $b[a_{i,j}, y_{i,j}] <_{t'} a[v_{i,j+1}, x_{i,j+1}]$, ce qui implique $f_a(x') \geq y$.

La preuve du Point 2 est similaire, en observant cette fois que $a[v_{i,j}, x_{i,j}] <_r b[a'_{i,j}, z_{i,j}] <_r a[v_{i+1,j}, x_{i+1,j}]$, et en considérant u_j, u'_j. □

On conclut que l'étiquetage de G qui associe la valeur $x_{i,j}$ à chaque sommet $v_{i,j}$ est un étiquetage préadmissible de G. □

5.1.4 Conséquences

Les résultats de difficulté pour SLCS présentés dans la section précédente ont des conséquences pour la complexité du problème LCS, et d'autres problèmes considérés dans [5, 6] et classés comme « durs pour la W-hiérarchie ».

Conséquences pour la complexité de Lcs

Le problème Longest Common Subsequence (Lcs) demande : étant données k séquences $s_1, ..., s_k$ sur un même alphabet Σ de cardinal m, et un entier q, existe t-il une séquence s sur Σ de longueur $\geq q$ qui soit une sous-séquence de chaque s_i ? On donne une réduction paramétrée simple de Slcs à Lcs :

Lemme 5.16. *Il existe une réduction polynomiale préservant les paramètres de* Slcs$[q, k]$ *à* Lcs$[q, k]$.

Démonstration. Etant donnée une instance $I = (\mathcal{C}, q, k)$ de Slcs$[q, k]$, on construit une instance $I' = (\mathcal{C}', q, k)$ de Lcs$[q, k]$. Etant donnée la collection de p-séquences $\mathcal{C} = \{s_1, ..., s_k\}$ sur L, on définit la collection de séquences $\mathcal{C}' = \{s'_1, ..., s'_k\}$, où chaque s'_i est obtenu à partir de s_i en insérant au début, à la fin, et entre chaque étiquettes consécutives de s_i, une énumération des étiquettes de $L \backslash L(s_i)$ répétées q fois. Formellement, soit t_i une énumération arbitraire des étiquettes de $L \backslash L(s_i)$, soit $u_i = t_i^q$, alors $s'_i = u_i s_i[1] u_i s_i[2] ... u_i s_i[|s_i|] u_i$.

La réduction préserve les paramètres, et est clairement polynomiale. Sa correction résulte du fait que : $\mathcal{C}$ a une séquence compatible de longueur $\geq q$ si et seulement si les séquences de $\mathcal{C}'$ ont une sous-séquence commune de longueur $\geq q$. En effet, si $s = a_1 ... a_q$ est une séquence compatible de $\mathcal{C}$, alors s est une sous-séquence de chaque s'_i : puisque s s'accorde avec s_i, les lettres communes à s et s_i apparaissent dans s'_i selon le même ordre, et les lettres restantes de s peuvent être choisies dans les blocs u_i de s'_i. Réciproquement, si les séquences de s' ont une sous-séquence commune $s = a_1 ... a_q$, alors s est une séquence compatible de $\mathcal{C}$: puisque les lettres communes à s et s_i n'apparaissent pas dans les blocs u_i de s'_i, il s'ensuit qu'elle doivent apparaître selon le même ordre dans s et s_i, et donc s s'accorde avec s_i. □

Le Lemme 5.16, en conjonction avec les Propositions 5.13 et 5.15, implique les résultats suivants pour Lcs :

Proposition 5.17. *(i)* Lcs$[q, k]$ *est* W[1]*-complet ; (ii)* Lcs$[k]$ *est* WNL*-complet ; (iii)* Lcs$[k, m]$ *est* WNL*-complet.*

Démonstration. Point (i) : la W[1]-difficulté découle de la Proposition 5.13 et du Lemme 5.16, et l'appartenance à W[1] est établie dans [3].

Point (ii) : le résultat de difficulté découle de la Proposition 5.15 et du Lemme 5.16. L'appartenance à WNL se montre par réduction à Ntmc$[k]$. Soit $I = (\mathcal{C}, q, k)$ une instance de Lcs$[k]$ avec $\mathcal{C} = \{s_1, ..., s_k\}$. On construit une MTN M, dont les k premières cellules du ruban représentent un tuple $(p_1, ..., p_k)$ avec p_i entier entre 0 et $|s_i|$. La machine démarre avec le tuple $(0, ..., 0)$ sur son ruban. A chaque étape $i \leq q$, si le contenu du ruban est $t = (p_1, ..., p_k)$, M remplace de façon non-déterministe t par un tuple $t' = (p'_1, ..., p'_k)$ tel que (i) $s_1[p'_1] = ... = s_k[p'_k]$, (ii) $p'_j > p_j$ pour tout $j \in [k]$. S'il n'existe aucun tuple t' de la sorte, M rejette. Alors M accepte le mot vide en $q' = O(kq)$ pas et en

utilisant un espace $\leq 2k$, si et seulement si les séquences s_i ont une sous-séquence commune de longueur q.

Point (iii) : la difficulté résulte d'une réduction paramétrée de Lcs$[k]$ à Lcs$[k, m]$ décrite dans [2], et l'appartenance résulte du Point (ii). □

La Proposition 5.17 constitue une amélioration significative par rapport aux résultats de complexité antérieurs pour le problème Lcs. D'une part, elle fournit une preuve de W[1]-difficulté pour Lcs$[q, k]$ plus simple que la preuve originale de [3]. D'autre part, elle classe précisément la complexité du problème Lcs$[k]$: le problème était seulement connu pour être W[t]-dur pour tout $t \geq 1$ [3], et nous obtenons un résultat de WNL-complétude.

Conséquences pour la complexité d'autres problèmes

Dans [5, 6], le résultat de W[t]-difficulté pour Lcs$[k]$ établi dans [3] est transféré à d'autres problèmes par réduction paramétrée. Les problèmes considérés sont les suivants :

Nom : Colored Cutwidth
Instance : un graphe $G = (V, E)$, un coloriage des arêtes $c : E \to [k]$, un entier r
Paramètre : k, r
Question : existe t-il un ordre linéaire f de G telle que pour chaque couleur $j \in [k]$, et pour chaque $i \leq |V|$, $|\{uv \in E : c(uv) = j, f(u) \leq i, f(v) > i\}| \leq r$?

Nom : Feasible Register Assignment
Instance : un digraphe acyclique $G = (V, A)$ d'ordre n, un entier positif k, une assignation des registres aux sommets $r : V \to \{R_1, ..., R_k\}$
Paramètre : k
Question : existe t-il un ordre linéaire f de G, et une séquence $S_1, ..., S_n$ de sous-ensembles de V, tels que $S_0 = \emptyset$, S_n contienne tous les sommets de G de degré interne 0, et pour tout i, $1 \leq i \leq n$, .

Nom : Domino Treewidth
Instance : un graphe $G = (V, E)$, un entier k
Paramètre : k
Question : la *domino treewidth* de G est-elle inférieure à k ? La *domino treewidth* [4] d'un graphe G est le poids minimum d'une *domino tree decomposition*. Une *domino tree decomposition* de G est une décomposition arborescente $D = (T, \{V_x : x \in V(T)\})$ de G telle que chaque $x \in V$ appartienne à au plus deux ensembles V_y ; le poids de D est $\max_{x \in V(T)}(|V_x| - 1)$.

Nom : Triangulating Colored Graphs
Instance : un graphe G, un entier k, c coloriage admissible de G
Question : existe t-il un graphe chordal $G' \supset G$ tel que $\omega(G') \leq k$ et c soit un

coloriage admissible de G' ?

L'existence de réductions paramétrées depuis LCS$[k]$, en conjonction avec la Proposition 5.15, implique la WNL-difficulté de ces problèmes :

Proposition 5.18. *Les problèmes* COLORED CUTWIDTH, FEASIBLE REGISTER ASSIGNMENT, DOMINO TREEWIDTH, TRIANGULATING COLORED GRAPHS *sont* WNL*-durs.*

On présente maintenant un résultat de WNL-complétude pour un problème sur les automates introduit dans [18]. Le problème BOUNDED DFA INTERSECTION (BDFA) demande : étant donné une famille de k automates déterministes $\mathcal{A} = \{A_1, ..., A_k\}$ sur un même alphabet Σ, et étant donné un entier q, existe t-il un mot de Σ^q qui est accepté par chaque A_i ? Le problème BDFA2 est la restriction de BDFA aux instances avec $|\Sigma| = 2$. On montre :

Proposition 5.19. *(i)* BDFA$[k]$ *est* WNL*-complet ; (ii)* BDFA2$[k]$ *est* WNL*-complet.*

Démonstration. Point (i). La WNL-difficulté se montre par réduction depuis LCS$[k]$. Etant donnée une instance $I = (\mathcal{C}, q, k)$ de LCS$[k]$, on crée une instance $I' = (\mathcal{A}, q, k)$ de BDFA$[k]$, où $\mathcal{A} = \{A_1, ..., A_k\}$ est tel que pour tout i, A_i est un automate déterministe reconnaissant l'ensemble des sous-mots de s_i. La correction de cette réduction est immédiate, et la polynomialité de la réduction résulte du fait que chaque automate A_i a une taille $O(|\Sigma||s_i|)$. L'appartenance à WNL se montre par réduction à NTMC$[k]$. Soit $I = (\mathcal{A}, q, k)$ une instance de BDFA$[k]$ avec $\mathcal{A} = \{A_1, ..., A_k\}$, $A_i = (Q_i, \Sigma, \delta_i, q_i^0, F_i)$. On construit une MTN M dont le ruban contient un tuple $t = (q_1, ..., q_k)$, avec $q_i \in Q_i$. M effectue q étapes : à l'étape i elle va effectuer $O(k)$ pas pour deviner la $i$ème lettre d'un mot solution. M démarre avec le tuple $t = (q_1^0, ..., q_k^0)$ sur son ruban. A chaque étape $i \leq N$, si le contenu du ruban est $t = (q_1, ..., q_k)$, la machine choisit de façon non déterministe une lettre $a \in \Sigma$, et remplace t par le nouveau tuple $t' = (\delta_1(q_1, a), ..., \delta_k(q_k, a))$. A la fin de l'étape q, la machine accepte si et seulement si le contenu du ruban a la forme $t = (q_1, ..., q_k)$ avec $q_i \in F_i$ pour tout $i \in [k]$. Il est facile de voir qu'il y a équivalence entre : (i) M accepte le mot vide en $q' = O(kq)$ pas en utilisant un espace $\leq 2k$, (ii) il existe un mot de Σ^q accepté par chaque A_i.

Point (ii). L'appartenance résulte du Point (i). La difficulté se montre par une réduction simple depuis BDFA$[k]$. Etant donnée une instance $I = (\mathcal{C}, q, k)$ de BDFA$[k]$ avec $\mathcal{A} = \{A_1, ..., A_k\}$ famille d'automates sur un même alphabet Σ, on construit une instance $I' = (\mathcal{C}', q', k)$ de BDFA2$[k]$ de la façon suivante. Soit Σ' un alphabet à deux éléments, on encode chaque lettre $a \in \Sigma$ par un mot $w_a \in \Sigma'^r$ avec $r = \lceil \log_2 |\Sigma| \rceil$. Chaque automate A_i' est défini à partir de A_i de la façon suivante : pour chaque état q de A_i, on remplace les transitions sortantes de q par un automate arborescent qui décode un mot $w_a \in \Sigma'^r$ et passe dans l'état $\delta(q, a)$. En posant $q' = qr$, il est alors clair que : I est une instance positive de BDFA$[k]$ si et seulement si I' est une instance positive de BDFA2$[k]$. □

5.1.5 Remarques

Remarques sur Slcs

On peut considérer deux variantes du problèmes SLCS, auxquelles on peut adapter l'algorithme de programmation dynamique de la Proposition 5.11 :

1. une variante pondérée WEIGHTED-SLCS dans laquelle on dispose d'une collection $\mathcal{C}$ sur L et d'une fonction de coût $w : L \to \mathbb{R}^+$, et où l'objectif est de trouver un ensemble $L' \subseteq L$ de coût maximum tel que $\mathcal{C}|L'$ soit compatible. Cette variante peut se résoudre en temps $O(kn^k)$ en adaptant par les relations de récurrence décrites en Proposition 5.11. Etant donnée une position π dans $\mathcal{C}$, on définit $\text{SLCS}(\pi)$ comme le coût maximum d'une séquence compatible de $\mathcal{C}|L(\pi)$. Le calcul de $\text{SLCS}(\pi)$ se fait comme en Proposition 5.11 à partir de deux valeurs $\text{SLCS}_1(\pi), \text{SLCS}_2(\pi)$, où l'on pose maintenant :

$$\text{SLCS}_1(\pi) = \max\{\text{SLCS}(succ_i(\pi)) : i \in I(\pi)\}$$
$$\text{SLCS}_2(\pi) = \max\{w(a) + \text{SLCS}(succ_a(\pi)) : a \in F(\pi)\}$$

2. une variante CONSTRAINED-SLCS dans laquelle on dispose d'une collection $\mathcal{C}$ sur L, d'un ensemble $L_0 \subseteq L$, et où l'objectif est de chercher un ensemble $L' \subseteq L$ *contenant* L_0 et de cardinal maximum tel que $\mathcal{C}|L'$ soit compatible. Cette variante peut se résoudre en temps $O(kn^k)$, en adaptant les relations de récurrence décrites en Proposition 5.11 de la façon suivante. Etant donnée une position π dans $\mathcal{C}$, on définit $\text{SLCS}(\pi)$ comme la longueur maximum de s séquence compatible de $\mathcal{C}|L(\pi)$ telle que $L_0 \cap L(\pi) \subseteq L(s)$. Le calcul de $\text{SLCS}(\pi)$ se fait à partir des valeurs $\text{SLCS}_1(\pi), \text{SLCS}_2(\pi)$ définies comme suit :

$$\text{SLCS}_1(\pi) = \max\{\text{SLCS}(succ_i(\pi)) : i \in I(\pi), \pi[i] \notin L_0\}$$
$$\text{SLCS}_2(\pi) = \max\{1 + \text{SLCS}(succ_a(\pi)) : a \in F(\pi)\}$$

Concernant l'approximation et la complexité paramétrique du problème SLCS, on peut mentionner deux questions ouvertes :

1. Quel est le ratio d'approximabilité pour SLCS et son complémentaire, en fonction du paramètre k ? On conjecture que ce ratio est $2^{\Omega(k)}$ pour SLCS, mais il pourrait être $\Omega(\log k)$ pour CSLCS.
2. Le problème est-il soluble en temps $p^{O(k)}n^c$? Un algorithme avec cette complexité serait préférable à l'algorithme en $O(k^p \times kn)$ de la Proposition 5.8, pour des petites valeurs de k.

Remarques sur la notion de WNL-difficulté

Les problèmes paramétrés suivants sont des candidats plausibles pour un résultat de WNL-difficulté.

Nom : Shortest Common Supersequence
Instance : k séquences $s_1, ..., s_k$ sur un même alphabet Σ, un entier q.
Paramètre : k
Question : existe t-il une séquence s sur Σ de longueur $\leq q$ telle que s soit une super-séquence de s_i pour tout i ?

Ce problème a été montré W[t]-dur pour tout t [12].

Nom : Bandwidth
Instance : un graphe $G = (V, E)$ et un entier k
Paramètre : k
Question : existe t-il f ordre linéaire de G tel que $uv \in E$ implique $|f(u) - f(v)| \leq k$?

Ce problème a été montré W[t]-dur pour tout t [5].

Nom : Proper Intervalizing Colored Graphs
Instance : un graphe G et c coloriage admissible de G, un entier k
Paramètre : k
Question : existe t-il un graphe d'intervalles propres $G' \supset G$ tel que $\omega(G') \leq k$ et c soit un coloriage admissible de G' ?

Ce problème a été montré W[t]-dur pour tout t [14].

Nom : Edge Disjoint Paths
Instance : un digraphe acyclique $G = (V, A)$, un entier k, des sommets $s_1, t_1, ..., s_k, t_k$
Paramètre : k
Question : existe t-il dans G k chemins arêtes-disjoints $P_1, ..., P_k$, avec P_i chemin de s_i à t_i ?

Ce problème a été montré W[1]-dur [17].

Nom : Hypertree Width
Instance : un hypergraphe $H = (V, E)$, un entier k
Paramètre : k
Question : la *hypertree width* [10] de H est-elle inférieure à k ? La *hypertree width* de H est le poids minimum d'une hypertree décomposition de H.

Une *tree decomposition* de H est un couple (T, χ) où T est un arbre et $\chi : V(T) \to 2^{V(H)}$ une fonction associant un ensemble de sommets $\chi(t) \subseteq V(H)$ à chaque sommet t de T, tel que pour chaque $e \in E(H)$ il existe $t \in V(T)$ tel que $e \subseteq \chi(t)$, et pour tout $v \in V(H)$ l'ensemble $\{t \in V(T) : v \in \chi(t)\}$ est connexe dans T.

Une *hypertree decomposition* de H est un couple $D = (T, \chi, \lambda)$ où (T, χ) est une tree-decomposition de H, et $\lambda : V(T) \to 2^{E(H)}$ est une fonction associant un ensemble d'hyperarêtes $\lambda(t) \subseteq E(H)$ à chaque chaque sommet t de T, tel que :

- pour chaque $t \in V(T)$, on a $\chi(t) \subseteq \cup\lambda(t)$;
- pour chaque $t \in V(T)$, $(\cup\lambda(t)) \cap \chi(T_t) \subseteq \chi(t)$, où T_t désigne le sous-arbre de T enraciné en t.

Le poids de D est $\min\{|\lambda(t)| : t \in V(T)\}$.

Ce problème a été montré W[2]-dur [10].

Mentionnons enfin le problème Lcs pour un alphabet borné ; ce problème a été montré W[1]-dur pour un alphabet binaire [15].

5.2 Problème Smast

Cette section est consacrée au problème Smast, et s'organise selon le plan suivant. En Section 5.2.1, on introduit des définitions relatives au problème. En Section 5.2.2, on présente des résultats algorithmiques pour Smast. En Section 5.2.3, on décrit une réduction paramétrée de Slcs à Smast, et on en déduit des résultats de difficulté paramétrique pour Smast. La Section 5.2.4 contient quelques remarques conclusives.

5.2.1 Préliminaires

Définitions

On introduit des définitions relatives aux *arbres*, aux *collections* et à la *compatibilité*. Soit L un ensemble de n éléments. Un *p-arbre* sur L est un arbre T tel que $L(T) \subseteq L$. On ne considère dans cette section que des arbres binaires.

Une *collection* sur L est une famille $\mathcal{T} = \{T_1, ..., T_k\}$ de p-arbres sur L. Pour les collections considérées, on supposera toujours que $L = \cup_{i\in[k]} L(T_i)$. Etant donné $L' \subseteq L$, la *restriction* de $\mathcal{T}$ à L' est la collection $\mathcal{T}|L'$ sur L' définie par : $\mathcal{T}|L' = \{T_1|L', ..., T_k|L'\}$.

Etant donnés deux p-arbres T, T' sur L, on dit que T *s'accorde avec* T' si et seulement si $T|\ L(T') = T'|L(T)$. Un *superarbre d'accord* pour $\mathcal{T}$ est un p-arbre sur L qui s'accorde avec tout $T_i \in \mathcal{T}$. Un *superarbre d'accord total* pour $\mathcal{T}$ est un superarbre d'accord pour $\mathcal{T}$ dont l'ensemble d'étiquettes est L. On dit que $\mathcal{T}$ est *compatible* si et seulement si elle admet un superarbre d'accord total. Un *conflit* entre $\mathcal{T}$ est un ensemble $L' \subseteq L$ tel que $\mathcal{T}|L'$ est non compatible.

Exemple 5.20. *Considérons la collection $\mathcal{T}$ sur $L = \{a, b, c, d, e, f\}$ représentée ci-dessous. Alors les p-arbres $((((a, b), c), e), f)$ et $(((a, b), c), (e, f))$ sont des superarbres d'accord pour $\mathcal{T}$. Cependant, $\mathcal{T}$ est non compatible, car elle contient par exemple le conflit $\{a, b, c, d\}$.*

Le problème Smast consiste, étant donnée une collection $\mathcal{T}$ sur L, à trouver un ensemble $L' \subseteq L$ de cardinal maximum tel que $\mathcal{T}|L'$ soit compatible. Cela revient à chercher un superarbre d'accord pour $\mathcal{T}$ de taille maximum. Le problème complémentaire CSmast consiste, étant donnée $\mathcal{T}$ collection sur L, à

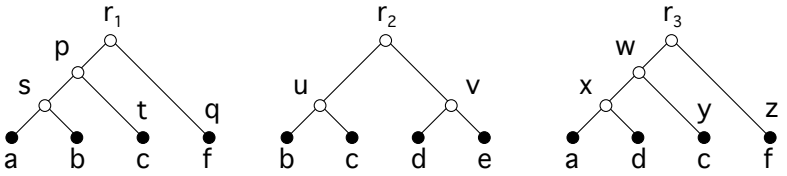


FIG. 5.3 – Une collection $\mathcal{T}$ de trois p-arbres sur l'ensemble d'étiquettes $L = \{a, b, c, d, e, f\}$.

trouver $L' \subseteq L$ de cardinal minimum tel que $\mathcal{T}|(L \backslash L')$ soit compatible. La taille d'une solution optimale pour SMAST, resp. CSMAST, sera notée SMAST($\mathcal{T}$), resp. CSMAST($\mathcal{T}$).

Exemple 5.21. *Pour la collection définie ci-dessus, on a* SMAST($\mathcal{T}$) = 5 *et* CSMAST($\mathcal{T}$) = 1.

Les définitions suivantes seront utiles en Section 5.2.2. Si T est un p-arbre sur L, on pose $N^{\perp}(T) = N(T) \cup \{\perp\}$. On étend la notation $T(u)$ à $u \in N^{\perp}(T)$ tel que $T(\perp)$ est le p-arbre vide. On étend la relation $\leq_T$ à $N^{\perp}(T)$ telle que $\perp \leq_T u$ pour tout $u \in N^{\perp}(T)$.

Une *position* dans $\mathcal{T}$ est un tuple $\pi = (x_1, ..., x_k)$ tel que $x_i \in N^{\perp}(T_i)$ pour tout $i \in [k]$. On note $\pi[i] = x_i$ la $i$ème composante de π. La *position initiale* est $\pi_{\top} = (r(T_1), ..., r(T_k))$. La *position finale* est $\pi_{\perp} = (\perp, ..., \perp)$. L'*ensemble d'indices* d'une position π est $I(\pi) = \{i \in [k] : \pi[i] \neq \perp\}$. L'*ensemble d'étiquettes* d'une position π est $L(\pi) = \cup_{i \in [k]} L(\pi[i])$. On définit une relation d'ordre $\leq_{\mathcal{T}}$ sur les positions dans $\mathcal{T}$ par : $\pi \leq_{\mathcal{T}} \pi'$ si et seulement si pour tout $i \in [k]$, $\pi[i] \leq_{T_i} \pi'[i]$. On note $<_{\mathcal{T}}$ sa partie stricte.

Exemple 5.22. *Pour la collection définie ci-dessus, considérons les positions* $\pi = \pi_{\top} = (r_1, r_2, r_3)$ *et* $\pi' = (p, u, w)$. *Alors* $\pi' <_{\mathcal{T}} \pi$. *Et* $L(\pi) = \{a, b, c, d, e, f\}$, $L(\pi') = \{a, b, c, d\}$.

Résultats

On obtient des résultats sur l'approximabilité et la complexité paramétrique de SMAST. D'une part, on montre que CSMAST est $2k$-approximable en temps $O(kn^2)$, et peut se résoudre en temps $O((2k)^p \times kn^2)$. D'autre part, on montre que SMAST est 2^k-approximable en temps $O(kn^3)$, et est soluble en temps $O((6n)^k)$ par programmation dynamique. On obtient également des résultats de difficulté paramétrique pour SMAST, en transférant les résultats obtenus pour SLCS en Section 5.1. On montre en particulier que le problème est WNL-dur pour le paramètre k, mais on n'a pas ici de résultat de complétude à la différence du problème SLCS.

Nos résultats sur la complexité paramétrique de SMAST sont résumés dans le tableau suivant :

Paramètres	Complexité de Smast	Source
q	W[1]-complet	Props 5.42
q, k	W[1]-complet	”
p	W[2]-dur	cf. Chap 6, Prop 6.30
k, p	FPT par un algorithme en $O((2k)^p \times kn^2)$	Prop 5.31
k	WNL-dur soluble en temps $O((6n)^k)$	Prop 5.42 Prop 5.34

Fig. 5.4 – Complexité paramétrique de Smast

On rappelle que q est une borne sur le nombre d'étiquettes à conserver, et que p est une borne sur le nombre d'étiquettes à supprimer, suivant la convention formulée en section 2.3.

5.2.2 Algorithmes

Algorithmes pour CSmast

On décrit en premier lieu des algorithmes FPT et des algorithmes d'approximation pour le problème CSmast (Proposition 5.31). Ils reposent sur le résultat suivant :

Proposition 5.23. *Etant donnée une collection $\mathcal{T}$, en temps $O(kn^2)$ on peut décider si $\mathcal{T}$ est compatible, renvoyer un superarbre d'accord total en cas de résultat positif, ou renvoyer un conflit entre $\mathcal{T}$ de cardinal $\leq 2k$ en cas de résultat négatif.*

Il est bien connu que la compatibilité d'une collection de triplets (et donc d'une collection d'arbres) peut se décider en temps polynomial, par l'algorithme Build ([1, 13]. Cependant, en cas d'incompatibilité, cet algorithme ne fournit pas de conflit responsable de l'incompatibilité. Notre objectif étant ici d'obtenir un conflit, on procède différemment. Tout comme Build, l'algorithme présenté ci-dessous utilise une approche récursive descendante. Chaque étape construit un graphe dont les composantes connexes correspondent à des sous-arbres du superarbre recherché. Ici, nous remplaçons le graphe utilisé dans Build par un graphe $G(\mathcal{T}, \pi)$; lorsque ce graphe est connexe, un arbre couvrant du graphe fournit un conflit de taille $\leq 2k$. Les étapes récursives de l'algorithme considèrent des positions particulières π de la collection $\mathcal{T}$. On définit d'abord ces positions, et on définit ensuite le graphe $G(\mathcal{T}, \pi)$.

On suppose dans la suite que $\mathcal{T} = \{T_1, ..., T_k\}$ est une collection sur L. Une position π dans $\mathcal{T}$ est *réduite* si et seulement si aucune de ses composantes n'est une feuille (i.e. chaque composante est soit $\perp$ soit un noeud interne). A chaque position π, on associe une position réduite $\pi \downarrow$ en remplaçant par $\perp$ chaque composante de π qui est une feuille. Etant donnée une position π de $\mathcal{T}$, on

définit la collection $\mathcal{T}(\pi)$ sur $L(\pi)$ par : $\mathcal{T}(\pi) = \{T_1(\pi[1]), ..., T_k(\pi[k])\}$; on dit que π est *compatible* si et seulement si $\mathcal{T}(\pi)$ est compatible. Observons que :

Lemme 5.24. *1. $\pi_\perp$ est compatible.*

2. π est compatible si et seulement si $\pi\downarrow$ est compatible.

3. si $\mathcal{T}$ est compatible alors $\mathcal{T}(\pi)$ est compatible, pour toute position π de $\mathcal{T}$.

Démonstration. Les points 1 et 2 résultent des définitions. Le point 3 résulte du fait que tout arbre de $\mathcal{T}(\pi)$ est une restriction d'un arbre de $\mathcal{T}$. La compatibilité de $\mathcal{T}(\pi)$ résulte donc de celle de $\mathcal{T}$. □

On définit maintenant le graphe $G(\mathcal{T}, \pi)$ pour π position réduite : (i) son ensemble de sommets est $V = \cup_{i \in I(\pi)} \mathsf{children}_{T_i}(\pi[i])$; (ii) deux sommets $x, y \in V$ sont adjacents si et seulement si $L(x) \cap L(y) \neq \emptyset$. En d'autres termes, $G(\mathcal{T}, \pi)$ est le graphe d'intersection du système d'ensembles $\{L(x) : x \in V\}$. Notons que construire une représentation explicite de ce graphe (par matrice d'adjacence ou listes d'incidences) prend un temps $O(k^2 n)$. On verra dans la suite comment effectuer un test de connexité sur ce graphe en temps $O(kn)$ sans construire explicitement le graphe, mais en utilisant le modèle d'intersection fourni par les ensembles $L(x)$.

Etant donné $V' \subseteq V$, on définit le successeur de π relativement à V', noté $succ_{V'}(\pi)$, comme la position π' telle que

- si $\pi[i] = \perp$, alors $\pi'[i] = \perp$;
- si $\pi[i]$ est un noeud interne u_i de T_i de fils v_i, v_i', alors

$$\begin{cases} \text{si } v_i \in V' \text{ et } v_i' \notin V' \text{ alors } \pi'[i] = v_i, \\ \text{si } v_i \notin V' \text{ et } v_i' \in V' \text{ alors } \pi'[i] = v_i', \\ \text{si } v_i \in V' \text{ et } v_i' \in V' \text{ alors } \pi'[i] = u_i, \\ \text{si } v_i \notin V' \text{ et } v_i' \notin V' \text{ alors } \pi'[i] = \perp. \end{cases}$$

En d'autres termes, $succ_{V'}(\pi)$ est la position dont la $i$ème composante est la racine du sous-arbre de T_i contenant les noeuds de V', ou est $\perp$ si V' ne contient aucun noeud de T_i. Ces définitions sont illustrées en Figure 5.5.A.

On donne ci-dessous un algorithme récursif pour décider la compatibilité d'une position donnée. Appeler cet algorithme avec la position $\pi_\top$ permettra de décider la compatibilité de $\mathcal{T}$. Chaque étape récursive de l'algorithme correspond à une position π dans $\mathcal{T}$. Pour le reste de l'algorithme, en considérant $\pi\downarrow$ au lieu de π, on peut supposer que π est une position réduite. Le cas de base correspond à $\pi = \pi_\perp$, l'algorithme répond alors positivement puisque $\pi_\perp$ est compatible. Le cas général de la récursion porte sur une position réduite $\pi \neq \pi_\perp$, pour laquelle l'algorithme essaie d'identifier deux successeurs π_1, π_2 correspondant aux sous-arbres fils d'un superarbre d'accord hypothétique de $\mathcal{T}(\pi)$.

Dans ce but, il effectue un test de connexité sur le graphe $G(\mathcal{T}, \pi)$. Si le graphe est non connexe, alors le test de connexité fournit une partition de V en deux ensembles déconnectés V_1, V_2 ; l'algorithme considère alors les positions successeurs $\pi_1 = succ_{V_1}(\pi), \pi_2 = succ_{V_2}(\pi)$, et émet des appels récursifs pour

ces positions. La correction de cette étape est énoncée dans le Lemme 5.28. Si le graphe est connexe, alors le test de connexité fournit un arbre couvrant de $G(\mathcal{T}, \pi)$, et on obtient un conflit en choisissant, pour chaque arête uv de l'arbre, une étiquette présente dans $L(u) \cap L(v)$; la correction de cette étape est énoncée dans le Lemme 5.29. Enfin, le test de connexité utilisé est décrit dans le Lemme 5.30.

Ce processus est illustré en Figure 5.5.

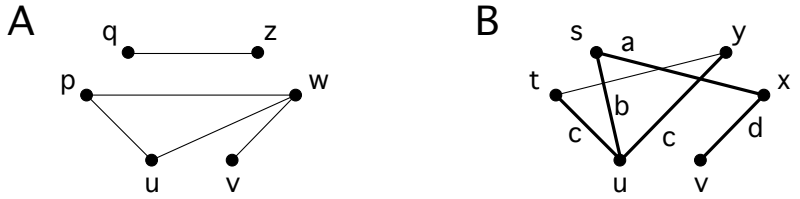


FIG. 5.5 – A. Le graphe $G(\mathcal{T}, \pi_\top)$ de la position $\pi_\top = (r_1, r_2, r_3)$ pour la collection représentée sur la Figure 5.3. Ce graphe est non connexe, les deux composantes connexes indiquent les deux positions successeurs de $\pi_\top$, $\pi_1 = (p, r_2, w)$ et $\pi_2 = (q, \perp, z)$. B. Le graphe $G(\mathcal{T}, \pi_1)$ est connexe. En choisissant un arbre couvrant de ce graphe (représenté en gras), et une étiquette commune aux deux sous-arbres correspondant aux extrémités de chaque arête de l'arbre couvrant, on obtient un conflit $C = \{a, b, c, d\}$ entre $\mathcal{T}$.

Etant donné $V' \subseteq V$, posons $L(V') = \cup_{x \in V'} L(x)$. On peut faire les observations suivantes :

Observation 5.25. *Deux ensembles $V_1, V_2 \subseteq V$ sont connectés dans $G(\mathcal{T}, \pi)$ si et seulement si $L(V_1) \cap L(V_2) \neq \emptyset$.*

Démonstration. V_1, V_2 sont connectés si et seulement si il existe $u \in V_1, v \in V_2$ tels que $L(u) \cap L(v) \neq \emptyset$, ce qui est équivalent à $L(V_1) \cap L(V_2) \neq \emptyset$. □

Observation 5.26. *Soit $V' \subseteq V$, et soit $\pi' = succ_{V'}(\pi)$. Alors $L(\pi') = L(V')$.*

Démonstration. Pour chaque $i \in [k]$, soit V'_i l'ensemble des noeuds de T_i présents dans V'. Observons que $L(V') = \cup_{i \in [k]} L(V'_i)$, et que $L(\pi') = \cup_{i \in [k]} L(\pi'[i])$. On conclut en remarquant que pour tout $i \in [k]$, $L(V'_i) = L(\pi'[i])$, par définition de la notation *succ*. □

Observation 5.27. *Soit $V_1, V_2 \subseteq V$, et soient $\pi_1 = succ_{V_1}(\pi), \pi_2 = succ_{V_2}(\pi)$.*
(i) si $V_1 \cup V_2 = V$, alors $L(\pi_1) \cup L(\pi_2) = L(\pi)$;
(ii) si de plus V_1, V_2 sont déconnectés dans $G(\mathcal{T}, \pi)$, alors $L(\pi_1), L(\pi_2)$ forme une partition de $L(\pi)$.

Démonstration. Point (i) : par l'Observation 5.26, on a $L(\pi_j) = L(V_j)$, $L(\pi) = L(V)$, et on conclut en observant que $L(V_1) \cup L(V_2) = L(V)$.

Point (ii) : on a $L(\pi_1) \cup L(\pi_2) = L(\pi)$ par le point (i). De plus, comme $L(\pi_1) = L(V_1), L(\pi_2) = L(V_2)$, et comme V_1, V_2 sont déconnectés dans $G(\mathcal{T}, \pi)$, par l'Observation 5.25 on conclut que $L(\pi_1), L(\pi_2)$ sont disjoints. □

Considérons une position réduite $\pi \neq \pi_\top$. Le lemme suivant fournit une caractérisation récursive de la compatibilité pour une telle position.

Lemme 5.28. *Soit π une position réduite $\neq \pi_\bot$. Les points suivants sont équivalents :*

- *π est compatible ;*
- *il existe une partition V_1, V_2 de V telle que (i) V_1, V_2 sont déconnectés dans $G(\mathcal{T}, \pi)$, (ii) $succ_{V_1}(\pi)$ et $succ_{V_2}(\pi)$ sont compatibles.*

Démonstration. ($\Rightarrow$). Supposons que π est compatible. Soit S un superarbre d'accord total pour $\mathcal{T}(\pi)$. Comme $\pi \neq \pi_\bot$ et comme π est réduite, on a $|L(\pi)| \geq 2$, et donc $S = (S_1, S_2)$. Comme S est un superarbre d'accord total pour $\mathcal{T}(\pi)$, on a $T_i(\pi[i]) \leq S$ pour chaque $i \in [k]$. Définissons une partition V_1, V_2 de V de la façon suivante. Posons $u_i = \pi[i]$, et supposons que u_i est un noeud interne de T_i, de fils v_i, v_i'. Alors $T_i(u_i) = (T_i(v_i), T_i(v_i'))$. Alors $T_i(u_i) \leq S$ implique que $T_i(v_i) \leq S_1$ ou $T_i(v_i) \leq S_2$: on ajoute v_i à V_1 dans le premier cas, à V_2 dans le second cas. On procède de façon similaire pour v_i'.

Montrons d'abord le point (i). Observons que $L(V_1) \subseteq L(S_1)$ et $L(V_2) \subseteq L(S_2)$. Comme $L(S_1), L(S_2)$ sont disjoints, il résulte que V_1, V_2 sont déconnectés par l'Observation 5.25. Montrons maintenant le point (ii). Posons $\pi_j = succ_{V_j}(\pi)$, alors π_j est une position de $\mathcal{T}(\pi)$, et comme $\mathcal{T}(\pi)$ est compatible par hypothèse, il résulte que π_j est compatible par le point 3 du Lemme 5.24.

($\Leftarrow$). Supposons qu'il existe une partition V_1, V_2 de V vérifiant les points (i), (ii). Posons $\pi_j = succ_{V_j}(\pi)$. Puisque π_j est compatible par hypothèse, il existe S_j superarbre d'accord total pour $\mathcal{T}(\pi_j)$. Par l'Observation 5.26, on a $L(S_j) = L(\pi_j) = L(V_j)$. Comme V_1, V_2 sont déconnectés dans $G(\mathcal{T}, \pi)$, il résulte que $L(\pi_1), L(\pi_2)$ forment une partition de $L(\pi)$ par l'Observation 5.27. On peut donc définir le p-arbre $S = (S_1, S_2)$ sur $L(\pi)$. Montrons que S est un superarbre d'accord total pour $\mathcal{T}(\pi)$. On doit montrer que $T_i(\pi[i]) \leq S$ pour tout $i \in [k]$.

Fixons un tel i, soit $u_i = \pi[i]$. Si $u_i = \bot$, le résultat est clair. Supposons maintenant que u_i est un noeud interne de T_i, et soient v_i, v_i' ses deux fils. On considère trois cas. Si $v_i, v_i' \in V_1$: alors $\pi_1[i] = u_i$, et on a donc $T_i(u_i) \leq S_1$, dont on déduit $T_i(u_i) \leq S$. Si $v_i, v_i' \in V_2$: alors $\pi_2[i] = u_i$, et on a donc $T_i(u_i) \leq S_2$, dont on déduit $T_i(u_i) \leq S$. Si $v_i \in V_1, v_i' \in V_2$: alors $\pi_1[i] = v_i$, ce qui implique $T_i(v_i) \leq S_1$, et $\pi_2[i] = v_i'$, ce qui implique $T_i(v_i') \leq S_2$. On conclut que $T_i(u_i) = (T_i(v_i), T_i(v_i')) \leq (S_1, S_2) = S$. □

Si le graphe $G(\mathcal{T}, \pi)$ est connexe, on montre qu'un arbre couvrant du graphe permet d'identifier un conflit entre $\mathcal{T}$ de petite taille.

Lemme 5.29. *Soit π une position réduite $\neq \pi_\bot$. Supposons que $G(\mathcal{T}, \pi)$ est connexe, et soit $T = (V, F)$ un arbre couvrant de $G(\mathcal{T}, \pi)$. Pour chaque arête $e = \{u, v\} \in F$, choisissons $l_e \in L(u) \cap L(v)$. Alors $C = \{l_e : e \in F\}$ est un conflit entre $\mathcal{T}$.*

Démonstration. On montre que $\mathcal{T}' := \mathcal{T}|C$ est incompatible. Pour chaque $i \in I(\pi)$, soit $u_i = \pi[i]$, et soient v_i, v_i' ses deux fils dans T_i. Par définition de C, les ensembles $L(v_i) \cap C, L(v_i') \cap C$ sont non vides, et donc aux noeuds v_i, v_i', u_i

il correspond des noeuds $\tilde{v_i}, \tilde{v_i'}, \tilde{u_i}$ dans $T_i|C$. Définissons π' position dans $\mathcal{T}'$ en posant $\pi'[i] = \perp$ si $i \notin I(\pi)$, $\pi'[i] = \tilde{u_i}$ si $i \in I(\pi)$. Considérons le graphe $G(\mathcal{T}', \pi')$, alors par définition de C, pour chaque arête $\{x, y\}$ de T, l'arête $\{\tilde{x}, \tilde{y}\}$ est présente dans $G(\mathcal{T}', \pi')$, donc l'arbre T' formé de ces arêtes est un arbre couvrant de $G(\mathcal{T}', \pi')$, qui est donc connexe. Par le Lemme 5.28, on conclut que π' est une position non compatible de $\mathcal{T}'$, et donc $\mathcal{T}'$ est non compatible par le Point 3 du Lemme 5.24. □

On décrit maintenant le test de connexité utilisé.

Soit $G = (V, E)$ un graphe. Un *modèle d'intersection* de G est un tuple $M = (S, \{S_x : x \in V\})$, où S est un ensemble, chaque S_x est un sous-ensemble de S, tels que : pour tous $x, y \in V$, $xy \in E$ si et seulement si $S_x \cap S_y \neq \emptyset$. Une *partition déconnectée* de G est une partition de V en deux ensembles V_1, V_2 tels qu'aucune arête ne joigne V_1 à V_2.

Ainsi, si $G(\mathcal{T}, \pi) = (V, E)$, alors $M = (L, \{L(x) : x \in V\})$ est un modèle d'intersection du graphe. Pour décider la connexité de $G(\mathcal{T}, \pi)$, on utilisera donc la procédure ConnectivityTest décrite ci-après.

Lemme 5.30. *Il existe une procédure* ConnectivityTest *qui : étant donné un graphe $G = (V, E)$ représenté par un modèle d'intersection $M = (S, \{S_x : x \in V\})$, en temps $O(|V||S|)$ effectue un test de connexité sur G, et :*

- *si G est connexe, renvoie un arbre couvrant de G ;*
- *si G est non connexe, renvoie une partition déconnectée de G.*

Démonstration. La procédure effectue un parcours du graphe G, en démarrant à un sommet arbitraire, et en maintenant l'information suivante au cours du parcours : (i) l'ensemble U des noeuds déjà visités, (ii) un ensemble F d'arêtes formant un arbre couvrant de $G[U]$. Tant que $U \subset V$, l'algorithme cherche une *arête transverse*, qui est une arête $e = uv \in E$ avec $u \in U, v \in \bar{U}$. S'il trouve une telle arête, alors v est ajouté à U et e est ajouté à F. Si l'algorithme parvient à trouver une arête transverse à chaque étape du parcours, alors lorsque tous les sommets ont été visités il répond positivement et renvoie (V, F) arbre couvrant de G. Si au contraire à une étape l'algorithme n'a pas trouvé d'arête transverse, alors il répond négativement et renvoie $(U, \bar{U})$ partition déconnectée de G.

On montre qu'en utilisant des structures de données appropriées, une étape de l'algorithme s'effectue en temps $O(|S|)$. Pour chaque $x \in S$, soit $V_x = \{v \in V : x \in S_v\}$. On maintient pour chaque $x \in S$, deux listes représentant les ensembles $U_x = V_x \cap U$ et $\bar{U_x} = V_x \cap \bar{U}$. L'initialisation de ces listes au début de l'algorithme se fait en temps $O(|V||S|)$. De plus, à une étape de l'algorithme : (i) on peut trouver une arête transverse en temps $O(|S|)$, (ii) on peut mettre à jour ces listes en temps $O(|S|)$. Pour justifier le point (i), observons que trouver une arête transverse revient simplement à trouver un élément $x \in S$ tel que $U_x, \bar{U_x}$ sont non vides ; si on trouve un tel élément x alors en choisissant $u \in U_x, v \in \bar{U_x}$ on obtient une arête transverse uv ; clairement, ces opérations peuvent se faire en temps $O(|S|)$. Pour justifier le point (ii), observons que lors de la visite d'un nouveau sommet v, on doit, pour chaque $x \in S_v$, ajouter v à U_x et supprimer v de $\bar{U_x}$, ce qui peut se faire en temps $O(|S|)$ par un chaînage approprié. □

Grâce aux lemmes précédents, on est maintenant en mesure de prouver la Proposition 5.23.

Preuve de la Proposition 5.23. On définit une procédure IsCompatible(π) qui prend une position π dans $\mathcal{T}$, décide si π est compatible, ou renvoie un conflit de taille $\leq 2k$ en cas d'incompatibilité. La procédure est la suivante : (i) calculer $\pi' = \pi\downarrow$; (ii) renvoyer vrai si $\pi' = \pi_\perp$; (iii) si $\pi' \neq \pi_\perp$ tester si $G(\mathcal{T}, \pi')$ est connexe par la procédure ConnectivityTest du Lemme 5.30.

- si le graphe est connexe, alors soit $T = (V, F)$ un arbre couvrant de $G(\mathcal{T}, \pi')$, pour chaque arête $e = uv \in F$ choisir $l_e \in L(u) \cap L(v)$, et renvoyer $C = \{l_e : e \in F\}$.
- si le graphe est non connexe, alors soit V_1, V_2 une partition de V en deux ensembles déconnectés. Construire les positions π_1, π_2 où $\pi_i = succ_{V_i}(\pi')$. Renvoyer vrai si IsCompatible(π_1) ou IsCompatible(π_2) renvoie vrai ; sinon renvoyer le conflit obtenu par l'un des deux appels récursifs.

Pour décider si $\mathcal{T}$ est compatible, on appelle IsCompatible($\pi_\top$).

La correction de la procédure IsCompatible résulte des Lemmes 5.28 et 5.29. Justifions maintenant le temps d'exécution. Lorsqu'un appel à la procédure IsCompatible(π) fait deux appels récursifs pour des positions π_1, π_2, les ensembles $L(\pi_1), L(\pi_2)$ forment une partition de $L(\pi)$ par l'Observation 5.27. Il résulte que le nombre total d'appels récursifs à IsCompatible est $O(n)$. Chaque appel prend un temps $O(kn)$, dominé par le temps d'exécution de la procédure ConnectivityTest. On conclut que le temps d'exécution total est $O(kn^2)$. □

On obtient un algorithme FPT et un algorithme d'approximation pour CMast, comme conséquence de la Proposition 5.23.

Proposition 5.31. *(i)* CSmast *peut être résolu en temps* $O((2k)^p \times kn^2)$*, (ii)* CMast *peut être approximé à un facteur* $2k$ *en temps* $O(kn^2)$*.*

Démonstration. Le point (i) résulte de la Proposition 5.23 en utilisant la recherche bornée. Le point (ii) s'obtient en adaptant l'algorithme de la Proposition 5.23 de façon à éliminer les conflits à la volée. Plus précisément, l'algorithme maintient un ensemble P d'étiquettes impliquées dans des conflits disjoints. Etant donnée π position dans $\mathcal{T}$, on définit le graphe $G(\mathcal{T}, \pi, P) = (V, E)$ (variante du graphe $G(\mathcal{T}, \pi)$) comme le graphe d'intersection du système d'ensembles $\{L(x)\backslash P : x \in V\}$.

La procédure IsCompatible(π) procède maintenant comme suit. Elle calcule $\pi' = \pi\downarrow$, renvoie vrai si $\pi' = \pi_\perp$. Si $\pi' \neq \pi_\perp$, elle répète l'opération suivante :

- tester si $G := G(\mathcal{T}, \pi', P)$ est connexe par la procédure ConnectivityTest du Lemme 5.30 ;
- si G est connexe, obtenir un conflit C de cardinal $\leq 2k$ comme auparavant, et faire $P \leftarrow P \cup C$.

tant que G est connexe. Lorsque finalement G est non connexe, la procédure construit π_1, π_2 comme auparavant, et appelle récursivement IsCompatible(π_1) et IsCompatible(π_2).

On exécute l'algorithme pour $\mathcal{T}$ en initialisant P à $\emptyset$, en appelant la procédure IsCompatible($\pi_\top$), et en renvoyant l'ensemble P obtenu.

Il est clair que l'algorithme calcule une $2k$-approximation de CSmast. Justifions maintenant son temps d'exécution. Soit T_{rc} l'arbre des appels récursifs de la procédure IsCompatible. Chaque noeud u de T_{rc} correspond à un appel à IsCompatible ; au cours de cet appel, notons m_u le nombre d'éléments ajoutés dans P. Observons qu'au cours de l'appel à IsCompatible associé à u, chaque appel à la procédure ConnectivityTest (sauf le premier) est consécutif à l'ajout d'au moins un élément à P. Le nombre total d'appels à ConnectivityTest est donc $\leq \sum_{u \text{ noeud de } T_{rc}} (m_u + 1) = O(n)$. Comme chaque appel prend un temps $O(kn)$, on conclut que le temps total de l'algorithme est $O(kn^2)$. □

Algorithmes pour Smast

On présente dans cette section des algorithmes polynomiaux ainsi qu'un algorithme d'approximation pour le problème Smast.

Les deux résultats suivants sont les analogues de résultats similaires pour Slcs (Propositions 5.9 et 5.10). On montre que Smast peut se résoudre en temps polynomial pour des collections particulières. On dit qu'une collection $\mathcal{T} = \{T_1, ..., T_k\}$ sur L est *complète* si et seulement si chaque étiquette de L apparaît dans chaque p-arbre. On dit que $\mathcal{T}$ est *précomplète* si et seulement si chaque étiquette de L apparaît dans 1 ou k p-arbres.

Proposition 5.32. Smast *peut être résolu en temps* $O(kn^3)$ *si l'instance est une collection complète, ou une collection précomplète.*

Démonstration. Si $\mathcal{T}$ est une collection complète sur L, alors résoudre Smast sur l'instance $\mathcal{T}$ revient à résoudre le problème Mast sur l'instance $\mathcal{T}$, et peut donc se faire en temps $O(kn^3)$ par la Proposition 4.23. Si $\mathcal{T}$ est une collection précomplète sur L, soit L' l'ensemble des étiquettes qui apparaissent dans chaque p-arbre, et soit L'' les autres étiquettes. Par le même argument qu'en Proposition 5.9, on a $\text{Smast}(\mathcal{T}) = \text{Smast}(\mathcal{T}|L') + |L''|$, et on conclut que $\text{Smast}(\mathcal{T})$ peut être calculé en temps $O(kn^3)$. □

Comme conséquence du résultat précédent, on obtient un algorithme d'approximation pour le problème Smast général.

Proposition 5.33. Smast *peut être approximé à un facteur* 2^k *en temps* $O(kn^3)$.

Démonstration. On utilise la technique d'approximation par partitionnement, de façon similaire à la preuve de la Proposition 5.10. □

On montre maintenant que Smast peut être résolu en temps polynomial pour k fixé, par programmation dynamique.

Proposition 5.34. Smast *peut être résolu en temps* $O((6n)^k)$ *et en espace* $O((2n)^k)$.

L'algorithme calcule par programmations dynamiques des valeurs $\textsc{Smast}(\pi)$ pour chaque position π. Soit $\mathcal{T} = \{T_1, ..., T_k\}$ une collection et soit π une position dans $\mathcal{T}$. On dit qu'une étiquette $x \in L$ est *autorisée* en π si et seulement si pour tout $i \in [k]$ tel que $x \in L(T_i)$, on a $x \leq_{T_i} \pi[i]$. On note $A(\pi)$ l'ensemble des étiquettes autorisées en π. On définit $\textsc{Smast}(\pi)$ comme la taille d'un plus grand ensemble $L' \subseteq A(\pi)$ tel que $\mathcal{T}|L'$ soit compatible.

On va maintenant donner une seconde définition de $\textsc{Smast}(\pi)$, plus adaptée aux besoins de la preuve. Tout d'abord, on caractérise la relation « s'accorde avec » en terme de *plongements partiels*. Soient T, T' deux p-arbres, on dit qu'un plongement partiel de T dans T' est une fonction $\phi : N(T) \to N(T') \cup \{\perp\}$ telle que :

- pour tout u feuille de T, on a $\phi(u) = \perp$ si $u \notin L(T')$, ou $\phi(u) = u$ sinon,
- pour tout u noeud interne de T de fils $u_1, ..., u_p$, soit $N = \{j : \phi(u_j) \neq \perp\}$, alors (i) soit $N = \emptyset$ et $\phi(u) = \perp$, (ii) soit $N = \{i\}$ et $\phi(u) = \phi(u_i)$, (iii) soit $|N| \geq 2$ et $\phi(u_i) <_{T'} \phi(u)$ pour chaque $i \in N$, et les noeuds $\mathsf{child}_{T'}(\phi(u), \phi(u_i))$ $(i \in V)$ sont distincts.

Alors T s'accorde avec T' si et seulement si il existe un plongement partiel de T dans T' (ou de façon équivalente un plongement partiel de T' dans T).

On note $\textsc{Ast}(\pi)$ l'ensemble des p-arbres S sur L tels que (i) S est un super-arbre d'accord pour $\mathcal{T}$, (ii) pour chaque i, le plongement partiel $\phi_i : S \to T_i$ est tel que $\phi_i(r(S)) \leq_{T_i} \pi[i]$. On note $\textsc{Smast}(\pi)$ la taille d'un plus grand p-arbre de $\textsc{Ast}(\pi)$.

L'algorithme calcule les valeurs $\textsc{Smast}(\pi)$ pour chaque position π, à l'aide d'une relation de récurrence dont le cas de base est énoncé dans le Lemme 5.36 et donc le cas général est énoncé dans le Lemme 5.38. La relation de récurrence est fondée sur l'ordre $\leq_{\mathcal{T}}$ sur les positions : étant donnée une position π, $\textsc{Smast}(\pi)$ sera calculé à partir des valeurs $\textsc{Smast}(\pi')$ pour $\pi' <_{\mathcal{T}} \pi$. A la fin de l'algorithme, $\textsc{Smast}(\mathcal{T})$ est obtenu à partir de $\textsc{Smast}(\pi_\top)$.

Notons que :

Observation 5.35. *Si $\pi' \leq_{\mathcal{T}} \pi$, alors $\textsc{Ast}(\pi') \subseteq \textsc{Ast}(\pi)$.*

Démonstration. Supposons que $S \in \textsc{Ast}(\pi')$. Soit $i \in [k]$. Considérons le plongement partiel correspondant $\phi_i : S \to T_i$, alors $\phi_i(r(S)) \leq_{T_i} \pi'[i] \leq_{T_i} \pi[i]$. On conclut que $S \in \textsc{Ast}(\pi)$. □

Le cas de base de la récurrence correspond aux positions *terminales* : une position π est *terminale* si et seulement si pour chaque $i \in [k]$, $\pi[i]$ est soit une feuille, soit $\perp$. Pour une position terminale π, $L(\pi)$ est l'ensemble des étiquettes apparaissant comme composantes de π. On dit qu'un élément $x \in L(\pi)$ est *maximalement présent* si et seulement si pour chaque $i \in [k]$, $x \in L(T_i)$ implique $\pi[i] = x$. Notons $P(\pi)$ l'ensemble des éléments maximalement présents de $L(\pi)$. On montre :

Lemme 5.36. *Soit π une position terminale. Alors $\textsc{Smast}(\pi) = |P(\pi)|$.*

Démonstration. On montre d'abord que $\textsc{Smast}(\pi) \geq |P(\pi)|$. Pour cela, on montre que : si S est un p-arbre quelconque sur l'ensemble d'étiquettes $P(\pi)$, alors $S \in \textsc{Ast}(\pi)$. En effet, pour chaque $i \in [k]$ définissons ϕ_i comme suit :

– si $\pi[i] = \perp$, alors $\phi_i(u) = \perp$ pour chaque $u \in L(S)$;
– si $\pi[i]$ est une feuille x de T_i, alors $\phi_i(u) = x$ si $x \in S(u)$, sinon $\phi_i(u) = \perp$.
Alors ϕ_i est un plongement partiel de S dans T_i tel que $\phi_i(r(S)) \leq_{T_i} \pi[i]$. On conclut que $S \in \text{AST}(\pi)$.

On montre à présent que $\text{SMAST}(\pi) \leq |P(\pi)|$. Pour cela, on montre que : si $S \in \text{AST}(\pi)$, alors $L(S) \subseteq P(\pi)$. En effet, considérons un tel p-arbre S, et pour chaque $i \in [k]$ considérons le plongement partiel $\phi_i : S \to T_i$. Soit un élément $x \in L(S)$, et soit $i \in [k]$ tel que $x \in L(T_i)$, montrons que $\pi[i] = x$. Par définition d'un plongement partiel, on a : $\phi_i(x) = x$. Comme $\phi_i(x) \leq_{T_i} \pi[i]$ et comme $\pi[i]$ est une feuille (puisque π est terminal) il résulte que $\pi[i] = x$. On conclut que $L(T) \subseteq P(\pi)$. □

On décrit maintenant le cas général de la relation de récurrence, correspondant aux positions non terminales. Si π est non terminale, alors $\text{SMAST}(\pi)$ est calculé à partir de deux valeurs, $\text{SMAST}_1(\pi)$ et $\text{SMAST}_2(\pi)$.

Définissons d'abord $\text{SMAST}_1(\pi)$. On dit qu'une position π' est un *successeur* de π si et seulement si il existe $i \in [k]$ tel que $\pi'[i]$ est un fils de $\pi[i]$, et $\pi'[j] = \pi[j]$ pour tout $j \neq i$. On note $Succ(\pi)$ l'ensemble des successeurs de π. On définit alors :

$$\text{SMAST}_1(\pi) = \max_{\pi' \in Succ(\pi)} \text{SMAST}(\pi')$$

Définissons maintenant $\text{SMAST}_2(\pi)$. On dit qu'une paire (π_1, π_2) de positions est une *décomposition* de π si et seulement si (i) $\pi_1 \neq \pi, \pi_2 \neq \pi$ et (ii) pour chaque $i \in [k]$:
– soit $\pi[i] = \perp$, auquel cas $\pi_1[i] = \pi_2[i] = \perp$;
– soit $\pi[i]$ est une feuille x, auquel cas $\{\pi_1[i], \pi_2[i]\} = \{\perp, x\}$;
– soit $\pi[i]$ est un noeud interne u ayant deux fils v, v', auquel cas on a soit $\{\pi_1[i], \pi_2[i]\} = \{\perp, u\}$, soit $\{\pi_1[i], \pi_2[i]\} = \{v, v'\}$.
On note $Dec(\pi)$ l'ensemble des décompositions de π. On définit alors :

$$\text{SMAST}_2(\pi) = \max_{(\pi_1, \pi_2) \in Dec(\pi)} (\text{SMAST}(\pi_1) + \text{SMAST}(\pi_2))$$

Notons que le calcul des valeurs $\text{SMAST}_1(\pi)$ et $\text{SMAST}_2(\pi)$ implique uniquement des valeurs $\text{SMAST}(\pi')$ avec $\pi' <_{\mathcal{T}} \pi$, par l'observation suivante.

Observation 5.37. *(i) si $\pi' \in Succ(\pi)$, alors $\pi' <_{\mathcal{T}} \pi$;*
(ii) si $(\pi_1, \pi_2) \in Dec(\pi)$, alors $\pi_1 <_{\mathcal{T}} \pi$ et $\pi_2 <_{\mathcal{T}} \pi$.

Démonstration. Le point (i) est clair. Pour le point (ii), observons que dans la définition d'une décomposition le point (ii) implique que $\pi_1 \leq_{\mathcal{T}} \pi$, $\pi_2 \leq_{\mathcal{T}} \pi$, et le point (i) implique que ces relations sont strictes. □

On est maintenant en mesure d'énoncer la relation de récurrence pour les positions non terminales.

Lemme 5.38. *Soit π une position non terminale. Alors*

$$\text{SMAST}(\pi) = \max(\text{SMAST}_1(\pi), \text{SMAST}_2(\pi)).$$

Démonstration. On prouve d'abord que $\textsc{Smast}_1(\pi) \leq \textsc{Smast}(\pi)$. Soit $S \in \textsc{Ast}(\pi')$ pour un certain $\pi' \in Succ(\pi)$, tel que $|S|$ soit maximal. Comme $\pi' <_{\mathcal{T}} \pi$ par l'Observation 5.37, on a $S \in \textsc{Smast}(\pi)$ par l'Observation 5.35, et on conclut.

On prouve maintenant que $\textsc{Smast}_2(\pi) \leq \textsc{Smast}(\pi)$. Soit $(\pi_1, \pi_2) \in Dec(\pi)$, et soient S_1, S_2 tels que $S_j \in \textsc{Ast}(\pi_i)$, $|S_j|$ maximal. Si l'un des S_j est vide, disons S_1, alors $\textsc{Smast}(\pi_1) = 0$, et on obtient $\textsc{Smast}_2(\pi) = |S_2| = \textsc{Smast}(\pi_2) \leq \textsc{Smast}(\pi)$ par les Observations 5.35 et 5.37. Supposons maintenant que S_1, S_2 sont non vides. Pour $j \in \{1, 2\}$, puisque $S_j \in \textsc{Ast}(\pi_j)$, il existe des plongements partiels $\phi_{j,i} : S_i \to T_i$ tels que $\phi_{j,i}(r(S_i)) \leq_{T_i} \pi_j[i]$ pour chaque $i \in [k]$. Posons $S = (S_1, S_2)$, et montrons que $S \in \textsc{Smast}(\pi)$. En effet, pour chaque $i \in [k]$ définissons un plongement partiel $\phi_i : S \to T_i$ comme suit. Posons $\phi_i(x) = \phi_{j,i}(x)$ si x est un noeud de S_j, et $\phi_i(x) = \mathsf{lca}_{T_i}(\phi_{1,i}(r(S_1)), \phi_{2,i}(r(S_2)))$ si x est la racine de S. On peut alors montrer que : (i) $L(S_1) \cap L(S_2) = \emptyset$, donc S est bien défini, (ii) ϕ_i est un plongement partiel de S dans T_i, (iii) $\phi_i(r(S)) \leq_{T_i} \pi[i]$. On conclut que $\textsc{Smast}_2(\pi) = |S_1| + |S_2| = |S| \leq \textsc{Smast}(\pi)$.

On prouve finalement que $\textsc{Smast}(\pi) \leq \max(\textsc{Smast}_1(\pi), \textsc{Smast}_2(\pi))$. Soit $S \in \textsc{Ast}(\pi)$ tel que $|S|$ soit maximal. Il existe alors des plongements partiels $\phi_i : S \to T_i$ tel que $\phi_i(r(S)) \leq_{T_i} \pi[i]$, pour chaque $i \in [k]$. Posons $u_i = \phi_i(r(S))$ pour tout i. On distingue deux cas.

Premier cas : il existe $i \in [k]$ tel que $u_i <_{T_i} \pi[i]$. Ce cas se présente en particulier si $|S| \leq 1$. Définissons π' à partir de π en substituant la $i$ème composante par $\mathsf{child}_{T_i}(\pi[i], u_i)$, alors $\pi' \in Succ(\pi)$. On vérifie que $S \in \textsc{Ast}(\pi')$: en effet, ϕ_i est un plongement partiel de S dans T_i tel que $\phi_i(r(S)) \leq_{T_i} \pi'[i]$ pour tout $i \in [k]$. On conclut que $|S| = \textsc{Smast}(\pi) \leq \textsc{Smast}(\pi') \leq \textsc{Smast}_1(\pi)$.

Deuxième cas : pour tout $i \in [k]$, $u_i = \pi[i]$. On a alors nécessairement $|S| \geq 2$, et donc $S = (S_1, S_2)$. Soit r la racine de S, soit r_j la racine de S_j dans S. Pour $j \in \{1, 2\}$, on définit π_j comme suit : étant donné $i \in [k]$, (i) si $\phi_i(r_j) = \pi[i]$, poser $\pi_j[i] = \pi[i]$, (ii) si $\phi_i(r_j) = \perp$, poser $\pi_j[i] = \perp$, (iii) si $\phi_i(r_j) <_{T_i} \pi[i]$, poser $\pi_j[i] = \mathsf{child}_{T_i}(\pi[i], \phi_i(r_j))$. On vérifie alors que $(\pi_1, \pi_2) \in Dec(\pi)$ et $S_j \in \textsc{Ast}(\pi_j)$. On conclut que $|S| = \textsc{Smast}(\pi) = |S_1| \leq \textsc{Smast}(\pi_1) + \textsc{Smast}(\pi_2) \leq \textsc{Smast}_2(\pi)$. □

On dispose enfin de tous les éléments pour établir la Proposition 5.34, à savoir que Smast est résoluble en espace $O((2n)^k)$ et en temps $O((6n)^k)$.

Preuve de la Proposition 5.34. L'algorithme calcule les valeurs $\textsc{Smast}(\pi)$ pour chaque π position dans $\mathcal{T}$, en utilisant les relations de récurrence énoncées dans les Lemmes 5.36 et 5.38. La correction de l'algorithme résulte des lemmes, et la terminaison de l'algorithme est assurée par l'Observation 5.37 et le fait que $<_{\mathcal{T}}$ est un ordre strict sur les positions dans $\mathcal{T}$.

On justifie maintenant la complexité en temps et en espace de l'algorithme.

Pour analyser la complexité en espace, observons que le nombre de positions dans $\mathcal{T}$ est au plus $(2n)^k$: en effet, une composante $\pi[i]$ a au plus $2n$ valeurs possibles (soit $\perp$, soit l'un des noeuds de T_i en nombre $\leq 2n-1$). La complexité en espace de l'algorithme est donc $O((2n)^k)$.

Pour analyser la complexité en temps, affinons l'analyse précédente. Notons N_p le nombre de positions π dont exactement p composantes sont des feuilles. Alors $N_p \leq \binom{k}{p} n^k$: en effet, chaque composante $\pi[i]$ qui est une feuille peut avoir n valeurs, et chaque composante $\pi[i]$ qui n'est pas une feuille peut avoir n valeurs (soit la valeur $\perp$, soit l'un des noeuds internes de T_i en nombre $\leq n-1$). Soit π une position, notons $n_j(\pi)$ le nombre de valeurs $\textsc{Smast}(\pi')$ qui sont examinées lors du calcul de $\textsc{Smast}_j(\pi)$, alors :

- $n_1(\pi) \leq k$, puisque le nombre de successeurs de π est borné par k.
- $n_2(\pi) \leq 2^p 4^{k-p}$, où p est le nombre de composantes feuilles de π. En effet, il existe deux valeurs possibles pour le couple $(\pi_1[i], \pi_2[i])$ si $\pi[i]$ est une feuille, et quatre valeurs possibles si $\pi[i]$ est un noeud interne.

On conclut que le temps total nécessaire au calcul des valeurs $\textsc{Smast}_1(\pi)$ est $O(k(2n)^k)$, tandis que le temps total nécessaire au calcul des valeurs $\textsc{Smast}_2(\pi)$ est $O(\sum_{p=0}^{k} 2^p 4^{k-p} N_p) = O(\sum_{p=0}^{k} 2^p 4^{k-p} \binom{k}{p} n^k) = O((6n)^k)$. □

5.2.3 Résultats négatifs

On présente dans cette section des résultats de difficulté paramétrique pour Smast, pour les paramètres q et k. On transfère en fait à Smast les résultats de difficulté pour Slcs établis en Section 5.1.3, au moyen d'une réduction paramétrée de $\textsc{Slcs}[q,k]$ à $\textsc{Smast}[q,k]$ (Proposition 5.41).

Cette réduction est effectuée en deux étapes. On introduit une variante de Slcs appelée Colored-Slcs. On réduit dans un premier temps $\textsc{Slcs}[q,k]$ à $\textsc{Colored-Slcs}[q,k]$ (Lemme 5.39), et dans un second temps $\textsc{Colored-Slcs}[q,k]$ à $\textsc{Smast}[q,k]$ (Lemme 5.40).

Le problème Colored-Slcs est défini comme suit. Etant donné un ensemble d'étiquettes L partitionné en q ensembles $L_1, ..., L_q$, une *séquence colorée* est une séquence de la forme $a_1...a_q$ avec $a_i \in L_i$. Le problème Colored-Slcs consiste, étant donnés q, k, une collection $\mathcal{C}$ de k séquences sur un ensemble d'étiquettes partitionné en q ensembles, à décider s'il existe une séquence colorée qui soit également une séquence compatible de $\mathcal{C}$ (on parlera de *séquence compatible colorée*).

Lemme 5.39. *Il existe une réduction polynomiale de* $\textsc{Slcs}[q,k]$ *à* $\textsc{Colored-Slcs}[q,k]$ *qui transforme une instance* $(\mathcal{C}, q, k)$ *de* $\textsc{Slcs}[q,k]$ *sur une instance* $(\mathcal{C}', q, 2k)$ *de* $\textsc{Colored-Slcs}[q,k]$.

Démonstration. Soit une instance $I = (\mathcal{C}, q, k)$ de $\textsc{Slcs}[q,k]$, où $\mathcal{C} = \{s_1, ..., s_k\}$ est une collection sur l'ensemble d'étiquettes L. On construit une instance $I' = (\mathcal{C}', q, 2k)$ de $\textsc{Colored-Slcs}[q,k]$ comme suit.

Construction. On définit d'abord l'ensemble d'étiquettes L'. Pour chaque $x \in L$ on crée q étiquettes $x^1, ..., x^q$, on pose $L'^i = \{x^i : x \in L\}$ et $L' = L'^1 \cup ... \cup L'^q$.

On définit maintenant la collection $\mathcal{C}'$. Considérons les morphismes de monoïdes

libres ϕ, ϕ', de L^* dans L'^*, définis comme suit : pour chaque $x \in L$,

$$\begin{cases} \phi(x) & = x^1...x^q \\ \phi'(x) & = x^q...x^1 \end{cases}$$

Pour chaque séquence $s_i \in \mathcal{C}$, définissons $s'_i = \phi(s_i)$ et $s''_i = \phi'(s_i)$. Posons $\mathcal{C}' = \{s'_1, s''_1, ..., s'_k, s''_k\}$. Alors $\mathcal{C}'$ est une collection de $2k$ séquences sur l'ensemble d'étiquettes L'.

Correction. La réduction est clairement calculable en temps polynomial. Pour établir que la réduction est correcte, on montre que :

Fait. $\mathcal{C}$ a une séquence compatible de longueur q si et seulement si $\mathcal{C}'$ a une séquence compatible colorée.

Preuve. ($\Rightarrow$) : supposons que s est une séquence compatible de $\mathcal{C}$, de longueur q. Alors $s = z_1...z_q$. Soit $s' = z_1^1...z_q^q$, montrons que s' est une séquence compatible colorée de $\mathcal{C}'$. Clairement, s' est une séquence colorée. Pour montrer que s' est une séquence compatible de $\mathcal{C}'$, on montre que pour tout $p \in [k]$:

- s' s'accorde avec s'_p : considérons $x, y \in L(s') \cap L(s'_p)$ tels que $x <_{s'} y$. Alors $x = z_i^i, y = z_j^j$ avec $i < j$. Comme $z_i <_s z_j$ et comme s s'accorde avec s_p, il résulte que $z_i <_{s_p} z_j$, et donc $z_i^i <_{s'_p} z_j^j$. On obtient bien $x <_{s_p} y$ comme attendu.
- s' s'accorde avec s''_p : le raisonnement est similaire.

($\Leftarrow$) : supposons que s est une séquence compatible colorée de $\mathcal{C}'$. Alors $s' = y_1...y_q$ avec $y_i \in L'^i$ pour tout i. Comme $y_i \in L'^i$, il existe donc $z_i \in L$ tel que $y_i = z_i^i$.

Observons que les étiquettes $z_1, ..., z_q$ sont distinctes : si $z_j, z_{j'}$ étaient égaux (à une même étiquette x), avec $j < j'$, alors en considérant une séquence s_i telle que $x \in L(s_i)$, on obtiendrait que $z_j^j <_{s'_i} z_{j'}^{j'}$ mais $z_{j'}^{j'} <_{s''_i} z_j^j$, impossible.

Définissons alors $s = z_1...z_q$, et montrons que s est une séquence compatible pour $\mathcal{C}$. Soit $p \in [k]$, on montre que s s'accorde avec s_p. Considérons $x, y \in L(s) \cap L(s_p)$ tels que $x <_s y$. Alors $x = z_i, y = z_j$ avec $i < j$. Comme $z_i^i <_{s'} z_j^j$ et comme s' s'accorde avec s'_p, on obtient que $z_i^i <_{s'_p} z_j^j$, et comme z_i, z_j sont distincts cela implique que $z_i <_{s_p} z_j$. On obtient bien $x <_{s_p} y$ comme attendu. □

Lemme 5.40. *Il existe une réduction polynomiale de* Colored-Slcs$[q, k]$ *à* Smast$[q, k]$ *qui envoie une instance* $(\mathcal{C}, q, k)$ *de* Colored-Slcs$[q, k]$ *sur une instance* $(\mathcal{T}, 2q + 1, k + 2)$ *de* Smast$[q, k]$.

Démonstration. Soit une instance $I = (\mathcal{C}, q, k)$ de Colored-Slcs$[q, k]$, où $\mathcal{C} = \{s_1, ..., s_k\}$ est une collection sur l'ensemble d'étiquettes L, partitionné en q ensembles $L_1, ..., L_q$. On construit une instance $I' = (\mathcal{T}, 2q+1, k+2)$ de Smast$[q, k]$ comme suit.

Construction. On définit d'abord l'ensemble d'étiquettes L'. On crée de nouvelles étiquettes $z_0, z_1, ..., z_q$, et pour chaque $i \in [q]$ on pose $L'_i = L_i \cup \{z_i\}$. On pose $L' = \{z_0\} \cup L'_1 \cup ... \cup L'_q$.

On définit maintenant la collection : $\mathcal{T} := \{S, S'\} \cup \{T_1, ..., T_k\}$. Les p-arbres de la collection sont construits comme suit. Pour chaque $i \in [q]$, soit $<_i$ un ordre total sur L'_i, et soit $R_i = rake(L'_i, <_i)$ et $R'_i = rake(L'_i, >_i)$. On pose alors $S := rake(z_0, R_1, ..., R_q)$ et $S' := rake(z_0, R'_1, ..., R'_q)$. Pour chaque séquence $s_i \in \mathcal{C}$, on pose $T_i := rake(z_0, s_i[1], ..., s_i[|s_i|])$. Alors $\mathcal{T}'$ est une collection de $k+2$ p-arbres sur l'ensemble d'étiquettes L'.

Correction. La réduction est clairement calculable en temps polynomial. Sa correction résulte du point suivant.

Fait. $\mathcal{C}$ a une séquence compatible colorée si et seulement si $\mathcal{T}$ a un superarbre d'accord de taille $2q+1$.

Preuve. ($\Rightarrow$) : supposons que s est une séquence compatible colorée pour $\mathcal{C}$. Alors $s = y_1...y_q$, avec $y_i \in L_i$. Soit $T = rake(z_0, (z_1, y_1), ..., (z_q, y_q))$, alors $|T| = 2q+1$. Montrons que T est un superarbre d'accord pour $\mathcal{T}$. En effet :

- clairement, T s'accorde avec S et avec S', puisque $R_i|\{z_i, y_i\} = R'_i|\{z_i, y_i\} = (z_i, y_i)$ pour chaque $i \in [q]$.
- de plus, T s'accorde avec chaque p-arbre T_i. En effet, si $s|L(s_i) = s_i|L(s) = y_{i_1}...y_{i_m}$ avec $i_1 < ... < i_m$, alors $T|L(T_i) = T_i|L(T) = rake(z_0, y_{i_1}, ..., y_{i_m})$.

($\Leftarrow$) : supposons que T est un superarbre d'accord pour $\mathcal{T}$ de taille $2q+1$. Commençons par observer que pour tout $i \in [q]$, $|L(T) \cap L'_i| \leq 2$: sinon, T ne pourrait pas s'accorder à la fois avec S et S'. Comme $|T| = 2q+1$, il résulte que $|L(T) \cap L'_i| = 2$ pour chaque $i \in [q]$, et que $z_0 \in L(T)$. Maintenant, pour chaque $i \in [q]$ choisissons $y_i \in L(T) \cap L'_i$ distinct de z_i, et posons $s = y_1...y_q$. Montrons que s est une séquence compatible colorée pour $\mathcal{C}$. Clairement, s est une séquence colorée. Soit $i \in [k]$, montrons que s s'accorde avec s_i. Comme T s'accorde avec T_i, on a $T|L(T_i) = T_i|L(T) = rake(z_0, y_{i_1}, ..., y_{i_m})$ avec $i_1 < ... < i_m$. Par définition de T_i et de s, il résulte que $s|L(s_i) = s_i|L(s) = y_{i_1}...y_{i_m}$, et donc s s'accorde avec s_i. □

En combinant les lemmes 5.39 et 5.40., on obtient :

Proposition 5.41. *Il existe une réduction polynomiale de* SLCS$[q, k]$ *à* SMAST$[q, k]$ *qui envoie une instance* $(\mathcal{C}, q, k)$ *de* SLCS$[q, k]$ *sur une instance* $(\mathcal{T}, 2q+1, 2k+2)$ *de* SMAST$[q, k]$.

La Proposition 5.41 transfère à SMAST les résultats obtenus pour SLCS en Section 5.1 :

Proposition 5.42. *On a les résultats de difficulté suivants pour* SMAST *:*

- W[1]*-difficulté pour les paramètres* q *et* (q, k) *;*
- WNL*-difficulté pour le paramètre* k*.*

Démonstration. Conséquence des résultats pour SLCS fournis par les Propositions 5.13, 5.15, et de la réduction fournie par la Proposition 5.41. □

5.2.4 Remarques

Les deux variantes suivantes du problème SMAST peuvent se résoudre en adaptant l'algorithme de programmation dynamique de la Proposition 5.34 :

1. une variante pondérée WEIGHTED-SMAST, dans laquelle on dispose d'une collection $\mathcal{T}$ sur L, d'une fonction de coût $w : L \to \mathbb{R}^+$, et où l'objectif est de trouver un ensemble $L' \subseteq L$ de coût maximum tel que $\mathcal{T}|L'$ est compatible. Ce problème peut se résoudre en adaptant l'algorithme de la Proposition 5.34 de la façon suivante : on remplace l'équation du Lemme 5.36 par $\text{SMAST}(\pi) = w(P(\pi))$, où par convention $w(X) = \sum_{x \in X} w(x)$.
2. une variante CONSTRAINED-SMAST, dans laquelle on dispose d'une collection $\mathcal{T}$ sur L, d'un ensemble d'étiquettes $L_0 \subseteq L$, et où l'objectif est de trouver un ensemble $L' \subseteq L$ *contenant* L_0 et de cardinal maximum tel que $\mathcal{T}|L'$ est compatible. Etant donnée π position dans $\mathcal{T}$, posons $C(\pi) = L_0 \cap A(\pi)$, et définissons $\text{SMAST}(\pi)$ comme la taille d'un plus grand ensemble $L' \subseteq A(\pi)$ tel que $C(\pi) \subseteq L'$ et $\mathcal{C}|L'$ est compatible. On adapte alors les relations de récurrence de la Proposition 5.34 comme suit :
   – on adapte le Lemme 5.36 : soit π une position terminale, alors $\text{SMAST}(\pi) = |P(\pi)|$ si $C(\pi) \subseteq P(\pi)$, $\text{SMAST}(\pi) = -\infty$ sinon.
   – on définit $Succ(\pi)$ comme l'ensemble des positions π' successeurs de π *telles que* $C(\pi') = C(\pi)$, et on pose

$$\text{SMAST}_1(\pi) = \max_{\pi' \in Succ(\pi)} \text{SMAST}(\pi')$$

   – on définit $Dec(\pi)$ comme l'ensemble des paires (π_1, π_2) décompositions de π *telles que* $C(\pi_1) \cup C(\pi_2) = C(\pi)$, et on pose

$$\text{SMAST}_2(\pi) = \max_{(\pi_1,\pi_2) \in Dec(\pi)} (\text{SMAST}(\pi_1) + \text{SMAST}(\pi_2))$$

   – on calcule $\text{SMAST}(\pi) = \max(\text{SMAST}_1(\pi), \text{SMAST}_2(\pi))$.

Pour finir, on évoque deux questions ouvertes concernant la complexité paramétrique et l'approximabilité de CSLCS. Ces questions sont analogues à celles soulevées pour SLCS en Section 5.1.

1. Quel est le ratio d'approximabilité de CSLCS en fonction du paramètre k ? Un algorithme d'approximation à un facteur $\Omega(\log k)$ pourrait avoir un intérêt pratique.
2. Le problème est-il soluble en temps $p^{O(k)}n^c$? Un tel algorithme serait préférable en pratique à l'algorithme en $O((2k)^p \times kn^2)$ de la Proposition 5.31, lorsque k est faible.

5.3 Bibliographie

[1] Aho (A. V.), Sagiv (Y.), Szymanski (T. G.) et Ullman (J. D.). – Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, vol. 10, n° 3, 1981, pp. 405–421.

[2] Bodlaender (H.L.), Downey (R.G.), Fellows (M.R.), Hallett (M.T.) et Wareham (H.T.). – Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, vol. 11, n° 1, 1995, pp. 49–57.

[3] Bodlaender (H.L.), Downey (R.G.), Fellows (M.R.) et Wareham (H.T.). – The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, vol. 147, n° 1–2, 1994, pp. 31–54.

[4] Bodlaender (H.L.) et Engelfriet (J.). – Domino treewidth. *Journal of Algorithms*, vol. 24, 1997, pp. 94–123.

[5] Bodlaender (H.L.), Fellows (M.R.) et Hallett (M.T.). – Beyond NP-completeness for problems of bounded width : Hardness for the W hierarchy (extended abstract). *In : Proceedings of STOC 1994*. pp. 449–458. – ACM.

[6] Bodlaender (H.L.), Fellows (M.R.) et Hallett (M.T.). – The hardness of perfect phylogeny, feasible register assignment and other problems on thin colored graphs. *Theoretical Computer Science*, vol. 244, n° 1, 2000, pp. 167–188.

[7] Chen (J.), Liu (Y.), Lu (S.), O'Sullivan (B.) et Razgon (I.). – A fixed-parameter algorithm for the directed feedback vertex set problem. *In : Proceedings of STOC 2008*, pp. 177–186.

[8] Even (G.), Naor (J.), Schieber (B.) et Sudan (M.). – Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, vol. 20, 1998, pp. 151–174.

[9] Fellows (M.R.), Hallett (M.T.) et Stege (U.). – Analogs & duals of the MAST problem for sequences & trees. *Journal of Algorithms*, vol. 49, n° 1, 2003, pp. 192–216.

[10] Gottlob (G.), Grohe (M.), Musliu (N.), Samer (M.) et Scarcello (F.). – Hypertree decompositions : Structure, algorithms and applications. *In : Proceedings of WG 2005*, pp. 1–15.

[11] Halldórsson (M.M.). – *Approximation via Partitioning*. – Rapport technique n° IS-RR-95-0003F, School of Information Science, Japan Advanced Institute of Science and Technology, Hokuriku, 1995.

[12] Hallett (M.T.). – *An integrated complexity analysis of problems from computational biology*. – Thèse de doctorat, Department of Computer Science, University of Victoria, Victoria, B.C., Canada, 1996.

[13] Henzinger (M.R.), King (V.) et Warnow (T.). – Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology. *Algorithmica*, vol. 24, n° 1, 1999, pp. 1–13.

[14] Kaplan (H.) et Shamir (R.). – *Pathwidth, bandwidth and completion problems to proper interval graphs with small cliques*. – Rapport technique n° 258/93, Tel Aviv University, 1993.

[15] Pietrzak (K.). – On the Parameterized Complexity of the fixed alphabet Shortest Common Supersequence and Longest Common Subsequence Problems. *Journal of Computer and System Sciences*, vol. 67, n° 4, 2003, pp. 757–771.

[16] Seymour (P.D.). – Packing directed circuits fractionally. *Combinatorica*, vol. 15, 1995, pp. 281–288.

[17] Slivkins (A.). – Parameterized Tractability of Edge-Disjoint Paths on Directed Acyclic Graphs. *In : Proceedings of ESA 2003*, pp. 482–493.

[18] Wareham (H.T.). – The parameterized complexity of intersection and composition operations on sets of finite-state automata. *In : Proceedings of ICIAA 2000*, pp. 302–310.

Chapitre 6

Collections de triplets enracinés

Un triplet enraciné est un arbre binaire enraciné à trois feuilles. Une collection de triplets est dite compatible s'il existe un arbre contenant tous les triplets de la collection comme sous-arbre. En présence d'une collection non compatible, on peut chercher à la rendre compatible de trois manières différentes : en conservant un nombre maximum de triplets, en supprimant un nombre minimum de triplets, en supprimant un nombre minimum d'étiquettes. Ces trois démarches conduisent à formuler respectivement les problèmes Mtc, Mti et Mli étudiés dans ce chapitre. On considère notamment ces problèmes restreints aux collections complètes, c'est-à-dire aux collections où chaque ensemble de trois étiquettes est présent dans au moins un triplet. Il existe des situations où l'on dispose d'un triplet pour chaque ensemble de trois étiquettes, par exemple lorsque chaque triplet a été obtenu par la méthode du maximum de vraisemblance [7] ou par des expériences d'hybridation du type Sibley-Ahlquist [14]. On dispose alors d'une collection complète, ce qui présente l'intérêt d'abaisser la complexité des problèmes Mti et Mli dans ce cas. Cette réduction de la complexité repose sur le fait que dans le cas d'une collection complète, une incompatibilité peut être circonscrite à un ensemble de quatre étiquettes.

6.1 Définitions et propriétés

Dans cette section, on introduit la notion de collection de triplets enracinés, la propriété de compatibilité, et on décrit différentes caractérisations de cette propriété.

6.1.1 Définitions

Une *collection de triplets enracinés* sur L est une collection $\mathcal{R} = \{t_1, ..., t_k\}$, où chaque t_i est un triplet enraciné sur L. $\mathcal{R}$ est *complète* (resp. *simple*) si et

seulement si chaque ensemble de trois étiquettes est présent dans au moins (resp. au plus) un triplet $\mathcal{R}$ est *minimalement complète* si et seulement si elle est à la fois simple et complète.

$\mathcal{R}$ est *compatible* si et seulement si il existe un arbre T sur L tel que $\mathcal{R} \subseteq rt(T)$; on dit que T est une *extension* de $\mathcal{R}$. Soit $\mathcal{R}$ une collection de triplets minimalement complète, on dit que $\mathcal{R}$ est *arborée* si et seulement si il existe un arbre T sur L tel que $\mathcal{R} = rt(T)$; on dit que $\mathcal{R}$ *représente* T.

6.1.2 Compatibilité : cas général

La compatibilité d'une collection de triplets est décidable en temps polynomial, par un algorithme dû à [1]. Cet algorithme suit une approche récursive descendante ; à chaque appel récursif, il construit un graphe auxiliaire appelé *graphe de Aho*.

Définition 6.1. *Soit $\mathcal{R}$ une collection de triplets sur L, et soit $L' \subseteq L$. Le graphe de Aho de $\mathcal{R}$ par rapport à L' est le graphe $G(\mathcal{R}, L')$ tel que :*
- *son ensemble de sommets est L' ;*
- *il comporte une arête xy si et seulement si il existe $z \in L'$ tel que $xy|z \in \mathcal{R}$.*

Cette définition est illustrée ci-dessous :

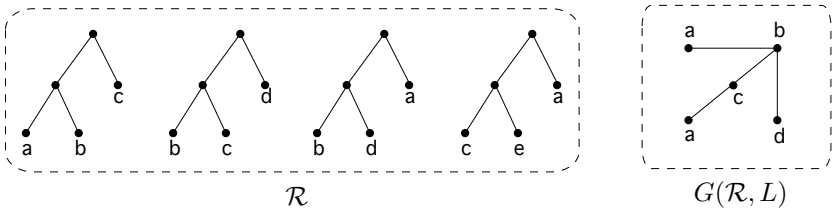


FIG. 6.1 – Une collection de triplets $\mathcal{R}$ sur $L = \{a, b, c, d, e\}$, et le graphe $G(\mathcal{R}, L)$

Soit $G = (V, E)$ un graphe non orienté. Une *partition déconnectée* de G est une partition de V en deux sous-ensembles V_1, V_2, tels que G ne contienne aucune arête joignant V_1 et V_2.

L'algorithme procède de la façon suivante. Il utilise une approche récursive descendante. A un pas de la récursion, il examine un sous-ensemble $L' \subseteq L$, et construit le graphe $G(\mathcal{R}, L')$. Si le graphe est connexe, alors il conclut que $\mathcal{R}$ est incompatible. Si le graphe est non connexe, les composantes connexes de $G(\mathcal{R}, L')$ fournissent les clades supérieurs du superarbre, et les clades restant sont obtenus par appels récursifs sur les composantes connexes.

Lemme 6.2. *Soit $\mathcal{R}$ une collection sur L, et soit $L' \subseteq L$.*
- *supposons que $\mathcal{R}$ est compatible, et soit $T = (T_1, T_2)$ une extension de $\mathcal{R}|L'$. Alors $L(T_1), L(T_2)$ est une partition déconnectée de $G(\mathcal{R}, L')$, et T_i est une extension de $\mathcal{R}|L'_i$.*
- *Soit L'_1, L'_2 une partition déconnectée de $G(\mathcal{R}, L')$, et soit T_i une extension de $\mathcal{R}|L'_i$. Alors $T = (T_1, T_2)$ est une extension de $\mathcal{R}$.*

Démonstration. ($\Rightarrow$) : supposons que T est une extension de $\mathcal{R}|L'$. Considérons L'_1, L'_2 définis par $L'_i = L(T_i)$. Alors T_i est une extension de $\mathcal{R}|L'_i$. De plus, $\mathcal{R}|L'$ ne peut pas contenir de triplet $xy|z$ avec $x \in L'_1, y \in L'_2$, et donc L'_1, L'_2 est une partition déconenctée de $G(\mathcal{R}, L')$.

($\Leftarrow$) : supposons qu'il existe L'_1, L'_2 vérifiant les propriétés annoncées. Montrons que T est une extension de $\mathcal{R}|L'$. Soit $xy|z \in \mathcal{R}|L'$, montrons que $xy|z \in rt(T)$. Si x, y, z sont tous contenus dans un ensemble L_i, cela résulte du fait que T_i est une extension de $\mathcal{R}|L'_i$. Si ce n'est pas le cas, comme L'_1, L'_2 sont déconnectés dans $G(\mathcal{R}, L')$, on doit avoir $x, y \in L'_1, z \in L'_2$ ou le cas symétrique, et l'on vérifie que $xy|z \in rt(T)$. □

Proposition 6.3. *[1] Soit $\mathcal{R}$ une collection sur L. $\mathcal{R}$ est compatible si et seulement si pour tout $L' \subseteq L$ de cardinal ≥ 3, le graphe $G(\mathcal{R}, L')$ est non connexe.*

Démonstration. ($\Rightarrow$) : soit $L' \subseteq L$, soit T une extension de $\mathcal{R}|L'$. Comme $|L'| \geq 3$, on a $T = (T_1, T_2)$. Par le Lemme 6.2, $L(T_1), L(T_2)$ est une partition déconnectée de $G(\mathcal{R}, L')$.

($\Leftarrow$) : on montre que $\mathcal{R}|L'$ est compatible par induction sur $|L'|$. Le résultat est clair si $|L'| \leq 2$. Supposons maintenant que $|L'| \geq 3$. Comme $G(\mathcal{R}, L')$ est non connexe, il admet une partition déconnectée L'_1, L'_2. Alors $\mathcal{R}|L'_i$ est compatible par hypothèse d'induction, donc admet une extension T_i. Par le Lemme 6.2, (T_1, T_2) est une extension de $\mathcal{R}|L'$. Donc $\mathcal{R}|L'$ est compatible. □

La Proposition précédente fournit un algorithme polynomial pour décider la compatibilité.

Proposition 6.4. *[1] Soit $\mathcal{R}$ une collection de m triplets sur un ensemble L de n étiquettes. On peut décider en temps $O(n(n+m))$ si $\mathcal{R}$ est compatible.*

Démonstration. On utilise l'algorithme récursif IsCompatible($\mathcal{R}, L'$) qui décide si $\mathcal{R}|L'$ est compatible. L'algorithme répond positivement si $|L'| \leq 2$; sinon,

- il construit le graphe $G(\mathcal{R}, L')$;
- si le graphe est connexe : il répond négativement ;
- sinon il trouve une partition déconnectée L_1, L_2 du graphe, et appelle récursivement IsCompatible($\mathcal{R}, L_i$) pour $i \in \{1, 2\}$.

La compatibilité de $\mathcal{R}$ est alors obtenue par appel de IsCompatible($\mathcal{R}, L$).

La correction de l'algorithme résulte de la Proposition 6.3. Justifions que son temps d'exécution est en $O(n(n+m))$: en effet, l'algorithme fait au plus n appels récursifs à IsCompatible, et à chaque appel la construction de $G(\mathcal{R}, L')$ se fait en temps $O(n+m)$, d'où la complexité annoncée. □

6.1.3 Compatibilité : cas des collections complètes

Dans le cas des collections complètes, la compatibilité est équivalente à la propriété suivante. Disons qu'une collection complète $\mathcal{R}$ sur L est *arborée* si et seulement si il existe un arbre T sur L tel que $\mathcal{R} = rt(T)$.

Dans cette section, on montre que pour une collection complète, le fait d'être arborée est une propriété « locale ». Un *conflit* de $\mathcal{R}$ est un ensemble $C \subseteq L$ t.q. $\mathcal{R}|C$ est incompatible. Un *t-conflit* de $\mathcal{R}$ est un ensemble $\mathcal{S} \subseteq \mathcal{R}$ incompatible.

Proposition 6.5. *Soit $\mathcal{R}$ une collection complète non arborée. Alors : (i) $\mathcal{R}$ contient un conflit de cardinal ≤ 4 ; (ii) $\mathcal{R}$ contient un t-conflit de cardinal ≤ 3.*

Avant d'établir ce résultat, observons qu'il permet de donner une caractérisation simple des collections arborées. Une contradiction directe de $\mathcal{R}$ est un ensemble de triplets $ab|c, bc|a \in \mathcal{R}$. On montre :

Proposition 6.6. *Soit $\mathcal{R}$ une collection complète. $\mathcal{R}$ est arborée si et seulement si $\mathcal{R}$ ne contient pas de contradictions directes, et vérifie :*

$$(P) \quad \forall a, b, c, d \in L(\mathcal{R}), ab|c \in \mathcal{R} \wedge bc|d \in \mathcal{R} \Rightarrow ab|d \in \mathcal{R} \wedge ac|d \in \mathcal{R}$$

Démonstration. ($\Rightarrow$) : supposons que $\mathcal{R}$ représente T. Clairement, $\mathcal{R}$ ne contient pas de contradiction directe. Soient $a, b, c, d \in L(\mathcal{R})$ tels que $ab|c \in \mathcal{R}$ et $bc|d \in \mathcal{R}$. Comme $(((a, b), c), d)$ est l'unique arbre sur 4 étiquettes contenant les triplets $ab|c$ et $bc|d$, il en résulte que $T|\{a, b, c, d\} = (((a, b), c), d)$, et donc $ab|d \in \mathcal{R}$ et $ac|d \in \mathcal{R}$.

($\Leftarrow$) : par contraposée. Supposons que $\mathcal{R}$ est non arborée et ne contient pas de contradiction directe, on montre que (P) n'est pas vérifiée. Comme $\mathcal{R}$ est non arborée, par la Proposition 6.5 $\mathcal{R}$ admet un conflit C de cardinal ≤ 4. $|C| = 3$ est impossible, puisque $\mathcal{R}$ ne contient pas de contradiction directe. Donc $|C| = 4$, et une analyse de cas montre qu'on a l'alternative suivante :

- soit $\mathcal{R}$ contient deux triplets $ab|c, bc|d$ avec $C = \{a, b, c, d\}$. Si $\mathcal{R}$ contient les triplets $ab|d, ac|d$, alors $\mathcal{R}|C$ représente l'arbre $(((a, b), c), d)$, contredisant l'hypothèse que C est un conflit. On a donc $ab|d \notin \mathcal{R}$ ou $ac|d \notin \mathcal{R}$.
- soit $\mathcal{R}$ contient les triplets $ab|c, cd|b, ab|d, cd|a$ avec $C = \{a, b, c, d\}$: alors $\mathcal{R}|C$ représente l'arbre $((a, b), (c, d))$, contredisant l'hypothèse que C est un conflit.

□

Etablissons maintenant la Proposition 6.5. On va donner deux preuves.

La première preuve repose sur un algorithme de reconnaissance des collections arborées, utilisant une approche itérative : il examine les étiquettes une à une, et maintient l'arbre T représenté par la restriction de $\mathcal{R}$ à l'ensemble d'étiquettes déjà examinées. Lors de l'examen d'une nouvelle étiquette x, il cherche à déterminer où insérer x dans T. Pour ce faire, il vérifie que les différentes paires d'étiquettes ℓ, ℓ' de T indiquent le même point d'insertion pour x, en examinant les triplets impliquant ℓ, ℓ', x. Si c'est le cas, il met à jour T en insérant x à l'endroit indiqué. Sinon, il identifie un conflit de taille ≤ 4.

Lemme 6.7. *Il existe un algorithme* Insert-Label-Or-Find-Conflict$(\mathcal{R}, X, x, T)$ *qui prend en argument une collection complète $\mathcal{R}$, un ensemble $X \subseteq L(\mathcal{R})$, un élément $x \in L(\mathcal{R})\backslash X$ et un arbre T tq $\mathcal{R}|X$ représente T, et décide en temps $O(n^2)$ si $\mathcal{R}' = \mathcal{R}|(X \cup \{x\})$ est arborée. De plus, l'algorithme renvoie l'arbre T'*

représenté par $\mathcal{R}'$ en cas de réponse positive, ou renvoie un conflit C entre $\mathcal{R}'$ de cardinal ≤ 4 en cas de réponse négative.

Démonstration. Lors d'une première étape, l'algorithme vérifie si $\mathcal{R}$ contient deux triplets distincts sur le même ensemble de trois étiquettes. Dans ce cas, ces étiquettes forment un conflit de cardinal 3, qui est alors renvoyé par l'algorithme.

S'il n'a trouvé aucun conflit de la sorte, l'algorithme passe à une deuxième étape, durant laquelle il détermine, pour chaque noeud interne u de T, le sous-arbre dans lequel u accepterait d'insérer x : son sous-arbre gauche (L), son sous-arbre droit (R), ou le sous-arbre extérieur à u (A), formé de l'ensemble des noeuds qui ne sont pas descendants de u. Dans ce but, l'algorithme vérifie que les triplets x, ℓ, ℓ', avec ℓ, ℓ' étiquettes descendant de u dans T, indiquent toutes le même sous-arbre. Plus formellement, soit v le fils gauche de u, et v' son fils droit (on considère T comme un arbre ordonné pour les besoins de l'algorithme). Une *u-fourche* est une paire d'étiquettes $\{\ell, \ell'\}$ avec $\ell \in L(v), \ell' \in L(v')$. Chaque u-fourche $\{\ell, \ell'\}$ exprime une opinion $o_{\ell,\ell'}$ sur la position de x par rapport à u, déterminée comme suit : si $\ell x|\ell' \in \mathcal{R}$ alors $o_{\ell,\ell'}$ est mis à L, si $\ell' x|\ell \in \mathcal{R}$ alors $o_{\ell,\ell'}$ est mis à R, sinon $\ell\ell'|x \in \mathcal{R}$ et $o_{\ell,\ell'}$ est mis à A. L'algorithme considère chaque noeud u successivement, et calcule les opinions $o_{\ell,\ell'}$ des u-fourches $\{\ell, \ell'\}$. Si deux u-fourches indiquent un sous-arbre différent, alors l'algorithme identifie un conflit : on a alors ℓ, ℓ_1, ℓ_2 tel que $o_{\ell,\ell_1} \neq o_{\ell,\ell_2}$ (ou $o_{\ell_1,\ell} \neq o_{\ell_2,\ell}$), auquel cas $C = \{x, \ell_1, \ell_2, \ell\}$ est un conflit, qui est alors retourné par l'algorithme. Dans le cas contraire, toutes les u-fourches indiquent une même opinion, qui est alors définie comme l'opinion de u notée o_u.

Lors d'une troisième étape, l'algorithme vérifie que les opinions des différents noeuds u de T indiquent une même position où insérer x dans T. Les opinions sont compatibles si et seulement si pour chaque arête u, v de T avec u père de v, on a : (i) si v est le fils gauche de u, alors $o_u = R \Rightarrow o_v = A$, (ii) si v est le fils droit de u, alors $o_u = L \Rightarrow o_v = A$, (iii) $o_u = A \Rightarrow o_v = A$. Si une paire de noeuds u, v ne vérifie pas ces conditions, alors en considérant $\{\ell, \ell'\}$ v-fourche et $\{\ell, \ell''\}$ u-fourche, on obtient un conflit $C = \{x, \ell, \ell', \ell''\}$. Sinon, considérons l'ensemble des noeuds u tels que $o_u \neq A$, ils forment un chemin dans T partant de la racine et finissant en un noeud v. Alors $\mathcal{R}|(X \cup \{x\})$ est arborée, et représente l'arbre obtenu à partir de T en insérant x au-dessus de v. Cet arbre est alors renvoyé par l'algorithme.

Il est facile de voir que le temps d'exécution de l'algorithme est $O(n^2)$. □

Proposition 6.8. *Il existe un algorithme* Find-Tree-Or-Conflict($\mathcal{R}$) *qui prend en argument une collection complète $\mathcal{R}$, et en temps $O(n^3)$ décide si $\mathcal{R}$ est arborée, renvoie l'arbre T représenté par $\mathcal{R}$ en cas de réponse positive, ou renvoie un conflit C entre $\mathcal{R}$ de cardinal ≤ 4 en cas de réponse négative.*

Démonstration. On utilise la procédure Insert-Label-Or-Find-Conflict comme suit. On insère itérativement chaque étiquette, en démarrant avec un arbre vide, jusqu'à ce que : (i) soit chaque étiquette a été insérée, auquel cas $\mathcal{R}$ est arborée et on a obtenu l'arbre T représenté par $\mathcal{R}$, qu'on renvoie alors, (ii) soit un conflit de cardinal ≤ 4 a été trouvé, qu'on renvoie alors. □

Clairement, ce résultat établit la Proposition 6.5.

On présente maintenant une preuve alternative de la Proposition 6.5 utilisant le graphe de Aho.

Démonstration. Comme $\mathcal{R}$ est non arborée, $\mathcal{R}$ est incompatible, et par la Proposition 6.3 il existe $C \subseteq L$ tq $G(\mathcal{R}, C)$ est connexe. Choisissons un tel ensemble C minimal pour l'inclusion, et soit $G = G(\mathcal{R}, C)$. Alors C est non vide. Considérons $z \in C$ et $G' = G(\mathcal{R}, C\backslash\{z\})$. Alors G' est non connexe, par définition de C, et a donc une partition déconnectée C_1, C_2. On a alors l'alternative suivante.

Premier cas : pour tout $a \in C_1, b \in C_2$, G contient l'arête ab (et donc $ab|z \in \mathcal{R}$). Comme G est connexe, G contient une arête za induite par un élément b, et on peut supposer $a \in C_1$. Si $b \in C_2$, on a alors $ab|z \in \mathcal{R}, az|b \in \mathcal{R}$, et on conclut que $C' = \{a, b, z\}$ est un conflit dans $\mathcal{R}$. Si $b \in C_1$, considérons $c \in C_2$ quelconque, on a alors $ac|z \in \mathcal{R}, bc|z \in \mathcal{R}, az|b \in \mathcal{R}$. On conclut que $C' = \{a, b, c, z\}$ est un conflit dans $\mathcal{R}$, en vérifiant que $G(\mathcal{R}, C')$ est connexe.

Deuxième cas : il existe $a \in C_1, b, c \in C_2$ tq G contienne l'arête ab mais pas l'arête ac. On a alors : $ab|z \in \mathcal{R}, bc|a \in \mathcal{R}$ et ($az|c \in \mathcal{R}$ ou $cz|a \in \mathcal{R}$). On conclut que $C' = \{a, b, c, z\}$ est un conflit dans $\mathcal{R}$, en vérifiant que $G(\mathcal{R}, C')$ est connexe.

Troisième cas : G ne contient aucune arête joignant C_1 à C_2. Alors pour tout $a \in C_1, b \in C_2$, on a : $az|b \in \mathcal{R}$ ou $bz|a \in \mathcal{R}$. Comme G est connexe, G contient des arêtes az, bz avec $a \in C_1, b \in C_2$. On a alors $u, v \in C\backslash\{z\}$ tq $az|u \in \mathcal{R}, bz|v \in \mathcal{R}$. Supposons dans un premier temps que $az|b \in \mathcal{R}$: on a alors $az|b \in \mathcal{R}, bz|v \in \mathcal{R}$ et ($av|b \in \mathcal{R}$ ou $bv|a \in \mathcal{R}$), donc $C' = \{a, b, z, v\}$ est un conflit dans $\mathcal{R}$. Supposons maintenant que $bz|a \in \mathcal{R}$: alors $bz|a \in \mathcal{R}, az|u \in \mathcal{R}$ et ($au|b \in \mathcal{R}$ ou $bu|a \in \mathcal{R}$), donc $C' = \{a, b, z, u\}$ est un conflit dans $\mathcal{R}$. □

6.2 Problème Mtc

Dans cette section, on considère le problème consistant à maximiser le nombre de triplets consistants d'une collection donnée en entrée (Mtc). On définit d'abord le problème en Section 6.2.1. On présente ensuite un algorithme de 3-approximation en Section 6.2.2. En section 6.2.3, on montre que pour une collection de triplets aléatoires environ 1/3 des triplets est satisfait, et on donne une construction déterministe d'une collection de triplets possédant cette propriété. On utilise cette construction en Section 6.2.4 pour établir que Mtc reste NP-dur si l'instance est minimalement complète. On termine par des remarques conclusives en Section 6.3.4.

6.2.1 Définition du problème

Soit $\mathcal{R}$ une collection de triplets sur L. Etant donné un arbre binaire T sur L, posons $C(\mathcal{R}, T) = \mathcal{R} \cap rt(T)$ et $n(\mathcal{R}, T) = |C(\mathcal{R}, T)|$.

Le problème Maximum Triplet Consistency (Mtc) consiste à chercher une collection compatible $\mathcal{S} \subseteq \mathcal{R}$ de cardinal maximum. Cela revient à chercher

un arbre T sur L tel que $n(\mathcal{R}, T)$ soit maximum. On note MTCC la restriction du problème aux collections minimalement complètes.

6.2.2 Algorithmes

Soit $\mathcal{R}$ une collection de triplets. Notons qu'il existe toujours un arbre qui satisfait un tiers des triplets de $\mathcal{R}$. En effet, considérons le processus qui renvoie un arbre aléatoire T sur L, alors

$$\mathbb{E}[n(\mathcal{R}, T)] = \sum_{t \in \mathcal{R}} \mathbb{P}[t \in rt(T)] = \frac{|\mathcal{R}|}{3}$$

et on conclut en considérant un arbre T tel que $n(\mathcal{R}, T) \geq \mathbb{E}[n(\mathcal{R}, T)]$.

La remarque précédente est un argument probabiliste qui pourrait donner lieu à un algorithme randomisé de $3+\epsilon$-approximation. Dans la suite, on présente un algorithme déterministe de 3-approximation pour MTC.

Proposition 6.9. MTC *est 3-approximable par un algorithme polynomial.*

Démonstration. Soit $\mathcal{R}$ une collection de triplets sur $L = \{l_1, ..., l_n\}$. L'algorithme construit un arbre T sur L par le processus agglomératif suivant. Il maintient une collection d'arbres enracinés $\mathcal{S} = \{T_1, ..., T_r\}$.

1. Construire l'ensemble $\mathcal{S} = \{S_1, .., S_n\}$ où chaque S_i est un arbre consistant en une unique feuille étiquetée l_i.
2. Répéter $n-1$ fois :
   (a) Pour chaque $S_i, S_j \in \mathcal{S}$, remettre $score(S_i, S_j)$ à 0 ;
   (b) Pour chaque $xy|z \in \mathcal{R}$ tel que $x \in S_i, y \in S_j$ et $z \in S_k$ pour trois arbres différents S_i, S_j, S_k, mettre à jour *score* de la manière suivante :
   $score(S_i, S_j) := score(S_i, S_j) + 2$;
   $score(S_i, S_k) := score(S_i, S_k) - 1$;
   $score(S_j, S_k) := score(S_j, S_k) - 1$.
   (c) Choisir $S_i, S_j \in \mathcal{S}$ tel que $score(S_i, S_j)$ soit maximum ;
   (d) Créer un nouvel arbre $\mathcal{S}_k = (\mathcal{S}_i, \mathcal{S}_j)$;
   (e) $\mathcal{S} := \mathcal{S} \cup \{S_k\} \backslash \{S_i, S_j\}$.
3. Renvoyer l'unique arbre de $\mathcal{S}$.

On justifie à présent le ratio d'approximation. Introduisons les notations suivantes. Etant donné un noeud u de fils u_1, u_2, notons $\mathcal{R}(u) = \{xy|z \in \mathcal{R} : \exists a, b, c \in \{x, y, z\} \text{tq} a \in L(u_1), b \in L(u_2), c \notin L(u_1) \cup L(u_2)\}$. Partitionnons $\mathcal{R}(u)$ en $\mathcal{R}'(u)$ et $\mathcal{R}''(u)$, où $\mathcal{R}'(u)$ est l'ensemble des triplets de $\mathcal{R}(u)$ qui sont consistants avec T, et $\mathcal{R}''(u) = \mathcal{R}(u) \backslash \mathcal{R}'(u)$.

Observation 1. $|\mathcal{R}'(u)| \geq \frac{1}{3}|\mathcal{R}(u)|$.
Preuve. Considérons l'itération où le noeud u est créé comme noeud père des deux arbres S_i, S_j choisis à l'étape 2c. Clairement $score(S_i, S_j) \geq 0$. De plus,

par la définition de *score* aux étapes 2a et 2b et par construction de T, on a $score(S_i, S_j) = 2|R(u)'| - |R(u)''|$. Comme $|R(u)''| = |R(u)| - |R(u)'|$, on obtient $|\mathcal{R}(u)'| \geq \frac{1}{3}|\mathcal{R}(u)|$. □

De l'Observation 1 et du fait que les ensembles $\mathcal{R}(u)$ partitionnent $\mathcal{R}$, on conclut que l'algorithme produit bien une 3-approximation. □

6.2.3 Collections de triplets aléatoires et pseudo-aléatoires

Cette section porte sur des propriétés de collections de triplets complètes construites d'une manière aléatoire ou pseudo-aléatoire. On montre que pour une collection de triplet aléatoire, générée selon un modèle uniforme, la fraction maximum de triplets qui peuvent être satisfaits par un arbre est approximativement 1/3 (Proposition 6.10). On adapte ensuite une construction de [2] pour obtenir une construction déterministe d'une collection de triplets ayant des propriétés similaires (Proposition 6.13, Corollaire 6.14).

Introduisons les notations suivantes. Soit un ensemble L de cardinal n, et soit $<$ un ordre total arbitraire sur L. Soit $k \in \mathbb{N}$. On note L^k l'ensemble des tuples $(x_1, ..., x_k)$ avec $x_i \in L$ pour tout i. On note $[L]^k$ l'ensemble des tuples $(x_1, ..., x_k) \in L^k$ avec $x_1 > ... > x_k$. On note $\langle L \rangle^k$ l'ensemble des tuples $(x_1, ..., x_k) \in L^k$ ayant des coordonnées distinctes.

Par commodité, une collection de triplets simple $\mathcal{R}$ sera assimilée à une fonction partielle $\mathcal{R} : \langle L \rangle^3 \to \mathbb{Z}_3$, telle que : pour tout $x_0, x_1, x_2 \in L$ distincts, $\mathcal{R}(x_0, x_1, x_2) = i$ si et seulement si $x_{i+1}x_{i+2}|x_i \in \mathcal{R}$. En d'autres termes, $\mathcal{R}(x_0, x_1, x_2)$ indique l'élément séparé des deux autres par un triplet de $\mathcal{R}$. Notons que la fonction $\mathcal{R}$ est déterminée par sa restriction à $\langle L \rangle^3$.

On décrit maintenant une construction d'une collection de triplets aléatoire. Considérons la collection de triplets complète $\mathcal{R}$ sur L générée par le processus aléatoire suivant : pour chaque $t \in [L]^3, \mathcal{R}(t)$ est un élément choisi dans $\mathbb{Z}_3$ selon une loi uniforme. La Proposition suivante montre que pour la collection ainsi construite, la proportion maximum de triplets satisfiables est d'environ 1/3.

Proposition 6.10. *Soit* $\mu = \frac{1}{3}\binom{n}{3}$. *Soit* $\delta(n)$ *une fonction telle que* $\delta(n) = \Omega(\frac{\log n}{n})$. *Avec probabilité élevée :* $\textsc{Mtc}(\mathcal{R}) < (1 + \delta(n))\mu$.

Démonstration. Fixons δ. Etant donné un arbre binaire T sur L, on calcule la probabilité que $n(\mathcal{R}, T)$ dévie de son espérance d'un facteur $1 + \delta$. Etant donné un triplet $t \in rt(T)$, notons $\chi(\mathcal{R}, t)$ la variable indicatrice valant 1 si $t \in \mathcal{R}$ et 0 sinon. Observons que $n(\mathcal{R}, T)$ est la somme de variables aléatoires i.i.d. :

$$n(\mathcal{R}, T) = \sum_{t \in rt(T)} \chi(\mathcal{R}, t)$$

Notons que $\mathbb{E}[\chi(\mathcal{R}, t)] = \frac{1}{3}$ et donc $\mathbb{E}[n(\mathcal{R}, t)] = \frac{1}{3}\binom{n}{3} = \mu$. Par application des bornes de Chernoff à la variable aléatoire $n(\mathcal{R}, T)$, on obtient donc :

$$\mathbb{P}[n(\mathcal{R}, T) > (1 + \delta)\mu] \leq exp(-c\mu\delta^2)$$

pour une certaine constante c. En appliquant les bornes d'union, on obtient :

$$\mathbb{P}[\textsc{Mtc}(\mathcal{R}) > (1+\delta)\mu] \leq \sum_{T} \mathbb{P}[n(\mathcal{R},T) > (1+\delta)\mu] \leq 2^{n \log n} exp(-c\mu\delta^2)$$

On conclut en observant que si $\delta = \Omega(\frac{\log n}{n})$, alors $c\mu\delta^2 = \Omega(n \log^2 n)$, et donc l'expression ci-dessus tend vers 0 lorsque n tend vers l'infini. □

On décrit maintenant une construction déterministe d'une collection de triplets pseudo-aléatoire. Elle utilise la construction algébrique suivante, qui généralise la construction de [2] en introduisant un paramètre additif q. La construction originale de [2] fournit un q-coloriage des hyperarêtes de l'hypergraphe r-uniforme complet, avec les propriétés pseudo-aléatoires énoncées dans le Lemme 6.12 ci-dessous.

Définition 6.11. *Considérons des entiers $r, s > 1$, un nombre premier p tel que $s|p-1$, et un élément $q \in \mathbb{Z}_p$. Soit g un générateur de $\mathbb{Z}_p^*$, soit H le sous-groupe de $\mathbb{Z}_p^*$ généré par g^s, et pour chaque $i \in [s]$ soit H_i le coset Hg^i.*

Etant donné $j \in \mathbb{Z}_p^*$, on pose $[j]_p^s = i$ si $j \in H_i$, et on pose $[0]_p^s = 0$. On définit $\phi_{p,q}^{r,s} : \mathbb{Z}_p^r \to \{0, ..., s-1\}$ tel que pour tout $j = (j_1, ..., j_r) \in \mathbb{Z}_p^r$, $\phi_{p,q}^{r,s}(j) = [j_1 + ... + j_r + q]_p^s$.

Etant donné un sous-ensemble A de $\mathbb{Z}_p^r$, et étant donné $0 \leq j < s$, on pose :

$$n_j(A) = |\{i \in A : \phi_{p,q}^{r,s}(i) = j\}|$$

Le lemme suivant, tiré de [2], établit que si $A \subset \mathbb{Z}_p^r$ est formé à partir d'un produit cartésien et est de cardinal suffisamment grand, alors la fraction des hyperarêtes de A ayant une couleur donnée est environ $1/s$.

Lemme 6.12 ([2], Lemme 2.5). *Soient $A_1, ..., A_r$ des sous-ensembles de $\mathbb{Z}_p$, et soit $A = \{i \in [\mathbb{Z}_p]^r : i_j \in A_j, j = 1...r\}$. Alors pour tout $0 \leq j < s$,*

$$|n_j(A) - |A|/s| \leq c_r |A|^{1/2} (\log |A|)^{r-1} p^{(r-1)/2}$$

pour une certaine constante universelle $c_r > 0$ qui dépend seulement de r.

Bien que la construction de la Définition 6.11 diffère de [2] par l'utilisation d'un paramètre additionnel q, un examen attentif de la preuve du Lemme 2.3 de [2] montre qu'il reste valide si l'on remplace la somme $i_1 + ... + i_r$ par la somme $i_1 + ... + i_r + q$. Il est alors facile de voir que les Lemmes 2.4 et 2.5 de [2] restent corrects avec la nouvelle définition de $[.]_p^s$. Par conséquent, notre Lemme 6.12 est correct dans ce cadre.

On applique la construction de la Définition 6.11 avec $r = s = 3$ afin d'obtenir une collection de triplets minimalement complète ayant des propriétés pseudo-aléatoires. Plus précisément, on définit la collection de triplets $\mathcal{R}_p$ sur l'ensemble d'étiquettes $\mathbb{Z}_p$ tel que pour tout $(x, y, z) \in [\mathbb{Z}_p]^3$, $\mathcal{R}_p(x, y, z) = \phi_{p,0}^{3,3}(x, y, z)$ (rappelons qu'on a $x > y > z$ en vertu des définitions ci-dessus). A titre d'exemple, la figure 6.2 représente la collection de triplets $\mathcal{R}_p$ pour $p = 7$.

$$l_0l_2|l_1, l_0l_3|l_1, l_2l_3|l_0, l_1l_2|l_3, l_1l_4|l_0, l_0l_2|l_4, l_1l_2|l_4,$$
$$l_0l_3|l_4, l_1l_3|l_4, l_3l_4|l_2, l_0l_1|l_5, l_0l_2|l_5, l_1l_2|l_5, l_0l_3|l_5,$$
$$l_3l_5|l_1, l_2l_5|l_3, l_4l_5|l_0, l_1l_5|l_4, l_2l_5|l_4, l_4l_5|l_3, l_0l_1|l_6,$$
$$l_0l_2|l_6, l_2l_6|l_1, l_3l_6|l_0, l_1l_6|l_3, l_2l_6|l_3, l_0l_6|l_4, l_1l_6|l_4,$$
$$l_4l_6|l_2, l_3l_4|l_6, l_0l_6|l_5, l_5l_6|l_1, l_2l_5|l_6, l_3l_5|l_6, l_4l_5|l_6.$$

FIG. 6.2 – Les 35 triplets de la collection $\mathcal{R}_7$ sur l'ensemble d'étiquettes $\mathbb{Z}_7$ identifié avec$\{l_0, l_1, l_2, l_3, l_4, l_5, l_6\}$. Dans la construction de la Définition 6.11, on choisit $g = 3$ comme générateur de $\mathbb{Z}_7^*$ de sorte que $g^s = g^3 = \overline{6}$, $H_0 = H = \{\overline{1}, \overline{6}\}$, $H_1 = \{\overline{3}, \overline{4}\}$, et $H_2 = \{\overline{2}, \overline{5}\}$. Par exemple, le triplet $l_0l_2|l_1$ est obtenu comme suit : on a $[\overline{0} + \overline{1} + \overline{2}]_7^3 = 1$, donc $\mathcal{R}_7(l_2, l_1, l_0) = 1$, ce qui implique que $l_0l_2|l_1 \in \mathcal{R}_7$. Une solution optimale pour $\mathcal{R}_7$ est l'arbre $T = ((((l_0, l_2), l_6), l_1), ((l_4, l_5), l_3))$ qui satisfait $n(\mathcal{R}_7, T) = 21$.

La proposition suivante montre que $\mathcal{R}_p$ est pseudo-aléatoire : pour p suffisamment grand, chaque arbre binaire sur $\mathbb{Z}_p$ est consistent avec approximativement un tiers des triplets de $\mathcal{R}_p$. Sa preuve repose sur le Lemme 6.12 et utilise le nouveau paramètre additif q.

Proposition 6.13. *Pour chaque arbre binaire T sur $\mathbb{Z}_p$, on a : $|n(\mathcal{R}_p, T) - \frac{1}{3}\binom{p}{3}| \leq cp^{5/2} \log p$ pour une certaine constante c.*

Démonstration. Fixons $z \in \mathbb{Z}_p$. Soient $L_{z,1}, ..., L_{z,m}$ les clades pendant le long du chemin joignant z à la racine dans T ; ces ensembles forment une partition de $\mathbb{Z}_p \backslash \{z\}$. Pour tout $i \in [m]$, notons $n_{z,i}$ le nombre de triplets de $\mathcal{R}_p \cap rt(T)$ de la forme $xy|z$ avec $x, y \in L_{z,i}$. On a donc : $n(\mathcal{R}_p, T) = \sum_{z \in \mathbb{Z}_p} \sum_i n_{z,i}$.

Fixons $i \in [m]$, et posons $A_{z,i} = [L_{z,i}]^2$. On va montrer que : $|n_{z,i} - |A_{z,i}|/3| \leq cp^{3/2} \log p$. Posons :

$$L_z = \{x \in L_i : x < z\}$$
$$R_z = \{x \in L_i : x > z\}$$

et partitionnons $A_{z,i}$ en trois ensembles $A_{z,i}^{(1)} = [L_z]^2$, $A_{z,i}^{(2)} = R_z \times L_z$, $A_{z,i}^{(3)} = [R_z]^2$. Définissons $f : [\mathbb{Z}_p \backslash \{z\}]^2 \to \mathbb{Z}_3$ en posant $f(x, y) = \phi_{p,z}^{2,3}(x, y)$. Etant donné $A \subset [\mathbb{Z}_p]^2, j \in \mathbb{Z}_3$, posons $n'_j(A) = |\{i \in A : f(i) = j\}|$. On a donc :

$$n_{z,i} = n'_1(A_{z,i}^{(1)}) + n'_2(A_{z,i}^{(2)}) + n'_3(A_{z,i}^{(3)})$$

Comme $f = \phi_{p,z}^{2,3}$, le Lemme 6.12 s'applique et donne l'inégalité suivante : pour chaque $j \in \mathbb{Z}_3$,

$$|n'_j(A_{z,i}^{(j)}) - |A_{z,i}^{(j)}|/3| \leq c'|A_{z,i}^{(j)}|^{1/2}(\log |A_{z,i}^{(j)}|)p^{1/2}$$

pour une certaine constante c'. En utilisant l'inégalité triangulaire et en sommant sur l'indice j, on obtient :

$$|n_{z,i} - |A_{z,i}|/3| \leq c'|A_{z,i}|^{1/2}(\log|A_{z,i}|)p^{1/2}$$

Posons $S = \sum_{z\in\mathbb{Z}_p}\sum_i |A_{z,i}|$ et $S' = \sum_{z\in\mathbb{Z}_p}\sum_i |A_{z,i}|^{1/2}$. En sommant sur les indices z, i dans l'inégalité précédente, on obtient :

$$|n(\mathcal{R}_p, T) - S/3| \leq cS'p^{1/2}\log p$$

pour une certaine constante c. On conclut en observant que $S = \binom{p}{3}$. et que $S' \leq p^2$ (cette dernière inégalité résultant du fait que pour z fixé, $\sum_i |A_{z,i}|^{1/2} \leq \sum_i |L_{z,i}| = p-1$). □

Corollaire 6.14. *On a :* $|\textsc{Mtc}(\mathcal{R}_p) - \frac{1}{3}\binom{p}{3}| \leq cp^{5/2}\log p$.

Observons que $|\mathcal{R}_p| = \binom{p}{3}$. Donc, $\textsc{Mtc}(\mathcal{R}_p)$ sera proche de $\frac{1}{3}.|\mathcal{R}_p|$ pour p suffisamment grand.

6.2.4 Résultats de difficulté

On montre dans cette section que le problème MTC général est APX-dur (Proposition 6.15), et que sa restriction aux collections minimalement complètes est NP-difficile (Proposition 6.16).

Proposition 6.15. MTC *est* APX-*dur.*

Démonstration. On donne une réduction préservant la mesure depuis MAXIMUM ACYCLIC SUBGRAPH.

Rappelons d'abord la définition du problème. Le problème MAXIMUM ACYCLIC SUBGRAPH (MAS) consiste, étant donné un graphe $G = (V, A)$, à trouver une permutation π de V telle que $n(G, \pi)$ soit maximum. On note $\textsc{Mas}(G)$ la valeur atteinte pour l'optimum.

Décrivons maintenant la réduction. Etant donnée une instance $G = (V, A)$ de MAXIMUM ACYCLIC SUBGRAPH, on lui associe une instance de MTC construite comme suit. L'ensemble d'étiquettes est $L = V \cup \{x\}$, et l'ensemble de triplets est $\mathcal{R} = \{ux|v : (u, v) \in A\}$.

La correction de la réduction résulte des observations suivantes.

Observation 1. Etant donnée π permutation de V, il existe T arbre sur L tel que $n(\mathcal{R}, T) \geq n(\mathcal{G}, \pi)$.
Preuve : soit $\pi = v_1...v_n$ une permutation de V. Considérons $T = rake(x, v_1, ..., v_n)$. Alors $n(\mathcal{R}, T) \geq n(G, \pi)$, puisque pour chaque arc $(u, v) \in C(G, \pi)$, le triplet $xu|v$ est dans $C(\mathcal{R}, T)$. □

Observation 2. Etant donné T arbre sur L, il existe π permutation de V telle que $n(G, \pi) \geq n(\mathcal{R}, T)$.

Preuve : soit T un arbre sur L. Soient $V_1, ..., V_p$ les clades pendant le long du chemin joignant x à la racine. Pour tout i, soit π_i une permutation arbitraire de V_i, et soit $\pi = \pi_1 ... \pi_p$. Alors $n(G, \pi) \geq n(\mathcal{R}, T)$, puisque pour chaque triplet $xu|v \in C(\mathcal{R}, T)$, on a $u \in V_i, v \in V_j$ avec $i < j$, et donc $(u, v) \in C(G, \pi)$. □

Des observations précédentes, il résulte que $\textsc{Mas}(G) = \textsc{Mtc}(\mathcal{R})$. On a donc une réduction préservant la mesure de Mas à Mtc, ce qui établit l'APX-difficulté de Mtc. □

Proposition 6.16. Mtcc *est* NP*-dur.*

Ce résultat est établi au moyen d'une technique de preuve dûe à [4, 2]. On donne une réduction de Mtc à Mtcc consistant à transformer une instance arbitraire $\mathcal{R}$ en une instance complète $\mathcal{R}'$. La transformation se fait en deux étapes. Dans un premier temps, on transforme $\mathcal{R}$ en éclatant chaque étiquette en p copies, il s'agit d'une opération appelée p-expansion dont la définition est donnée ci-dessous. Dans un second temps, on complète l'instance obtenue par ajout d'une collection de triplets pseudo-aléatoire ; on utilise à cette fin la collection $\mathcal{R}_p$ construite en Section 6.2.3.

La correction de la réduction provient du fait que lorsqu'on applique à $\mathcal{R}$ l'opération de p-expansion, la mesure de l'instance est multipliée par p^3, tandis que l'ajout d'une collection pseudo-aléatoire perturbe la mesure d'un terme additif ϵ. En choisissant p tel que $p^3 > \epsilon$, on peut donc déduire la mesure de $\mathcal{R}$, ayant calculé celle de $\mathcal{R}'$. En d'autres termes, compléter la collection obtenue après expansion introduit une erreur additive, qui peut être rendue faible par un choix judicieux de p, de façon à ce que la mesure de l'instance d'origine soit approximativement préservée (à une constante multiplicative près).

Définition 6.17. *Soit $\mathcal{R}$ une collection de triplets sur L, et soit $m \in \mathbb{N}$. La* m-expansion *de $\mathcal{R}$ est la collection de triplets $\mathcal{R}^m$ sur $L' = \{x_i : x \in L, i \in [m]\}$ définie comme suit. Pour chaques étiquettes distinctes $x, y, z \in L$, pour chaque $i, j, k \in [m]$, $\mathcal{R}^m(x_i, y_j, z_k)$ est défini et égal à $\mathcal{R}(x, y, z)$.*

Décrivons maintenant la réduction. Considérons une collection de triplets $\mathcal{R}$ sur L donnée comme instance de Mtc. Soit $n = |L|$, soit p un nombre premier, et soit $L' = \{x_i : x \in L, i \in \mathbb{Z}_p\}$. On construit une collection de triplets complète $\mathcal{R}'$ sur L' comme suit. Considérons $x_i, y_j, z_k \in L'$.

1. si x, y, z sont distincts et si $\{x, y, z\}$ est résolu par $\mathcal{R}$, alors $\mathcal{R}'(x_i, y_j, z_k) = \mathcal{R}(x, y, z)$;
2. sinon, si i, j, k sont distincts, alors $\mathcal{R}'(x_i, y_j, z_k) = \mathcal{R}_p(i, j, k)$;
3. sinon, $\mathcal{R}'(x_i, y_j, z_k)$ est choisi arbitrairement dans $\mathbb{Z}_3$.

Pour $i \in \{1, 2, 3\}$, soit $\mathcal{R}'_i$ l'ensemble de triplets défini par la condition i ci-dessus, on a donc $\mathcal{R}' = \mathcal{R}'_1 \cup \mathcal{R}'_2 \cup \mathcal{R}'_3$. Observons que $\mathcal{R}'_1 = \mathcal{R}^p$; comme annoncé ci-dessus, on voit donc que $\mathcal{R}'$ est obtenu à partir de la p-expansion de $\mathcal{R}$ par ajout d'une collection pseudo-aléatoire ($\mathcal{R}'_2$). Notons qu'on ajoute également une collection arbitraire $\mathcal{R}'_3$; on verra que le terme dominant dans

l'erreur introduite par $\mathcal{R}'_2 \cup \mathcal{R}'_3$ provient de $\mathcal{R}'_2$, ce qui fait que le choix de $\mathcal{R}'_3$ n'a pas d'incidence.

Le lemme suivant caractérise le nombre maximum de triplets consistants de $\mathcal{R}'_1$.

Lemme 6.18. $\text{MTC}(\mathcal{R}'_1) = p^3\text{MTC}(\mathcal{R})$.

Démonstration. $\text{MTC}(\mathcal{R}'_1) \leq p^3\text{MTC}(\mathcal{R})$: soit T un arbre binaire sur L tel que $n(\mathcal{R}, T) = \text{MTC}(\mathcal{R})$. Considérons l'arbre binaire T' sur L' obtenu à partir de T en substituant chaque feuille l par un arbre quelconque sur l'ensemble $\{l_1, ..., l_p\}$. Alors $\text{MTC}(\mathcal{R}'_1) \leq n(\mathcal{R}'_1, T') = p^3 n(\mathcal{R}, T) = p^3\text{MTC}(\mathcal{R})$.

$\text{MTC}(\mathcal{R}'_1) \geq p^3\text{MTC}(\mathcal{R})$: supposons que $L = \{v_1, ..., v_n\}$. Pour chaque suite d'indices $i_1, ..., i_n \in \mathbb{Z}_p$, on a $\text{MTC}(\mathcal{R}'_1|\{v_1^{i_1}, ..., v_n^{i_n}\} \geq \text{MTC}(\mathcal{R})$. En sommant sur ces suites, on obtient p^n termes où chaque triplet est compté p^{n-3} fois, d'où : $\text{MTC}(\mathcal{R}'_1) \geq \frac{p^n}{p^{n-3}}\text{MTC}(\mathcal{R}) = p^3\text{MTC}(\mathcal{R})$. □

Le lemme suivant caractérise le nombre de triplets consistants d'une m-expansion de $\mathcal{R}_p$.

Lemme 6.19. *Pour chaque arbre binaire T, $|n(\mathcal{R}_p^m, T) - m^3\binom{p}{3}/3| \leq cm^3p^{5/2}\log p$.*

Démonstration. Supposons que $\mathbb{Z}_p = \{v_1, ..., v_p\}$. Considérons une suite d'indices $\sigma = (i_1, ..., i_p)$ dans $[m]$, et soit $L_\sigma = \{v_1^{i_1}, ..., v_p^{i_p}\}$. Alors $\mathcal{R}_p^m|L_\sigma$ est isomorphe à $\mathcal{R}_p$, et on a donc par la Proposition 6.13 : $|n(\mathcal{R}_p^m|\ L_\sigma, T|\ L_\sigma) - \binom{p}{3}/3| \leq cp^{5/2}\ \log p$.

En sommant sur ces suites d'indices, chaque triplet commun à $\mathcal{R}_p^m, T$ est compté m^{p-3} fois, et donc :

$$n(\mathcal{R}_p^m, T) = \frac{1}{m^{p-3}} \sum_\sigma n(\mathcal{R}_p^m|L_\sigma, T|L_\sigma)$$

On en déduit :

$$\begin{aligned} |\ n(\mathcal{R}_p^m, T) - m^3\binom{p}{3}/3| &= |\frac{1}{m^{p-3}} \sum_\sigma (n(\mathcal{R}_p^m|\ L_\sigma, T|L_\sigma) - \binom{p}{3})| \\ &\leq \frac{1}{m^{p-3}} \sum_\sigma |n(\mathcal{R}_p^m|L_\sigma, T|L_\sigma) - \binom{p}{3}| \\ &\leq \frac{1}{m^{p-3}} \sum_\sigma cp^{5/2}\log p = cm^3p^{5/2}\log p \end{aligned}$$

ce qui est le résultat annoncé. □

Soit N_3 le nombre d'ensembles $\{x, y, z\} \subseteq L$ et non résolus par $\mathcal{R}$. Soit $N_2 = n(n-1)$ et soit $N_1 = n$. Soit $N = N_1 + 8N_2 + 27N_3$.

Le lemme suivant caractérise le nombre de triplets consistants de $\mathcal{R}'_2$.

Lemme 6.20. *Pour chaque T arbre binaire sur L', $|n(\mathcal{R}'_2, T) - N\binom{p}{3}\ /3| \leq cNp^{5/2}\log p$.*

Démonstration. Disons qu'un multiensemble $\{\{x, y, z\}\} \subseteq L$ est non résolu par $\mathcal{R}$ si et seulement si : (i) soit x, y, z sont distincts et $\{x, y, z\}$ est non résolu par $\mathcal{R}$, (ii) soit x, y, z sont non distincts. Pour un multiensemble $t = \{\{x, y, z\}\}$ non résolu par $\mathcal{R}$, soit $n(T, t)$ le nombre de triplets de $\mathcal{R}'_2$ qui impliquent x_i, y_j, z_k et sont consistants avec T. On a donc : $n(\mathcal{R}'_2, T) = \sum_t n(T, t)$.

Pour $t = \{\{x, y, z\}\}$ donné, posons $L_t = \{x_i, y_i, z_i : i \in \mathbb{Z}_p\}$. On a alors $n(T, t) = n(\mathcal{R}'_2|L_t, T|L_t)$. Soit $m \in \{1, 2, 3\}$ le nombre d'étiquettes distinctes dans t, alors $\mathcal{R}'_2|L_t$ est isomorphe à $\mathcal{R}^m_p$, et donc par le Lemme 6.19 : $|n(\mathcal{R}'_2|L_t, T|L_t) - m^3\binom{p}{3}/3| \leq cm^3p^{5/2}\log p$.

Observons que pour $i \in \{1, 2, 3\}$, il y a exactement N_i multiensembles pour lesquels $m = i$. On obtient le résultat annoncé en sommant sur chaque multiensemble t dans l'inégalité précédente. □

Le lemme suivant caractérise le nombre de triplets consistants de $\mathcal{R}'_3$.

Lemme 6.21. *Pour chaque T arbre binaire sur L', $n(\mathcal{R}'_3, T) \leq Np^2$.*

Démonstration. Cela résulte du fait que $\mathcal{R}'_3$ est défini seulement pour des ensembles $\{x_i, y_j, z_k\}$ avec i, j, k non distincts, et ces ensembles sont nombre $\leq N(p(p-1)+p) = Np^2$. □

On est maintenant en mesure de prouver la Proposition.

Preuve de la Proposition 6.16. Il résulte des lemmes précédents que $\textsc{Mtc}(\mathcal{R}')$ est une approximation de $\textsc{Mtc}(\mathcal{R}_1) + Np^3/18 = p^3\textsc{Mtc}(\mathcal{R}) + Np^3/18$ à une erreur additive près de l'ordre de $cNp^{5/2}\log p$ (pour une certaine constante c). En divisant par $\frac{p^3}{18}$, on obtient :

$$\left|\frac{18\textsc{Mtc}(\mathcal{R}')}{p^3} - (18\textsc{Mtc}(\mathcal{R}) + N)\right| \leq c'Np^{-1/2}\ \log p$$

pour une certaine constante c'. Comme $N = O(n^3)$, il existe un polynôme $P(n)$ tel que si $p > P(n)$ alors le membre droit de l'inégalité est inférieur à $\frac{1}{2}$, ce qui implique que $\lceil\frac{18\textsc{Mtc}(\mathcal{R}')}{p^3}\rceil = 18\textsc{Mtc}(\mathcal{R}) + N$. Ayant calculé $\textsc{Mtc}(\mathcal{R}')$, on peut donc obtenir $\textsc{Mtc}(\mathcal{R})$ en temps polynomial. □

6.2.5 Remarques

L'algorithme de la Proposition 6.9 est dû à Jansson [6]. [11] décrit un autre algorithme de 3-approximation qui a la propriété de toujours produire un peigne. La réduction de la Proposition 6.15 est dûe à [5]. Notons également que la restriction aux collections complètes, $\textsc{Mtcc}$, admet vraisemblablement un schéma d'approximation polynomial [13, 12].

6.3 Problèmes Mli et Mti

Dans cette section, on considère les problèmes visant à rendre une collection de triplets compatible par suppression d'un nombre minimum d'étiquettes (Mli) resp. de triplets (Mti). Certains résultats de cette section apparaissent dans [6]. Cette section s'organise selon le plan suivant. Après avoir défini les problèmes en Section 6.3.1, on présente des résultats algorithmiques en Section 6.3.2 et des résultats de difficulté en Section 6.3.3. La section 6.3.4 contient des remarques conclusives.

6.3.1 Définitions des problèmes

Soit $\mathcal{R}$ une collection de triplets sur L. On appelle *conciliateur* de $\mathcal{R}$ un ensemble d'étiquettes $P \subseteq L$ tel que $\mathcal{R}\backslash P$ soit compatible. Le problème Minimum Label Inconsistency (Mli) vise à trouver un conciliateur de cardinal minimum d'une collection de triplets donnée en entrée. On note Mlic la restriction du problème aux collections complètes.

Soit $\mathcal{R}$ une collection de triplets sur L. Soit T un arbre sur L, on note $M(\mathcal{R}, T) = \mathcal{R}\backslash rt(T)$ et $m(\mathcal{R}, T) = |M(\mathcal{R}, T)|$.

On appelle *t-conciliateur* de $\mathcal{R}$ un ensemble de triplets $S \subseteq \mathcal{R}$ tel que $\mathcal{R}\backslash S$ soit compatible. On désigne par Minimum Triplet Inconsistency (Mti) le problème consistant à trouver un t-conciliateur de cardinal minimum d'une collection $\mathcal{R}$. Le problème est équivalent à trouver un arbre T sur L tel que $m(\mathcal{R}, T)$ soit minimum. On note Mtic la restriction du problème Mti aux collections complètes.

6.3.2 Algorithmes

Algorithmes pour Mlic

Proposition 6.22. *(i)* Mlic *peut être résolu en temps* $\min(4^p n^3, n^4 + 3.12^p)$ *;*
(ii) Mlic *peut être approximé à un facteur 4 en temps* $O(n^3)$.

Démonstration. Point (i) : d'une part, on peut résoudre le problème en temps $O(4^p n^3)$ par une procédure de recherche bornée, en utilisant l'algorithme Find-Tree-Or-Conflict de la Proposition 6.8 pour décider la compatibilité ou obtenir un conflit de taille ≤ 4. D'autre part, la Proposition 6.5 permet de réduire le problème à 4HS en construisant l'ensemble des conflits de taille ≤ 4 ; en ayant recours à l'algorithme pour 4HS décrit par [10], on obtient un algorithme de temps $O(n^4 + 3.12^p)$.

Point (ii) : on obtient un algorithme de 4-approximation en adaptant l'algorithme de la Proposition 6.8 de façon à éliminer les conflits à la volée. On maintient un ensemble d'étiquettes L' (initialisé à l'ensemble vide), et un arbre T représenté par $\mathcal{R}|L'$ (initialisé à l'arbre vide). On tente d'insérer successivement chaque élément de L. Soit x un élément à insérer, on appelle l'algorithme Insert-Label-Or-Find-Conflict$(\mathcal{R}, L', x, T)$ du Lemme 6.7 ; si l'insertion

réussit et renvoie un arbre T', on fait $L' \leftarrow L' \cup \{x\}$ et $T \leftarrow T'$; si l'insertion échoue et renvoie un conflit C, on fait $L' \leftarrow L' \backslash C$ et $T \leftarrow T \backslash C$. □

Algorithmes pour Mtic

Proposition 6.23. Mtic *peut être résolu en temps* $O(4^p n^3)$.

Démonstration. L'algorithme utilise la recherche bornée ; son principe consiste à itérativement identifier un t-conflit, et inverser un triplet impliqué dans ce t-conflit, jusqu'à obtention d'une collection compatible. Pour interdire deux inversions successives d'un même triplet, l'algorithme verrouille les triplets inversés.

Décrivons une étape de l'algorithme. Soit $I = (\mathcal{R}, p)$ l'instance considérée, et soit F l'ensemble des triplets verrouillés. Soit t, t' deux triplets sur un même ensemble d'étiquettes tq $t \in \mathcal{R}$, l'opération de *transformation* de t en t' procède comme suit : (i) échouer si $p = 0$ ou si $t \in F$ (t est verrouillé), (ii) sinon, faire $\mathcal{R} \leftarrow \mathcal{R} - \{t\} + \{t'\}$, faire $p \leftarrow p - 1$, et $F \leftarrow F \cup \{t'\}$ (verrouiller t').

L'algorithme appelle la procédure Find-Tree-Or-Conflict de la Proposition 6.8 pour décider si $\mathcal{R}$ est compatible. Si la réponse est positive, l'algorithme s'arrête et répond positivement. Si la réponse est négative, alors : (i) si $p = 0$, l'algorithme échoue, (ii) si $p > 0$, on identifie un t-conflit C de cardinal 3, et on va chercher à inverser un triplet de C en procédant comme suit. C a la forme $t_1 = ab|c, t_2 = bc|d$ et :

- soit $t_3 = ad|b$ ou $t_3 = bd|a$; alors l'algorithme choisit l'un des quatre cas suivants : (i) transformer t_1 en $t'_1 = a|bc$, (ii) transformer t_1 en $t'_1 = ac|b$, (iii) transformer t_2 en $t'_2 = bd|c$, (iv) transformer t_3 en $t'_3 = ab|d$. La correction de cette règle se montre en vérifiant que tout arbre sur $\{a, b, c, d\}$ contient l'un des triplets $a|bc, ac|b, bd|c, ab|d$.
- soit $t_3 = ad|c$ ou $t_3 = cd|a$; alors l'algorithme choisit l'un des quatre cas suivants : (i) transformer t_1 en $t'_1 = a|bc$, (ii) transformer t_2 en $t'_2 = b|dc$, (iii) transformer t_2 en $t'_2 = bd|c$, (iv) transformer t_3 en $t'_3 = ac|d$. La correction de cette règle se montre en vérifiant que tout arbre sur $\{a, b, c, d\}$ contient l'un des triplets $a|bc, b|dc, bd|c, ac|d$.

Pour finir, analysons le temps d'exécution de l'algorithme. Chaque étape (test de compatibilité et substitution d'un triplet) prend un temps $O(n^3)$, et conduit à un choix comportant quatre alternatives ; chaque alternative conduit à inverser un nouveau triplet et à décrémenter p. Le temps d'exécution est donc $O(4^p n^3)$ comme annoncé. □

6.3.3 Résultats de difficulté

On présente deux résultats de NP-difficulté : on montre que Mlic est aussi difficile que 3HS (Proposition 6.24), et on montre que le problème Mli général est aussi difficile que HS (Proposition 6.25).

Difficulté de Mlic

Proposition 6.24. *Il existe une réduction préservant la mesure de* 3HS *à* MLIC.

Démonstration. Soit $H = (V, E)$ un hypergraphe 3-uniforme instance de 3HS. Soit $L = V \cup E$. Soit $<$ un ordre total sur V, étendons-le en un ordre total $<'$ sur L satisfaisant : pour tout $e = \{a, b, c\} \in E$ avec $a < b < c$, on a $a <' e <' b$. Soit $\mathcal{R}$ la collection sur L définie comme suit. $\mathcal{R}$ contient des triplets de deux types :

- triplets de type I : pour chaque $x, y, z \in L$ tel que $x <' y <' z$ et $\{x, y, z\} \notin E$: $\mathcal{R}$ contient le triplet $xy|z$;
- triplets de type II : pour chaque $x, y, z \in L$ tel que $x <' y <' z$ et $\{x, y, z\} \in E$: $\mathcal{R}$ contient le triplet $yz|x$.

Clairement, $\mathcal{R}$ est une collection complète. On montre que : H a un transversal de cardinal $\leq p$ si et seulement si $\mathcal{R}$ a un conciliateur de cardinal $\leq p$.

($\Rightarrow$) : supposons que H a un transversal S de cardinal $\leq p$. On montre que S est un conciliateur de $\mathcal{R}$. Soit $T = rake(L\backslash S, <')$, alors $\mathcal{R}\backslash S$ représente T : en effet, $\mathcal{R}\backslash S$ ne contient que des triplets de type I (puisque S est un transversal de H), et ces triplets ont la même résolution dans $\mathcal{R}$ et dans T. On conclut que $\mathcal{R}$ a un conciliateur de cardinal $\leq p$.

($\Leftarrow$) : supposons que $\mathcal{R}$ a un conciliateur P de taille $\leq p$. Considérons $H' = H\backslash P$, alors $H' = (V\backslash P, E')$ où $E' = \{e \in E : e \subseteq V\backslash P\}$. Construisons un transversal C de H' (par exemple par une méthode gloutonne), et soit $S = P - E + C$. Alors :

- $S \cap E = \emptyset$, par définition de S, et S est un transversal de H, puisque pour tout $e \in E$, soit e est couverte par P, soit $e \in E'$ est e est couverte par C.
- $|S| \leq |P|$: montrons d'abord que $|C| \leq |E'| \leq |P \cap E|$. La première inégalité résulte de la construction de C. La seconde inégalité résulte du fait que pour tout $e \in E'$, on doit avoir $e \in P \cap E$. En effet, si $e = \{a, b, c\} \in E'$ avec $a < b < c$, alors $a, b, c \notin C$, et si l'on avait $e \notin P \cap E$ on obtiendrait que $ae|b, be|c, bc|a \in \mathcal{R}\backslash P$, impossible car $\mathcal{R}\backslash P$ est incompatible. On a donc établi que $|C| \leq |P \cap E|$, et comme $S = P - E + C$ on conclut que $|S| \leq |P|$.

On conclut que H a un transversal de taille $\leq p$. □

Difficulté de Mti

On présente maintenant un résultat de difficulté pour le problème MTI général.

Proposition 6.25. *Sous l'hypothèse* $\mathsf{P} \neq \mathsf{NP}$*, le problème* MTI *n'est pas approximable à* $\Omega(\log n)$.

La preuve de ce résultat s'effectue en deux étapes. On considère une version pondérée du problème MTI, appelée MTIW.

Etant donné un ensemble d'étiquettes L, une *collection de triplets pondérée* est une fonction $\mathcal{R} : T(L) \to \mathbb{N}$. Etant donné un arbre binaire T sur L, on

définit $m(\mathcal{R}, T) = \sum_{t \in T(L) \setminus rt(T)} \mathcal{R}(t)$. Le problème MTIW prend une collection de triplets pondérée $\mathcal{R}$, définie sur un ensemble d'étiquettes L, et cherche un arbre binaire T sur L qui minimise $m(\mathcal{R}, T)$.

On donne une réduction préservant la mesure de HITTING SET à MTIW (Lemme 6.26), suivie d'une réduction préservant la mesure de MTIW à MTI (Lemme 6.27).

Lemme 6.26. *Il existe une réduction préservant la mesure de* HS *à* MTIW.

Démonstration. Soit $H = (V, E)$ un hypergraphe donné comme instance de HITTING SET. Pour chaque $e \in E$, posons $m_e = |e| - 1$. Pour chaque $e \in E, v \in V$, on définit une collection de triplets pondérée $\mathcal{R}_{e,v}$ comme suit. Soit $v_1, ..., v_{m_e}$ une énumération de $e \setminus \{v\}$. L'ensemble d'étiquettes $L_{e,v}$ consiste en : (i) une étiquette a, (ii) pour chaque $v \in V$, deux étiquettes b_v, c_v, (iii) pour chaque $v \in V, j \leq 1 < m_e$, une étiqette $d_{e,v,j}$. $\mathcal{R}_{e,v}$ donne un poids non nuls aux triplets suivants : (i) le triplet $t_v = b_v c_v | a$, (ii) le triplet $c_v d_{e,v,1} | b_{v_1}$, (iii) pour chaque $1 < j < m_e$, un triplet $d_{e,v,j-1} d_{e,v,j} | b_{v_j}$, (iv) le triplet $d_{e,v,m_e-1} a | b_{v_{m_e}}$. Le triplet t_v reçoit le poids 1, tandis que tous les autres reçoivent un poids élevé $W > n$.

Pour une arête $e \in E$, on définit la collection de triplets pondérée $\mathcal{R}_e$ sur l'ensemble $L_e = \cup_{v \in E} L_{e,v}$ comme l'extension commune des fonctions $\mathcal{R}_{e,v}$ ($v \in e$). On définit ensuite la collection de triplets pondérée $\mathcal{R}$ sur l'ensemble $L = \cup_{e \in E} L_e$ comme l'extension commune des fonctions $\mathcal{R}_e$ ($e \in E$).

L'intuition derrière cette construction est la suivante. Pour $e \in E, v \in e$ donnés, le graphe $G(\mathcal{R}_{e,v}, L_{e,v})$ est le chemin P_v formé par les sommets $b_v, c_v, d_{e,v,1}, ..., d_{e,v,m_e-1}, a$. Le graphe $G(\mathcal{R}_e, L_e)$ est l'union de ces graphes, il consiste donc en un sommet central a d'où partent les chemins P_v ($v \in e$). Ce graphe est connexe, donc $\mathcal{R}_e$ est incompatible. Cependant, la suppression d'un triplet t_v rend le graphe de Aho disconnexe, en déconnectant b_v du reste du graphe. En fait, la suppression d'un triplet t_v rend la collection $\mathcal{R}_e$ compatible, puisque lors de la récursion sur le sous-ensemble formé par $L_e \setminus \{b_v\}$, une arête est détruite sur chaque chemin $P_{v'}$ avec $v' \neq v$. Ceci implique que, pour rendre $\mathcal{R}$ compatible, pour chaque arête e au moins l'un des triplets t_v doit être violé, et les sommets correspondants forment un transversal de H.

On montre à présent formellement qu'on a bien une réduction préservant la mesure.

Fait 1. Etant donné un arbre binaire T sur L, on peut construire en temps polynomial un transversal de H de taille $\leq m(\mathcal{R}, T)$.

Preuve. Soit S l'ensemble des éléments $v \in V$ t.q. $t_v \in rt(T)$, alors $|H| \leq m(\mathcal{R}, T)$. Pour chaque $e \in E$, comme $G(\mathcal{R}_e, L_e)$ est connexe, il doit exister $v \in E$ t.q. $t_v \notin rt(T)$. Donc S est un transversal de H.

Fait 2. Etant donné S transversal de H, on peut construire en temps polynomial un arbre binaire T sur L t.q. $m(\mathcal{R}, T) = |S|$.

Preuve. Pour $v \in V$, on divise les étiquettes indexées par v (exceptée b_v si $v \in S$) en deux ensembles σ_v et σ'_v.

– si $v \in S$, on pose :

$$\sigma_v = \emptyset$$
$$\sigma'_v = \{c_v\} \cup \{d_{e,v,j} : e \ni v, 1 \leq j < m_e\}.$$

– si $v \notin S$, alors pour chaque arête $e \ni v$ soit l'énumération $v_1, ..., v_{m_e}$ de $e \backslash \{v\}$ considérée ci-dessus, et soit $j_{v,e}$ le plus petit j tel que $v_j \in S$ (un tel indice existe puisque e doit être couverte par un élément de S distinct de v). On pose :

$$\sigma_v = \{b_v, c_v\} \cup \{d_{e,v,j} : e \ni v, 1 \leq j < j_{v,e}\}$$
$$\sigma'_v = \{d_{e,v,j} : e \ni v, j_{v,e} \leq j \leq m_e - 1\}.$$

Supposons que $V = \{v_1, ..., v_n\}$. On définit les arbres τ, τ', τ'' comme suit.

– Pour chaque $v \in V$, soit τ_v un arbre binaire arbitraire sur l'ensemble d'étiquettes σ_v. Soit $\tau = rake(\tau_{v_1}, ..., \tau_{v_n}\})$.
– Soit τ' un arbre binaire arbitraire sur l'ensemble d'étiquettes $a \cup \sigma'_{v_1} \cup ... \cup \sigma'_{v_n}$.
– Soit τ'' un arbre binaire arbitraire sur l'ensemble d'étiquettes $\{b_v : v \in S\}$.

On pose $T = ((\tau, \tau'), \tau'')$. On va maintenant montrer que : pour chaque $e \in E, v \in e$, les triplets de $\mathcal{R}_{e,v}$ sont consistants avec T, à l'exception des triplets t_v ($v \in S$).

– triplets définis par la condition (i) : si $v \notin S$, alors b_v, c_v apparaissent dans τ, tandis que a apparaît dans τ', donc $b_v c_v | a$ est consistant avec T.
– triplets définis par la condition (ii) : on montre que $c_v d_{e,v,1} | b_{v_1}$ est consistant avec T, en considérant trois sous-cas :
  – si $v \in S$: alors $c_v, d_{e,v,1}$ apparaissent dans τ', tandis que b_{v_1} apparaît dans τ ou τ'' ;
  – si $v \notin S$ et $j_{v,e} > 1$: alors $c_v, d_{e,v,1}$ apparaissent dans τ_v, tandis que b_{v_1} apparaît dans τ'_{v_1} (clairement $v_1 \neq v$) ;
  – si $v \notin S$ et $j_{v,e} = 1$: alors c_v apparaît dans τ_v, $d_{e,v,1}$ apparaît dans τ' et b_{v_1} apparaît dans τ'' (on a $v_1 \in S$ puisque $j_{v,e} = 1$).
– triplets définis par la condition (iii) : on montre que $d_{e,v,j-1} d_{e,v,j} | b_{v_j}$ est consistant avec T, en considérant trois sous-cas :
  – si $v \in S$: alors $d_{e,v,j-1}, d_{e,v,j}$ apparaissent dans τ', tandis que b_{v_j} apparaît dans τ ou τ'' ;
  – si $v \notin S$ et $j < j_{e,v}$: alors $d_{e,v,j-1}, d_{e,v,j}$ apparaissent dans τ_v, tandis que b_{v_j} apparaît dans τ_v (avec $v_j \neq v$) ;
  – si $v \notin S$ et $j = j_{e,v}$: alors $d_{e,v,j-1}$ apparaît dans τ, $d_{e,v,j}$ apparaît dans τ', et b_{v_j} apparaît dans τ'' (on a $v_j \in S$ puisque $j = j_{e,v}$) ;
  – si $v \notin S$ et $j > j_{e,v}$: alors $d_{e,v,j-1}, d_{e,v,j}$ apparaissent dans τ', et b_{v_j} apparaît dans τ ou τ'.
– triplets définis par la condition (iv) : soit $m = m_e$, on montre que $d_{e,v,m-1} a | b_{v_m}$ est consistant avec T, en considérant trois sous-cas :
  – si $v \in S$: alors $d_{e,v,m-1}, a$ apparaissent dans τ', tandis que b_{v_m} apparaît dans τ ou τ'' ;

– si $v \notin S$ et $j_{v,e} < m$: alors $d_{e,v,m-1}, a$ apparaissent dans τ', tandis que b_{v_m} apparaît dans τ ou τ'' ;
– si $v \notin S$ et $j_{v,e} = m$: alors $d_{e,v,m-1}$ apparaît dans τ, a apparaît dans τ', et b_{v_m} apparaît dans τ'' (on a $v_m \in S$ puisque $j_{v,e} = m$).

On conclut que $m(\mathcal{R}, T) = |S|$ comme annoncé. □

La construction utilisée dans la preuve du Lemme 6.26 est illustrée ci-dessous.

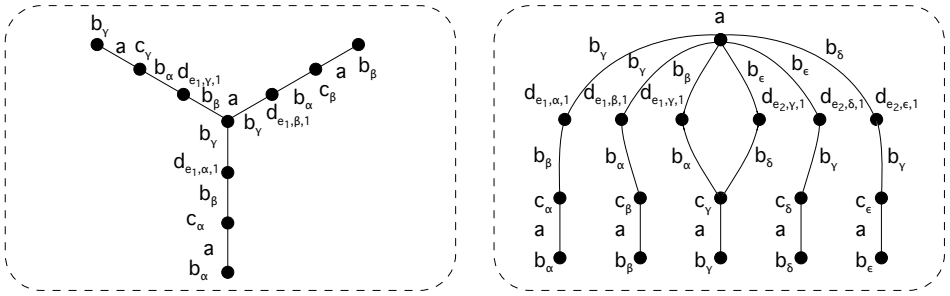


FIG. 6.3 – Les graphes de Aho correspondant au gadget construit pour $H = (\{\alpha, \beta, \gamma, \delta, \epsilon\}, \{e_1, e_2\})$, $e_1 = \{\alpha, \beta, \gamma\}$ and $e_2 = \{\gamma, \delta, \epsilon\}$.

Lemme 6.27. *Il existe une réduction préservant la mesure de* MTIW *à* MTI.

Démonstration. Etant donné une collection de triplets pondérée $\mathcal{R}$ sur L, on construit une collection de triplets non pondérée $\mathcal{R}'$ sur L', de la manière suivante. L'ensemble d'étiquettes l' est obtenu à partir de L en ajoutant des étiquettes t_i pour chaque $t \in T(L), 1 \leq i \leq \mathcal{R}(t)$. L'ensemble $\mathcal{R}'$ contient les triplets $xt_i|z, yt_i|z$ pour chaque $t = xy|z \in T(L), 1 \leq i \leq \mathcal{R}(t)$.

Les deux faits ci-dessous prouvent qu'on a bien une réduction préservant la mesure.

Fait 1. Soit un arbre binaire T sur L, on peut construire en temps polynomial un arbre binaire T' sur L' t.q. $m(\mathcal{R}', T') = m(\mathcal{R}, T)$.

Preuve. Soit $<$ un ordre total arbitraire sur L. On démarre avec L et on construit T' comme suit : pour chaque triplet $t = xy|z \in T(L)$ avec $x < y$, pour chaque $1 \leq i \leq \mathcal{R}(t)$, on insère t_i comme frère de x. Montrons que $m(\mathcal{R}', T') = m(\mathcal{R}, T)$. En effet, considérons $t = xy|z \in T(L)$ avec $x < y$, alors : (i) si $xy|z \in rt(T)$, alors pour chaque $1 \leq i \leq \mathcal{T}(t)$, les triplets $xt_i|z$ et $yt_i|z$ sont dans $rt(T')$, donc la contribution de ces triplets à $m(\mathcal{R}', T')$ est 0 ; (ii) si $xz|y \in rt(T)$, alors pour chaque $1 \leq i \leq \mathcal{R}(t), xt_i|z \in rt(T')$ mais $yt_i|z \notin rt(T')$, donc la contribution de ces triplets à $m(\mathcal{R}', T')$ est égale à $\mathcal{R}(t)$;

(iii) si $yz|x \in rt(T)$, le raisonnement est similaire.

Fait 2. Soit un arbre binaire T' sur L', on peut construire en temps polynomial un arbre binaire T sur L t.q. $m(\mathcal{R}, T) \leq m(\mathcal{R}', T')$.

Preuve. On monre que $T = T'|L$ est tel que $m(\mathcal{R}, T) \leq m(\mathcal{R}', T')$. En effet, considérons un triplet $t = xy|z \in T(L)\backslash rt(T)$. S'il existe un i tel que $xt_i|z \in rt(T')$ et $yt_i|z \in rt(T')$, on obtient $xy|z \in rt(T')$, impossible. Il s'ensuit que pour chaque $1 \leq i \leq \mathcal{R}(t)$, l'un des triplets $xt_i|z, yt_i|z$ n'est pas dans $\mathcal{R}'$, et la contribution de ces triplets à $m(\mathcal{R}', T')$ est donc $\geq \mathcal{R}(t)$. □

Preuve de la Proposiion 6.25. On déduit des lemmes 6.26 et 6.27 l'existence d'une réduction linéaire de HS à Mti. On conclut par les résultats d'inapproximabilité pour HS dûs à [9]. □

Notons que la réduction précédente prouve également :

Proposition 6.28. *Le problème* Mti *est* W[2]*-dur.*

Difficulté de Mli

On peut montrer les résultats suivants, d'une manière analogue à la preuve de la Proposition 6.25.

Proposition 6.29. *Sous l'hypothèse* P $\neq$ NP*,* Mli *n'est pas approximable à* $\Omega(\log n)$.

Proposition 6.30. *Le problème* Mli *est* W[2]*-dur.*

6.3.4 Remarques

Terminons par quelques questions ouvertes. Le problème Mtic admet-il une approximation à un facteur constant ? un schéma d'approximation polynomial ? Notons l'analogie avec le problème Directed Feedback Arc Set sur les tournois, pour lequel il existe plusieurs approximations à un facteur constant [8, 3, 16], et un PTAS [15]. Une seconde question concerne l'approximabilité du problème Mti : est-il approximable à un facteur $\Omega(\log^c n)$? Le meilleur ratio connu pour ce problème est $n - 1$ [11].

6.4 Bibliographie

[1] Aho (A. V.), Sagiv (Y.), Szymanski (T. G.) et Ullman (J. D.). – Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, vol. 10, n° 3, 1981, pp. 405–421.

[2] Ailon (N.) et Alon (N.). – Hardness of fully dense problems. *Information and Computation*, vol. 205, n° 8, 2007, pp. 1117–1129.

[3] Ailon (N.), Charikar (M.) et Newman (A.). – Aggregating inconsistent information : ranking and clustering. *In : Proceedings of STOC 2005*, pp. 684–693.

[4] Alon (N.). – Ranking Tournaments. *SIAM Journal of Discrete Mathematics*, vol. 20, n° 1, 2006, pp. 137–142.

[5] Bryant (D.). – *Building trees, hunting for trees and comparing trees : theory and method in phylogenetic analysis*. – Thèse de doctorat, University of Canterbury, Department of Mathemathics, 1997.

[6] Byrka (J.), Guillemot (S.) et Jansson (J.). – New Results on Optimizing Rooted Triplets Consistency. *In : Proceedings of ISAAC 2008*, pp. 484–495.

[7] Chor (B.), Hendy (M.) et Penny (S.). – Analytic solutions for three-taxon mlm c trees with variable rates across sites. *In : Proceedings of WABI 2001*, pp. 204–213.

[8] Coppersmith (D.), Fleischer (L.) et Rudra (A.). – Ordering by weighted number of wins gives a good ranking for weighted tournaments. *In : Proceedings of SODA 2006*, pp. 776–782.

[9] Feige (U.). – A Threshold of $\ln n$ for Approximating Set Cover. *Journal of the ACM*, vol. 45, n° 4, 1998, pp. 634–652.

[10] Fernau (H.). – Parameterized algorithmics : A graph-theoretic approach. – 2005. Habilitationsschrift, Universität Tübingen, Germany.

[11] Gasienec (L.), Jansson (J.), Lingas (A.) et Ostlin (A.). – On the Complexity of Constructing Evolutionary Trees. *Journal of Combinatorial Optimization*, vol. 3, n° 2–3, 1997, pp. 183–197.

[12] Jansson (J.), Lingas (A.) et Lundell (E.M.). – A Triplet Approach to Approximations of Evolutionary Tree. Poster H15 présenté à RECOMB 2004.

[13] Jiang (T.), Kearney (P.E.) et Li (M.). – A Polynomial Time Approximation Scheme for Inferring Evolutionary Trees from Quartet Topologies and Its Application. *SIAM Journal on Computing*, vol. 30, n° 6, 2000, pp. 1942–1961.

[14] Kannan (S.), Lawler (E.) et Warnow (T.). – Determining the evolutionary tree using experiments. *Journal of Algorithms*, vol. 21, n° 1, 1996, pp. 26–50.

[15] Kenyon-Mathieu (C.) et Schudy (W.). – How to rank with few errors. *In : Proceedings of STOC 2007*, pp. 95–103.

[16] Zuylen (A. Van), Hegde (R.), Jain (K.) et Williamson (D.P.). – Deterministic pivoting algorithms for constrained ranking and clustering problems. *In : Proceedings of SODA 2007*, pp. 405–414.

Titre : Approches combinatoires pour le consensus d'arbres et de séquences.

Résumé : Cette thèse étudie d'un point de vue algorithmique diverses méthodes de consensus portant sur des collections d'objets étiquetés. Les problèmes étudiés impliquent des objets étiquetés sans répétition d'étiquettes ; ces objets peuvent être des arbres enracinés ou des séquences, avec des applications à la bioinformatique. Ainsi, les problèmes sur les arbres considérés dans cette thèse peuvent trouver des applications pour l'estimation de congruence entre phylogénies, pour la construction de superarbres, et pour l'identification de transferts horizontaux de gènes. Pour leur part, les problèmes sur les séquences considérés dans cette thèse ont des applications potentielles pour le calcul de distance génomique basé sur les ordres de gènes. De manière générale, ce travail met à profit les théories de la complexité paramétrique et de l'approximabilité pour obtenir des algorithmes et des résultats de difficulté pour les problèmes étudiés.

Mots-clés : méthodes de consensus, algorithmique, complexité calculatoire, complexité paramétrique, algorithmes paramétrés, algorithmes d'approximation, bioinformatique.

Title : Combinatorial approaches for the consensus of trees and sequences.

Abstract : This thesis studies from an algorithmic point of view various consensus methods on collections of labeled objects. The problems under study involve labeled objects without repetition of labels ; these objects may be rooted trees or sequences, with applications to bioinformatics. For instance, the problems on trees considered in this thesis may find applications to the estimation of congruence between phylogenies, the construction of supertrees, and the identification of horizontal gene transfers. For their part, the problems on sequences considered in this thesis have potential applications for the computation of genomic distances based on gene orders. Overall, this work relies on the theories of parameterized complexity and approximability to obtain algorithms and hardness results for the problems studied.

Keywords : consensus methods, algorithmics, computational complexity, parameterized complexity, parameterized algorithms, approximation algorithms, bioinformatics.

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, CNRS-UMR 5506, 161 rue Ada, 34392 Montpellier - Cedex 5.