Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques

Guillaume Bagan

UNIVERSITÉ de CAEN/BASSE-NORMANDIE

U.F.R. : Sciences

ÉCOLE DOCTORALE : SIMEM

THÈSE

présentée par

Guillaume Bagan

et soutenue

le 2 Mars 2009

en vue de l'obtention du

DOCTORAT de l'UNIVERSITÉ de CAEN

spécialité : Informatique

(Arrêté du 7 août 2006)

Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques

MEMBRES du JURY

Bruno Courcelle	Professeur	Université Bordeaux 1	(rapporteur)
Georg Gottlob	Professeur	Université d'Oxford	(rapporteur)
Luc Segoufin	Directeur de recherche INRIA	ENS Cachan	(rapporteur)
Michel Habib	Professeur	Université Paris 7	
Joachim Niehren	Directeur de recherche INRIA	Université Lille 1	
Etienne Grandjean	Professeur	Université de Caen	(directeur)

Mis en page avec la classe thloria.

Table des matières

Table des figures

Liste des tableaux

Introduction

Cette thèse porte sur les problèmes d'énumération. Il s'agit de générer l'une après l'autre toutes les solutions d'un problème. La question de l'énumération des solutions intervient naturellement dans de nombreux domaines : combinatoire, problèmes logiques, requêtes dans les bases de données, etc. On peut citer les exemples combinatoires suivants : énumérer tous les arbres couvrants ou tous les couplages parfaits d'un graphe donné. Dans ce manuscrit, nous nous intéressons, non seulement au temps global du calcul de l'ensemble des solutions, mais aussi, plus précisément, au délai nécessaire pour produire chaque solution. Par exemple, Uno [Uno01] donne un algorithme qui énumère à délai constant tous les arbres couvrants d'un graphe donné après un prétraitement en temps linéaire en la taille du graphe. De même, Shioura et al. [STU97] prouvent qu'on peut énumérer tous les couplages parfaits avec un délai logarithmique en la taille du graphe, ceci si un couplage parfait est donné initialement.

Notre travail porte, de façon centrale, sur l'énumération des solutions d'une requête définie en logique. Il existe dans la littérature de nombreux résultats de complexité pour la génération de l'ensemble des solutions d'une requête [CM93, FFG02, Fri04, Yan81] mais on trouve peu de résultats concernant le délai de leur énumération. Dans cette thèse, nous cherchons à cerner la meilleure complexité des algorithmes d'énumération pour les requêtes logiques, ceci en fonction de la taille des structures ("data complexity"). Parfois, nous analysons également la complexité en fonction aussi de la taille de la formule ("combined complexity" : complexité combinée).

Le problème de l'énumération des solutions d'une requête logique est une extension du problème de la vérification d'un énoncé dans une structure ("model-checking"). On sait que le problème général de la vérification d'un énoncé de la logique du premier ordre (**FO**) ou du second ordre monadique (**MSO**) dans une structure quelconque, problèmes notés MC(**FO**) et MC(**MSO**), est PSPACE-complet ("combined complexity") dans les deux cas [Var82].

Pour surmonter cette difficulté, plusieurs auteurs ont étudié la complexité paramétrée du model-checking, le paramètre étant la taille de la formule (qu'on suppose être en général bien plus petite que la taille de la structure). Même dans ce cadre, Downey et al. [DFT96] ont démontré que la version paramétrée du problème MC(**FO**) est située très haut dans la hiérarchie des classes de complexité paramétrée. Au-delà, le problème de vérification MC(**MSO**) est difficile,

même pour un énoncé fixé, car un énoncé **MSO** peut exprimer un problème **NP**-complet, la 3-colorabilité par exemple. Plus encore, pour chaque niveau de la hiérarchie polynomiale $\Sigma_k P$, on peut trouver un énoncé **MSO** qui exprime un problème $\Sigma_k P$-complet. Pour rendre le problème du "model-checking" traitable, il est donc nécessaire de faire des restrictions. Il existe deux sortes de restrictions :

- des restrictions sur les formules ;
- des restrictions sur les structures.

Les *formules conjonctives* sont des formules **FO** très simples (formules **FO** n'utilisant que le quantificateur $\exists$ et le connecteur $\wedge$) qui sont fondamentales en bases de données et dans l'expression de problèmes de contraintes. On sait depuis 1977 [CM77] que le problème du model-checking des formules conjonctives reste **NP**-difficile. C'est pourquoi beaucoup de chercheurs en bases de données et en satisfaction de contraintes ont étudié des restrictions des formules conjonctives : Ils ont établi que la complexité du model-checking est polynomiale pour les classes de formules conjonctives suivantes :

- les formules conjonctives acycliques (Yannakakis [Yan81]) ;
- les formules conjonctives de largeur arborescente ("treewidth") bornée (Chekuri et Rajaraman [CR00]) ;
- les formules conjonctives de largeur hyperarborescente ("hypertreewidth") bornée (Gottlob et al. [GLS02]) ;
- les formules conjonctives de "component hypertreewidth" bornée (Gottlob et al. [GMS07]).

D'autre part, on connaît des classes de structures plus générales pour lesquelles le model-checking est **FPT** (le paramètre est toujours la taille de la formule). Ce sont :

- les formules conjonctives de largeur hyperarborescente généralisée ("hypertreewidth") bornée (Gottlob et al. [GLS02]) ;
- les formules conjonctives de largeur hyperarborescente fractionnelle ("fractional hypertreewidth") bornée (Grohe et Marx [GM06]).

Par ailleurs, si au lieu de restreindre la classe des formules autorisées, on restreint la classe des structures, alors, pour tout énoncé **FO** ou **MSO**, le problème du model-checking devient **FPT** dans les cas suivants :

Pour **MSO** :

- sur les arbres (Thatcher et Wright [TW68]) ;
- sur les graphes (ou les structures) de largeur arborescente bornée (Courcelle [Cou90b, Cou92]) ;
- sur les graphes de largeur de clique bornée (combinaison des résultats de Courcelle, Makowsky et Rotics [CMR00] et d'Oum [Oum05, HO07]).

Pour **FO** :

- sur les structures de degré borné (Seese [See96]) ;

– sur les structures de “faible degré” (“low degree”) (Grohe [Gro01]) ;
– sur les structures de largeur arborescente localement bornée (Frick et Grohe [FG01b]) ;
– sur les graphes avec un mineur exclu (Flum et Grohe [FG01a]) ;
– sur les graphes avec un mineur localement exclu (Dawar, Grohe et Kreutzer [DGK07]).

Il existe peu de travaux en complexité algorithmique qui étudient les problèmes d'énumération. Jusqu'à une date récente, tous ne portaient que sur la complexité polynomiale. Dans ce domaine, Johnson, Papadimitriou et Yannakakis [PY99] ont introduit les trois classes de complexité polynomiale emboîtées DelayP $\subseteq$ IncP $\subseteq$ TotalP.

– DelayP est la classe des problèmes énumérables avec un délai (entre deux solutions) polynomial en la taille de l'entrée ;
– IncP (polynomial incrémental) est la classe des problèmes énumérables avec un délai polynomial en la somme des tailles de l'entrée et des solutions précédemment énumérées ;
– TotalP est la classe des problèmes dont le temps total d'énumération est polynomial en la taille de l'entrée + la taille de la sortie.

Dans ce mémoire, on définit des classes de complexité plus fines, voire minimales, pour les problèmes d'énumération : par exemple, un problème d'énumération est dit CONSTANT-DELAY$_{lin}$ si on peut énumérer les solutions du problème, pour une entrée I, à *délai constant*, après un *précalcul linéaire* en la taille de I. En conséquence, le temps total de l'algorithme d'énumération est *linéaire* en la taille cumulée de l'entrée et de l'ensemble des solutions.

De façon plus générale, on élabore des algorithmes d'énumération agissant en deux phases successives :

– une *phase de prétraitement* (précalcul) dont le temps est fonction de la taille de l'entrée I seulement ;
– une *phase d'énumération* qui utilise la sortie du prétraitement et dont le délai entre deux solutions dépend essentiellement de la taille de la solution attendue et ne dépend que faiblement ou pas du tout de celle de l'entrée I.

Pour plusieurs problèmes, on étudie une question plus précise encore que l'énumération des solutions : la génération de la j-ème solution, selon un ordre donné. Là encore, le calcul se décomposera en deux phases :

– une *phase de prétraitement* comme pour l'énumération ; cette phase est indépendante du rang j ;
– une phase dite *phase dynamique* (le calcul, proprement dit) où l'on lit j et calcule la j-ème solution ; cette phase est en général de temps petit.

Un tel algorithme peut être utilisé pour énumérer toutes les solutions du problème en exécutant le précalcul, puis en itérant seulement la phase de calcul de la première solution, de la seconde, etc, donc à petit délai.

Principaux sujets étudiés dans cette thèse

Nos préliminaires fixent un cadre formel où nous inscrivons l'ensemble de nos contributions. Nous fixons d'abord très précisément notre modèle de calcul avec ses entrées/sorties, ses mesures de complexité en temps et en espace. A côté des notions classiques de complexité, nous introduisons notamment de nouvelles classes de complexité ; ces classes que nous nous sommes efforcés de rendre aussi naturelles et générales que possible, constitueront, tout au long de la thèse, un cadre précis et uniforme pour la présentation de l'ensemble de nos résultats de complexité. On y rappelle aussi un ensemble de notions et de résultats combinatoires sur les graphes, les hypergraphes et leurs décompositions arborescentes, etc. Enfin, on présente les quatre types de problèmes logiques sur lesquels porte la thèse : le model-checking, le comptage des solutions, leur énumération et la recherche de la j-ème solution, ainsi que les outils logiques qui seront utilisés pour étudier leur complexité, tout particulièrement les notions de traduction et d'interprétation (**FO** ou **MSO**), linéaires ou polynomiales.

Le chapitre 2 est consacré à l'étude des formules conjonctives et, tout particulièrement, l'étude des formules conjonctives acycliques (appelées ici formules **ACQ**). Yannakakis [Yan81] a montré qu'on peut énumérer les solutions d'une requête acyclique φ sur une structure quelconque $\mathcal{M}$ en temps total $O(|\varphi| \cdot |\mathcal{M}| \cdot |\varphi(\mathcal{M})|)$. Par la suite, Papadimitriou et Yannakakis [PY99] ont élargi cette notion en introduisant les formules conjonctives acycliques avec inégalités, c'est-à-dire des formules conjonctives acycliques où on autorise des inégalités $\neq$ (formules $\mathbf{ACQ}^{\neq}$). Ils ont prouvé qu'on peut générer toutes les solutions d'une requête $\mathbf{ACQ}^{\neq}$ en temps total $f(|\varphi|)\cdot|\mathcal{M}|\cdot|\varphi(\mathcal{M})|\cdot\log^2(|\mathcal{M}|)$, où f est une fonction exponentielle. Nous améliorons ce résultat en établissant que l'on peut générer toutes les solutions avec un délai $f'(|\varphi|)\cdot|\mathcal{M}|$ (pour une fonction exponentielle f') et donc en temps total $f'(|\varphi|)\cdot|\mathcal{M}|\cdot|\varphi(\mathcal{M})|$. Nous démontrons ce résultat par une forme d'élimination des quantificateurs sur les formules acycliques avec inégalités. Nous cherchons ensuite à caractériser les requêtes (avec ou sans inégalités) qui peuvent être énumérées, à délai constant, après un prétraitement linéaire (en la taille du modèle). Nous montrons que c'est le cas pour une classe de formules que nous introduisons : les formules *conjonctives connexe-acycliques*. Mieux, nous établissons que ce résultat est maximal dans un certain sens. Nous prouvons, sous une hypothèse de complexité, que les solutions d'une requête acyclique peuvent être énumérées, à délai constant, après précalcul linéaire si et seulement si cette requête est connexe-acyclique. Enfin, nous introduisons la notion de *largeur arborescente connexe* d'une formule qui est une variante de la notion de largeur arborescente. Chekuri et Rajaraman [CR00] ont prouvé qu'une requête conjonctive φ de largeur arborescente k peut être évaluée sur une structure $\mathcal{M}$ en temps $O(|\varphi| \cdot |\mathrm{Dom}(\mathcal{M})|^{k+1} \cdot |\varphi(\mathcal{M})|)$. Nous démontrons qu'une requête conjonctive φ fixée (avec ou sans inégalités) de largeur arborescente connexe k peut être évaluée sur une structure $\mathcal{M}$, à délai constant, après un prétraitement en temps $O(|\mathrm{Dom}(\mathcal{M})|^{k+1})$.

Le chapitre 3 est dédié à la complexité de l'évaluation des requêtes **MSO** sur les structures de *largeur arborescente bornée*. La largeur arborescente est un invariant introduit par Robertson et Seymour [RS84]. Un des principaux intérêts de la largeur arborescente est que de nombreux problèmes qui sont **NP**-complets dans le cas général deviennent polynomiaux, voire linéaires, lorsque l'on se restreint à des classes de graphes (ou de structures) de largeur arborescente bornée. Il s'avère en fait que de nombreuses classes de graphes rencontrées dans des applications (par exemple [LS88, Mö90, Tuz92]) ont une petite largeur arborescente. Dans le domaine de la logique, beaucoup d'études ont été conduites sur l'évaluation de requêtes **MSO** pour des classes de structures de largeur arborescente bornée. Tout d'abord, Thatcher et Wright [TW68] ont établi qu'on peut vérifier en temps linéaire si un énoncé **MSO** fixé est satisfait par un arbre donné. Courcelle [Cou90b, Cou92] a généralisé ce résultat aux classes de structures de largeur arborescente bornée. Plus tard, Arnborg, Largargren et Seese [ALS91] ont prouvé que l'on peut calculer en temps linéaire le nombre de solutions d'une requête **MSO** sur une structure de largeur arborescente bornée. Par ailleurs, Courcelle et Mosbah [CM93] ont démontré que l'on peut générer l'ensemble des solutions d'une telle requête en temps total polynomial en la taille cumulée de la structure et de la sortie. Enfin, Flum, Frick et Grohe [FFG02] ont précisé ce résultat en réduisant la complexité à un temps linéaire en la taille cumulée de la structure et de la sortie. Tous ces résultats se ramènent aux arbres binaires et s'appuient sur les automates d'arbres (déterministes bottom-up). Avec ces mêmes outils, nous montrons que l'on peut générer toutes les solutions d'une formule **MSO** sur une structure $\mathcal{M}$ de largeur arborescente bornée après un prétraitement linéaire en la taille de $\mathcal{M}$ et un délai (entre deux solutions consécutives) linéaire en la taille de la seconde solution. Nous prouvons également que l'on peut générer directement la j-ème solution (dans un certain ordre) après un prétraitement toujours linéaire et par un calcul en temps $O(|S| \times \log |\mathcal{M}|)$ où S est la solution demandée et $\mathcal{M}$ la structure. Remarquons que ces résultats ont été obtenus indépendamment par Courcelle [Cou06], à la différence que ses algorithmes d'énumération et de recherche du j-ème élément possèdent une phase de prétraitement qui n'est pas linéaire mais en temps $O(|\mathcal{M}| \times \log(|\mathcal{M}|)$.

Le chapitre 4 est consacré à l'évaluation des requêtes **FO** sur les structures de *degré borné*. Le premier résultat connu dans le domaine est celui de Seese [See96] qui montre que l'on peut décider en temps linéaire si un énoncé **FO** φ fixé est satisfait par une structure de degré au plus d (d fixé). Récemment, Durand et Grandjean [DG07] ont amélioré ce résultat en établissant que, pour toute formule **FO** fixée, on peut générer, à délai constant, l'ensemble de ses solutions sur une structure de degré borné après un précalcul linéaire. Nous précisons ici ces résultats en démontrant, d'une part, que l'on peut énumérer, à délai constant, l'ensemble des solutions d'une telle requête dans l'ordre lexicographique, et, d'autre part, que l'on peut trouver la j-ème solution de la requête en temps constant après un prétraitement linéaire en la taille de la structure.

Enfin, au chapitre 5, nous nous intéressons à la largeur de clique locale : cette notion "étend"

les deux notions de largeur de clique et de largeur arborescente locale (notion qui elle-même généralise le degré borné). La première question que nous résolvons est celle d'exhiber une classe naturelle de graphes qui soit de largeur de clique localement bornée mais qui ne soit ni de largeur de clique bornée ni de largeur arborescente localement bornée. Nous prouvons que les graphes d'intervalles unitaires satisfont ces trois conditions. En nous appuyant sur ce résultat et en adaptant des concepts et résultats de Frick et Grohe [FG01b], nous établissons que toute requête du premier ordre fixée est évaluée sur un graphe d'intervalles unitaires en temps linéaire en la taille cumulée de l'entrée et de la sortie. Enfin, nous prouvons que ce résultat ne peut se généraliser ni aux requêtes **MSO**, ni aux graphes d'intervalles quelconques.

Pour l'essentiel, les résultats du chapitre 2 ont fait l'objet d'une première publication dans [BDG07] et font l'objet d'un article étendu à soumettre où ils sont exposés plus complètement. Notre article de conférence [Bag06] fournit une première version des théorèmes du chapitre 3 : les notions utilisées, les résultats et leurs preuves sont ici développés de façon plus détaillée et plus précise. Le chapitre 4 est, à quelques détails près, une copie de l'article de revue [BDGO08]. Enfin, le chapitre 5 sera prochainement soumis pour publication.

Chapitre 1

Préliminaires

Sommaire

Le présent chapitre présente des outils généraux, en combinatoire, logique et complexité, et qui sont nouveaux pour quelques uns, en complexité tout particulièrement. Ces outils seront utilisés dans tous les chapitres de ce mémoire. (Par ailleurs, on donnera pour chaque chapitre des préliminaires spécifiques.) Avec les définitions des notions introduites et leurs notations, nous rappelons ici quelques unes des propriétés des objets introduits et nous démontrons aussi quelques résultats qui les mettent en oeuvre. C'est le cas surtout quand il s'agit de notions nouvelles : typiquement, en complexité, les classes des problèmes dits "dynamiques" $dyn(f, g)$ et surtout les classes de complexité d'énumération $enum(f, g)$. De façon générale, la présentation des notions de complexité et des résultats de base qui leur sont associés constitue la majeure partie de ces préliminaires.

1.1 Notations générales

On utilisera en général les notations standard. On précise en particulier les notations suivantes qu'on utilisera tout au long de la thèse.

Définition 1 (cardinal, ensemble des parties, union disjointe) *Soit E un ensemble. On note $card(E)$ la cardinalité de E. On désigne par $\mathcal{P}(E)$ l'ensemble des parties de E et par $\mathcal{P}_k(E)$ l'ensemble des parties de E de cardinalité k. Parmi les opérations ensemblistes usuelles, on utilisera tout particulièrement le produit cartésien, noté $A \times B$, de deux ensembles A et B et l'union disjointe, notée $A \uplus B$, de deux ensembles disjoints A et B.*

Définition 2 (notations O et O_k) *Soient des fonctions $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$. On écrit $f(n) = O(g(n))$ pour exprimer qu'il existe une constante c telle que, pour tout $n \in \mathbb{N}$, on a $f(n) \leq c \cdot g(n)$.*

Soient des fonctions $f : \mathbb{N}^2 \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$. On écrit $f(k, n) = O_k(g(n))$ pour exprimer qu'il existe une fonction $h : \mathbb{N} \to \mathbb{N}$ telle que, pour tout $k, n \in \mathbb{N}$, on a $f(k, n) \leq h(k) \cdot g(n)$. Cette notion se généralise à plusieurs paramètres $k_1, \ldots, k_p$ au lieu du seul paramètre k.

$\log x$ désigne le logarithme de x en base 2. On utilisera les intervalles entiers $[m, n] = \{i \in \mathbb{N} : m \leq i \leq n\}$ et $[m] = [1, m]$.

1.2 Graphes, hypergraphes et leurs décompositions

1.2.1 Généralités

On utilise les notions usuelles sur les graphes, souvent généralisées ici aux hypergraphes (voir par exemple [Ber73] ou [Die05]).

Définition 3 (hypergraphe, sommets, arêtes) *Un hypergraphe H est un couple (V, E) tel que V est un ensemble fini et E est un ensemble de parties non vides de V. Un élément de V est appelé sommet de H et un élément de E est appelé hyperarête de H. On notera souvent $V = V(H)$ et $E = E(H)$.*

Remarque 1 *Dans ce manuscrit, on supposera que, pour tout hypergraphe $H = (V, E)$, on a $\bigcup_{e \in E} e = V$, autrement dit tout sommet appartient à au moins une hyperarête.*

Définition 4 (graphe, arêtes) *Un graphe est un hypergraphe où toutes les hyperarêtes sont de cardinalité 2. Les hyperarêtes d'un graphe sont appelées* arêtes.

Définition 5 (voisin, voisinage, voisinage fermé, jumeaux) *Soit $H = (V, E)$ un hypergraphe. Deux sommets distincts x et y de V sont dits voisins s'il existe une hyperarête $e \in E$ qui contient x et y. Le voisinage d'un sommet $x \in V$, noté $N_H(x)$ (ou simplement $N(x)$), est l'ensemble des sommets $y \in V$, distincts de x et voisins de x. On appelle $N_H[x] = N_H(x) \cup \{x\}$ le voisinage fermé de x. Deux sommets x et y sont dits* jumeaux *si on a $N_H(x) - \{x, y\} = N_H(y) - \{x, y\}$.*

Définition 6 (degré) *Soit $H = (V, E)$ un hypergraphe. Soit x un sommet de H, le degré de x, noté $\deg_H(x)$ (ou plus simplement $\deg(x)$), est le nombre d'hyperarêtes $e \in E$ contenant x. Le degré d'un hypergraphe H est le degré maximal des sommets de H : $\deg(H) = \max\{\deg_H(x) : x \in V\}$.*

Définition 7 (hypergraphe induit, sous-hypergraphe) *Soit $H = (V, E)$ un hypergraphe et une partie $A \subseteq V$. L'hypergraphe de H induit par A, noté $H[A]$, est l'hypergraphe*

$$H[A] = (A, \{e \cap A : e \in E \text{ et } e \cap A \neq \emptyset\})$$

$H' = (V', E')$ est un sous-hypergraphe de H si $V' \subseteq V$ et $E' \subseteq E$.

Définition 8 (chemin, chemin minimal) *Un chemin de longueur k entre deux sommets distincts x et y d'un hypergraphe $H = (V, E)$ est un tuple $(x_0, \ldots, x_k)$ formé de sommets tous distincts avec $x = x_0$ et $y = x_k$ et tel que, pour tout $i < k$, il existe une hyperarête $e \in E$ qui contient x_i et x_{i+1}. Un chemin $(x_0, \ldots, x_k)$ est dit* minimal *si, pour tous les indices i, j tels que $i + 1 < j$, les sommets x_i et x_j ne sont pas voisins.*

Définition 9 (connexité, composantes connexes) *Soit $H = (V, E)$ un hypergraphe. Les composantes connexes de H sont les parties A de V maximales pour l'inclusion, telles que, pour tous $x, y \in A$, il existe un chemin dans H entre x et y.*

H est dit connexe s'il admet une seule composante connexe.

Définition 10 (distance, diamètre, boule) *Soient $H = (V, E)$ un hypergraphe et x, y deux sommets de H appartenant à la même composante connexe. Alors la distance entre x et y, notée $dist_H(x, y)$ (ou plus simplement $dist(x, y)$), est la longueur minimale d'un chemin entre x et y.*

Soit $H = (V, E)$ un hypergraphe connexe. Le diamètre de H, noté $diam(H)$, est la plus grande distance entre deux sommets de H.

Soient $H = (V, E)$ un hypergraphe, x un sommet de H et un entier $k \geq 0$. La boule de centre x et de rayon k, notée $N_H^k(x)$ (ou plus simplement $N^k(x)$), est l'ensemble $\{y \in V : dist_H(x, y) \leq k\}$.

Il existe plusieurs définitions de la notion de cycle dans un hypergraphe. Nous utiliserons ici une notion de cycle qui généralise celle de cycle dans les graphes.

Définition 11 (cycle) *Dans un hypergraphe $H = (V, E)$, un cycle de longueur k est un tuple $P = (x_1, \ldots, x_k)$ de sommets tous distincts tel que*

1. *pour tout indice $i \in [k-1]$, il existe une hyperarête $e \in E$ qui contient x_i et x_{i+1} ;*
2. *il existe une hyperarête $e \in E$ qui contient x_1 et x_k ;*
3. *aucun triplet de sommets de P n'est inclus dans une hyperarête de H ;*
4. *aucune paire de sommets non consécutifs dans P n'est incluse dans une hyperarête de H.*

Remarque 2 *Nous avons défini le concept de cycle d'un hypergraphe de façon à obtenir l'équivalence suivante : $P = (x_1, \ldots, x_k)$ est un cycle de l'hypergraphe H si et seulement si l'hypergraphe induit $H[P]$ est un graphe-cycle $(a_1, \ldots, a_k)$, si on élimine les hyperarêtes singletons.*

Définition 12 (clique) *On appelle clique d'un hypergraphe $H = (V, E)$ un ensemble $C \subseteq V$ formé de sommets deux à deux voisins.*

La notion de graphe induit est légèrement différente de celle d'hypergraphe induit : la seule différence est qu'un hypergraphe induit peut contenir des arêtes singletons.

Définition 13 (graphe induit) *Soient $G = (V, E)$ un graphe et S une partie de V. Le graphe induit de G par S, noté $G[S]$, est le graphe (S, E_S) où $E_S = \{\{x, y\} \in E : (x, y) \in S^2\}$.*

1.2.2 Arbres

On utilise les définitions et notations usuelles sur les arbres.

Définition 14 (forêt, arbre, noeuds) *Une forêt est un graphe sans cycle. Un arbre est une forêt connexe. Les sommets d'un arbre ou d'une forêt sont appelés noeuds.*

Définition 15 (arbre enraciné, racine, arc, parent, enfant, noeud interne, feuille) *Un arbre enraciné est un arbre muni d'un noeud choisi, appelé racine de l'arbre. Le choix d'une racine r détermine une orientation de chaque arête, à partir de la racine r : chaque arête devient un arc (x, y) ; on dira alors que x est parent de y et que y est enfant de x.*

Un noeud est interne s'il a au moins un enfant. Sinon on dit que c'est une feuille.

Définition 16 (chemin orienté, ancêtre, descendant) *Dans un arbre enraciné, un chemin orienté de x à y est une suite $(x_1 = x, x_2, \ldots, x_k = y)$ tel que chaque couple (x_i, x_{i+1}) est un arc de l'arbre enraciné. On dit alors que x est ancêtre de y et que y est descendant de x.*

1.2.3 Décompositions

Nous utiliserons dans cette thèse les diverses notions de décomposition et de largeur arborescentes (d'un graphe ou d'un hypergraphe) que l'on trouve dans la littérature, notions que nous appliquerons d'une part aux structures, d'autre part aux formules (en fait, à l'hypergraphe associé à chaque formule conjonctive).

Décomposition arborescente d'un hypergraphe

La notion de largeur arborescente a été introduite par Robertson et Seymour [RS84]. Pour plus d'informations, on pourra se référer à [Bod93]. Intuitivement, plus petite est la largeur arborescente d'un graphe ou d'un hypergraphe, plus il ressemble à un arbre.

Définition 17 (décomposition arborescente, sac, largeur d'une décomposition) *On appelle décomposition arborescente d'un hypergraphe $H = (V, E)$ un couple $(T, (B_t)_{t \in V(T)})$ où T est un arbre et qui possède les propriétés suivantes.*

1. *Pour chaque noeud t de T, l'élément associé B_t, appelé sac (en anglais "bag"), est un ensemble de sommets de H, $B_t \subseteq V$, et on a $\bigcup_{t \in V(T)} B_t = V$;*
2. *Chaque hyperarête de H est incluse dans un sac : $\forall e \in E \; \exists t \in V(T) \; e \subseteq B_t$;*
3. *T possède la propriété de connexité : pour tout sommet $x \in V$, l'ensemble des noeuds t tels que $x \in B_t$ forme une partie connexe de T.*

On notera souvent $T = (T, (B_t)_{t \in V(T)})$ par abus de notation. On appelle largeur de la décomposition arborescente T, notée $width(T)$, le maximum des cardinalités des sacs de T moins 1 : $width(T) = \max_{t \in V(T)} card(B_t) - 1$.

Définition 18 (largeur arborescente) *On appelle largeur arborescente d'un hypergraphe H, notée $tw(H)$, la largeur minimale d'une décomposition arborescente T de H.*

Clique-décomposition d'un graphe

La largeur de clique est une notion qui généralise d'une certaine manière la notion de largeur arborescente. Cette généralisation a été introduite par Courcelle et al. [CER90] puis [CO00] et se définit par une construction inductive de graphes étiquetés. Parmi les classes de graphes connues ayant une largeur de clique bornée, on trouve les cliques, les cographes et les graphes distance-héréditaires [GR99].

Définition 19 (graphe étiqueté) *Un graphe k-étiqueté est un triplet $G' = (V, E, \lambda)$ constitué d'un graphe $G = (V, E)$ et d'une fonction d'étiquetage $\lambda : V \to [k]$.*

Définition 20 (classe CW_k) *CW_k est la classe minimale de graphes k-étiquetés close pour les opérations suivantes :*

- *une opération d'arité 0 : pour $i \in [k]$, $\bullet_i$ désigne le graphe constitué d'un unique sommet v_1 d'étiquette $\lambda(v_1) = i$;*
- *l'opération binaire $\uplus$: $G_1 \uplus G_2$ désigne l'union disjointe des graphes k-étiquetés G_1 et G_2 ;*
- *deux opérations unaires $\rho_{a,b}$ et $\eta_{a,b}$: pour des étiquettes distinctes $a, b \in [k]$ et pour un graphe k-étiqueté $G = (V, E, \lambda)$,*
  - *$\rho_{a,b}(G)$ désigne le graphe $G' = (V, E, \lambda')$ où on prend $\lambda'(x) = b$ si on a $\lambda(x) = a$ et $\lambda'(x) = \lambda(x)$ sinon.*
  - *$\eta_{a,b}(G)$ est le graphe $G' = (V, E', \lambda)$ où on prend $E' = E \cup \{\{x, y\} : (x, y) \in V^2$ et $\lambda(x) = a$ et $\lambda(y) = b\}$ (autrement dit, on ajoute les arcs du graphe biparti complet entre les sommets d'étiquette a et ceux d'étiquette b).*

Définition 21 (clique-décomposition) *On appelle clique-décomposition d'un graphe étiqueté $G' = (V, E, \lambda) \in CW_k$ le terme T (arbre binaire étiqueté) qui représente G' dans CW_k. On dira alors que le graphe $G = (V, E)$ (sous-jacent à $G' = (V, E, \lambda)$) possède la clique-décomposition T de largeur k.*

Définition 22 (largeur de clique) *La largeur de clique d'un graphe $G = (V, E)$, notée $cw(G)$, est la largeur minimale d'une clique-décomposition T de G.*

La largeur de clique est une généralisation de la largeur arborescente dans le sens suivant : il existe une fonction $f : \mathbb{N} \to \mathbb{N}$ telle que, pour tout entier k et tout graphe G, $\mathrm{tw}(G) \leq k$ implique $\mathrm{cw}(G) \leq f(k)$ [CO00] [CR05]. Il s'en suit que si $\mathcal{G}$ est une classe de graphes de largeur arborescente bornée (par k) alors $\mathcal{G}$ est une classe de graphes de largeur de clique bornée (par $f(k)$).

La réciproque est fausse : en effet, pour le graphe complet à k sommets K_k, on a $\mathrm{cw}(K_k) = 2$ et $\mathrm{tw}(K_k) = k - 1$.

Largeur arborescente locale

La largeur arborescente locale est une autre généralisation de la largeur arborescente. Cette notion a été introduite par Eppstein [Epp00] puis par Frick et Grohe [FG01b]. Intuitivement, une classe de graphes $\mathcal{G}$ a une largeur arborescente localement bornée si toute boule d'un graphe $G \in \mathcal{G}$ de rayon k a un diamètre qui est fonction de k.

Définition 23 (largeur arborescente localement bornée) *Une classe de graphes $\mathcal{G}$ est dite de largeur arborescente localement bornée s'il existe une fonction (calculable) f telle que, pour tout entier $k \geq 0$, pour tout graphe $G \in \mathcal{G}$ et tout sommet $x \in V(G)$, on a $tw(N_G^k(x)) \leq f(k)$.*

Parmi les classes de graphes ayant une largeur arborescente localement bornée, on trouve les graphes planaires, les graphes de degré borné ainsi que les graphes représentables sur une surface de genre borné [FG01b].

1.2.4 Hypergraphes acycliques

La notion d'hypergraphe acyclique dans ses diverses variantes (α-acyclique, β-acyclique, γ-acyclique) a été introduite et étudiée dans de nombreux articles ([Yan81, BFMY83, PY99]). Dans le chapitre 2 de cette thèse, nous étudierons, à la suite des articles de Yannakakis [Yan81] et Papadimitriou et al [PY99], la notion de requête conjonctive acyclique et ses généralisations. Pour les définir, on associera à chaque requête conjonctive un hypergraphe dont on exige qu'il soit acyclique. Il s'agit de la notion la plus générale d'acyclicité d'hypergraphe : l'α-acyclicité (voir par exemple [BFMY83]) qu'on définit comme suit :

Définition 24 (hypergraphe acyclique, structure arborescente) *Un hypergraphe $H = (V, E)$ est dit acyclique (α-acyclique) s'il possède une structure arborescente, c'est-à-dire un arbre T vérifiant les conditions suivantes :*

- *L'ensemble des noeuds de T est l'ensemble E des hyperarêtes de H : $V(T) = E$;*
- *T vérifie la propriété de connexité, c'est-à-dire que, pour tout sommet $x \in V$, l'ensemble des hyperarêtes qui contiennent x, $\{e \in E : x \in e\}$, forme un ensemble de noeuds connexe dans T.*

Nous donnons maintenant une caractérisation des hypergraphes acycliques, due à Beeri et al [BFMY83], qui nous sera utile dans le chapitre 2.

Définition 25 *Un hypergraphe H est dit conforme si toute clique de H est contenue dans une hyperarête.*

Théorème 1 *[BFMY83] Un hypergraphe H est acyclique si et seulement si il ne contient pas de cycle (selon notre définition de cycle) et est conforme.*

1.3 Logique : notions de base

On utilise les notions usuelles en logique et théorie des modèles finis (voir par exemple [EF99] ou [Lib04]) en les adaptant à nos besoins. On récapitule ici quelques notions et notations sans les définir de façon complète.

Définition 26 (signature, arité) *Une signature (notée généralement σ ou τ) est ici un ensemble fini de symboles de fonctions unaires et/ou de relations. A chaque symbole de relation R, encore appelé prédicat, est associé un entier appelé son* arité *et qu'on note $arity(R)$.*

Définition 27 (structure, structure relationnelle, structure unaire) *Une structure de signature σ ou σ-structure $\mathcal{M}$ est composée d'un domaine $D = Dom(\mathcal{M})$ constituant l'ensemble des éléments de la structure, d'une interprétation $R^{\mathcal{M}} \subseteq D^k$ de chaque symbole de relation R d'arité k de σ et d'une interprétation $f^{\mathcal{M}} : D \to D$ de chaque symbole de fonction unaire f de σ.*

Une structure (resp. signature) est dite relationnelle *si elle ne contient que (des symboles de) relations. Une structure* unaire *ou structure* fonctionnelle *(resp. signature unaire ou fonctionnelle) est une structure (resp. signature) qui ne contient que des (symboles de) fonctions et prédicats unaires.*

On utilisera fréquemment dans cette thèse la notion d'expansion d'une structure.

Définition 28 (expansion d'une structure) *Soit $\mathcal{M}$ une σ-structure et soit σ' une signature qui étend σ : $\sigma \subseteq \sigma'$. On dit qu'une σ'-structure $\mathcal{M}'$ étend $\mathcal{M}$ ou est une* expansion *de $\mathcal{M}$ si $\mathcal{M}$ est la restriction de $\mathcal{M}$ à la signature σ.*

Au chapitre 4, on s'intéressera aux *structures de degré borné*. La notion de degré d'une structure est une généralisation du degré d'un graphe.

Définition 29 (degré) *Dans une structure relationnelle $\mathcal{M}$, le degré d'un élément $x \in Dom(\mathcal{M})$, noté $deg_{\mathcal{M}}(x)$, est le nombre de k-tuples appartenant à une relation (d'arité k) de $\mathcal{M}$ et auxquels x appartient. Le degré d'une structure relationnelle $\mathcal{M}$ est le maximum des degrés de ses éléments : $deg(\mathcal{M}) = Max_{x \in Dom(\mathcal{M})} deg_{\mathcal{M}}(x)$.*

Définition 30 (logique du premier ordre, logique monadique du second ordre) *Pour une signature σ, l'ensemble des σ-formules du premier ordre est construit à partir d'un nombre infini de variables $x, y, z \ldots$ (dites variables du premier ordre), de prédicats (symboles de relations) $R \in \sigma$, du symbole d'égalité $=$, des symboles de fonctions $f \in \sigma$, des connecteurs $\vee$, $\wedge$, $\neg$, $\to$, $\leftrightarrow$ et des quantificateurs $\exists$ et $\forall$. Cette logique (ensemble de formules) est notée* **FO** *(First Order).*

On étend la logique du premier ordre par des variables (monadiques ou ensemblistes) du second ordre X, Y, $Z\ldots$ qui portent sur des parties du domaine de la structure. Pour une signature σ, la logique monadique du second ordre (notée **MSO** *: Monadic Second Order) est la clôture de* **FO** *par l'ajout de nouveaux atomes bâtis sur des variables (monadiques) du second ordre, c'est-à-dire de la forme X(y), où X est une variable du second ordre et y une variable du premier ordre, et de quantifications du second ordre $\exists X$ et $\forall X$ (où X est une variable du second ordre).*

On note var(φ) l'ensemble des variables d'une formule φ.

Définition 31 (variable libre, énoncé) *Une variable libre x (resp. X) d'une formule* **FO** *ou* **MSO** *φ est une variable qui a dans φ (au moins) une occurrence qui n'est pas "quantifiée", c'est-à-dire qui ne se trouve dans la portée d'aucun quantificateur $\exists x$ ou $\forall x$ (resp. $\exists X$ ou $\forall X$). On écrit $\varphi(X_1, \ldots, X_m, x_1, \ldots, x_n)$ pour indiquer que la liste des variables libres de la formule φ est $X_1, \ldots X_m, x_1, \ldots, x_n$. On note free($\varphi$) l'ensemble des variables libres d'une formule φ.*

Un énoncé est une formule ne possédant pas de variables libres.

Définition 32 (satisfaction) *Pour une σ-structure $\mathcal{M}$ de domaine D, une σ-formule $\varphi(X_1, \ldots, X_m, x_1, \ldots, x_n)$, des parties $A_1, \ldots, A_m \subseteq D$ et des éléments $a_1, \ldots, a_n \in D$, on note $(\mathcal{M}, A_1, \ldots, A_m, a_1, \ldots, a_n) \models \varphi(X_1, \ldots, X_m, x_1, \ldots, x_n)$ (ou $\mathcal{M} \models \varphi(A_1, \ldots, A_m, a_1, \ldots, a_n)$) pour dire que $\mathcal{M}$ satisfait φ si les variables $X_1, \ldots, X_m$ sont interprétées par les ensembles $A_1, \ldots A_m$ et les variables $x_1, \ldots, x_n$ sont interprétées par les éléments $a_1, \ldots a_n$.*

Définition 33 (requête logique) *On appelle résultat $\varphi(\mathcal{M})$ de la requête définie par une σ-formule $\varphi(\bar{X}, \bar{x})$ sur une σ-structure $\mathcal{M}$ de domaine D, l'ensemble*

$$\varphi(\mathcal{M}) = \{(\bar{A}, \bar{a}) \in \mathcal{P}(D)^m \times D^n : (\mathcal{M}, \bar{A}, \bar{a}) \models \varphi(\bar{X}, \bar{x})\}$$

. Les éléments de l'ensemble $\varphi(\mathcal{M})$ sont appelés les solutions de la requête φ sur la structure $\mathcal{M}$.

Définition 34 (formule prénexe, matrice) *Une formule* **FO** *ou* **MSO** *est dite sous forme prénexe si elle est de la forme*

$$Q_1 X_1 \ldots Q_m X_m Q'_1 x_1 \ldots Q'_n x_n \psi$$

où les Q_i, Q'_j sont des quantificateurs et ψ est une formule sans quantificateur qu'on appelle la matrice de la formule.

Remarque 3 *Toute formule* **FO** *(resp.* **MSO***) peut se mettre sous une forme prénexe* **FO** *(resp.* **MSO***) équivalente.*

Définition 35 (formule existentielle, Σ_1) *On appelle formule existentielle une formule* **FO** *équivalente à une formule* **FO** *qui est sous forme prénexe avec seulement des quantificateurs $\exists$. On note Σ_1 la classe des formules existentielles.*

1.4 Le modèle de calcul

Le modèle de calcul RAM que nous utiliserons et qui est décrit ci-dessous a été essentiellement décrit et étudié par Grandjean et ses collaborateurs dans une série d'articles [Gra96, GS02, GO04].

1.4.1 La machine RAM et ses entrées/sorties

Définition 36 *Une machine RAM est composée de*

- *une suite infinie de registres d'entrée $I(0), I(1), \ldots$, une suite infinie de registres de travail $R(0), R(1), \ldots$ et deux registres spéciaux de travail A et B. Tous ces registres contiennent des entiers qui sont initialisés à 0 ;*
- *une suite finie d'instructions étiquetées et qui constitue le programme de la RAM.*

Les instructions autorisées sont les suivantes :

- *$A \leftarrow I(A)$: affecte au registre A le contenu du registre $I(i)$ où i est le contenu du registre A ;*
- *$A \leftarrow R(A)$: affecte au registre A le contenu du registre $R(i)$ où i est le contenu du registre A ;*
- *$B \leftarrow A$: affecte au registre B le contenu du registre A ;*
- *$R(A) \leftarrow B$: affecte au registre $R(i)$ le contenu du registre B où i est le contenu du registre A ;*
- *$A \leftarrow A + B$: affecte au registre A la somme des contenus des registres A et B ;*
- *$A \leftarrow A - B$: affecte au registre A la différence des contenus des registres A et B ;*
- *goto(i) if $A > 0$: saute à l'instruction i si le contenu du registre A est strictement supérieur à 0 sinon passe à l'instruction suivante ;*
- *write(A) écrit le contenu du registre A en sortie ;*
- *stop arrête la machine.*

L'ensemble des *entrées/sorties* d'une RAM est l'ensemble des mots binaires non vides de $\{0,1\}^+$. Une telle entrée $w \in \{0,1\}^+$, encore appelée *donnée*, est initialement présente dans les registres initiaux de la RAM sous une forme standard appelée la *présentation logarithmique* de w et que nous décrivons ci-dessous.

Définition 37 (présentation logarithmique) *On appelle présentation (ou représentation) logarithmique d'un mot binaire $w \in \{0,1\}^+$ de longueur $n \geq 1$ la liste de mots binaires et d'entiers $(m, w_1, w_2, \ldots, w_m, l, r)$ définie de la manière suivante :*

- $l = l(n) = \lceil \frac{1}{2} log_2(n+1) \rceil$;
- $m = \lceil \frac{n}{l} \rceil$;
- $w = w_1 w_2 \dots w_m$ *(concaténation)* ;
- *pour tout* $i < m$, *longueur*$(w_i) = l$;
- $r = n - (m-1)l$ *et longueur*$(w_m) = r$ *(r comme "reste").*

L'entier m est appelé la *taille de la représentation*. Le choix de la fonction $l(n)$ pourrait être autre. Par exemple, $l(n) = \lceil \frac{3}{4} \log_2(n+1) \rceil$ ou $l(n) = \lfloor \frac{1}{3} \log_2(n) + 1 \rfloor$.

Points essentiels à respecter par $l(n)$:

- $\forall n \geq 1, l(n) \geq 1$;
- $l(n)$ est "facile à calculer" à partir de n ;
- $l(n) = \Theta(log(n))$;
- $2^{l(n)} = O(m)$ où $m = \lceil \frac{n}{l(n)} \rceil$.

Convention : on assimile chaque mot binaire w_i à l'entier qu'il représente en binaire.

Conséquences de tous ces points :

- $m = \Theta(\frac{n}{log(n)})$;
- $w_i = O(m)$ pour tout $i \in [m]$ (car $w_i < 2^{l(n)} = O(m)$) ;
- $0 < r \leq l(n) = O(m)$ (car $l(n) = \Theta(log(n)) = O(\frac{n}{log(n)}) = O(m)$). Autrement dit, tous les éléments (entiers) d'une présentation logarithmique d'un mot w de longueur $n \geq 1$ $(m, w_1, \dots, w_m, l, r)$ sont $O(m)$ avec la taille $m = \Theta(\frac{n}{log(n)})$.

L'entier m sera appelé la *taille logarithmique* (ou simplement la *taille*) de l'entrée w.

Remarques :

- A une présentation logarithmique correspond un seul mot binaire w.
- Le passage d'un mot à sa présentation logarithmique et le passage réciproque sont faciles à calculer.
- Les paramètres m, l et r sont nécessaires à mémoriser à cause du fait que deux mots binaires dont l'un est obtenu à partir de l'autre en ajoutant des zéros à gauche représentent le même entier ; pour éviter cette ambiguïté, la longueur (l ou r) du mot doit être précisée.

On dit qu'une machine RAM a pour *entrée* le mot binaire non vide $w \in \{0,1\}^+$ si ses $m+3$ premiers registres d'entrée $I(0), \dots, I(m+2)$ contiennent les entiers $m, w_1, \dots, w_m, l, r$ de la présentation logarithmique de w (m dans $I(0)$, w_i dans $I(i)$, $i \in [m]$, enfin l et r dans $I(m+1)$ et $I(m+2)$).

Point essentiel 1 : La complexité (en temps, espace, etc.) de la RAM sera toujours fonction de la taille logarithmique de l'entrée.

Point essentiel 2 : Tous les contenus des registres d'entrée sont $O(m)$ où m est la taille logarithmique du mot d'entrée w qui, à 3 près, est le nombre de registres occupés par l'entrée.

La sortie d'une machine RAM est le mot binaire $w' = w'_1 w'_2 \dots w'_p$ (concaténation) où w'_i est la représentation binaire du i-ème entier écrit en sortie par la machine RAM, $i \in [p]$.

1.4.2 Complexité d'une machine RAM : capacité, espace et temps

Nous décrivons ici ce que nous entendons par complexité en espace et en temps d'une machine RAM. On mesurera la complexité avec le *coût uniforme*. De façon générale, il est utile de parler de *capacité* et de *base* d'une RAM.

Définition 38 (capacité et base d'une RAM) *On dit qu'une RAM est de capacité (ou borne : "bound") $B(m)$ si, pour toute entrée de taille m, les entiers contenus dans ses registres de travail sont inférieurs à $B(m)$.*

On appelle base d'une RAM M sur une entrée I de taille m l'entier $B = B(m)$: $B-1$ est le plus grand entier que peut contenir un registre de travail de M.

Remarque 4 *Il est évident que si une RAM est de capacité $B(m)$, alors elle est aussi de capacité $B'(m)$, pour toute fonction $B'(m) \geq B(m)$.*

Dans son calcul sur une entrée I, une RAM de capacité $B(m)$ peut évidemment manipuler des entiers supérieurs à la base $B = B(|I|)$. Typiquement, un entier a inférieur à B^k est donné par sa représentation $(a_0, a_1, \dots, a_{k-1})$ en base B, avec $a = a_0 + a_1 B + \dots + a_{k-1} B^{k-1}$ et $a_i \in [0, B-1]$, donc sur k registres de travail.

Nos classes de complexité en espace et en temps sont définies sur des RAM de capacité bornée : voir nos définition ci-dessous.

Notation 2 (taille d'une donnée) *La taille d'une donnée I est notée $|I|$*[1].

Points essentiels : La taille $|I|$ d'une donnée I est (à 3 près dans le codage ci-dessus) le nombre de registres occupés par la présentation logarithmique de I.

Dans les cas courants (I est par exemple un graphe, un hypergraphe, une formule logique, etc), l'entier $|I|$ est, à un facteur multiplicatif près, la taille "intuitive" de la donnée :

- par exemple, pour une formule propositionnelle, c'est le nombre d'occurrences de variables dans la formule, ou encore le nombre d'occurrences de variables, de connecteurs et de parenthèses ;
- pour un graphe $G = (V, E)$, c'est $|G| = \mathrm{card}(V) + 2 \cdot \mathrm{card}(E)$;
- pour un hypergraphe $H = (V, E)$, c'est $H = \mathrm{card}(V) + \sum_{e \in E} \mathrm{card}(e)$.

[1] Il est important de remarquer que notre notation diffère de celle qui est adoptée par Flum, Frick et Grohe dans leurs divers articles ou livres. Ces auteurs désignent par $||I||$ la taille d'une donnée I et réservent la notation $|E|$ pour désigner la cardinalité d'un ensemble E

Vérifions le par exemple pour un graphe $G = (V, E)$, codé de façon normalisé avec $V = [v]$ sous forme de la liste de ses v sommets suivie de la liste de ses e arêtes (couples de sommets). En conséquence, la "taille intuitive" de G est intuitive-size$(G) = v + 2e$. La longueur n du mot binaire qui représente G est donc

$$n = \text{binary-length}(G) = \Theta((v+e)\log v) = \Theta((v+e)\log(v+e))$$

Finalement, on en déduit que la taille (logarithmique) m de ce mot binaire est

$$m = \Theta(\frac{n}{\log n}) = \Theta(v+e) = \Theta(\text{intuitive-size}(G))$$

comme attendu.

Robustesse de la taille : Comme les notions de complexité en temps et en espace, la notion de taille d'une donnée est définie de façon robuste, à un facteur multiplicatif près. Ainsi, pour un graphe $G = (V, E)$, on peut aussi écrire $|G| = \text{card}(V) + 2 \cdot \text{card}(E)$, ou encore, de façon plus robuste (c'est-à-dire indépendante du codage), $|G| = \Theta(\text{card}(V) + \text{card}(E))$.

Remarque 5 (taille et cardinalité) *Pour un ensemble E, il faut distinguer sa taille $|E|$ et sa cardinalité card(E), excepté dans le cas, assez fréquent, où les éléments de E sont de taille constante. Ainsi, pour un hypergraphe $H = (V, E)$, on pourra écrire $|H| = |V| + |E| = |V| + \sum_{e \in E} |e|$. Pour un ensemble d'hyperarêtes, on a en général card$(E) \neq |E|$. A l'inverse, pour un graphe $G = (V, E)$, on peut écrire $|E| =$ card(E).*

Définition 39 (complexité en temps, complexité en espace) *Une RAM est de complexité en espace $O(S(m))$ si, sur toute entrée de taille m, elle n'utilise que des registres $R(i)$ d'adresses $i = O(S(m))$ et de capacité $B(m) = O(m + S(m))$.*

Une RAM est de complexité en temps $O(T(m))$ si, sur toute entrée de taille m, elle exécute $O(T(m))$ instructions et est de complexité en espace $O(T(m))$.

Exemple 3 *Une RAM est de complexité en temps constant, si, sur toute entrée de taille m, elle exécute $O(1)$ instructions et n'utilise que $O(1)$ registres de travail de capacité $O(m)$.*

Une RAM est de complexité en temps $O(m^2)$ et en espace $O((\log m)^3)$ si, sur toute entrée de taille m, elle exécute $O(m^2)$ instructions et n'utilise que $O((\log m)^3)$ registres de travail de capacité $O(m)$.

Il est utile de préciser la correspondance entre la complexité ainsi définie (avec le coût uniforme) et la complexité pour le coût logarithmique (coût en nombre de bits). En particulier, une RAM de complexité en *temps linéaire*, donc $O(m)$, est, pour le coût logarithmique, de complexité en temps $O(m \log m) = O(n)$ où n est la longueur (en nombre de bits) de l'entrée. Elle utilise

$O(m)$ registres de capacité $O(m)$, donc chacun de nombre de bits $O(\log(m))$, et, par conséquent, une mémoire totale $O(m\log(m)) = O(n)$ en nombre de bits.

Le tableau suivant donne, pour quelques fonctions usuelles de complexité en temps, leur conversion en termes de coût logarithmique, donc en fonction du nombre de bits n de l'entrée. Afin de rendre ces complexités plus tangibles, nous donnons, pour chacune, un ou des exemples de problèmes qui ont cette complexité.

Coût uniforme	Coût logarithmique	Exemple de problème
$O(m)$	$O(n)$	tri, normalisation [a], parcours de graphe.
$O(m(\log m)^d)$	$O(n(\log n)^d)$	minimisation d'un automate déterministe ($d = 1$)
$O(m^d)$	$O((\frac{n}{\log n})^d \log n)$	produit de matrices
$O(c^m)$	$O(c'^{\frac{n}{\log n}})$	problèmes dits "exponentiels"
$O(1)$	$O(\log n)$	somme de deux éléments d'un tableau
$O(\log m)$	$O((\log n)^2)$	recherche dichotomique dans un tableau trié

[a] Grandjean [Gra96] a établi que, dans le modèle RAM qu'on a présenté ci-dessus, le tri d'une liste (formée d'un nombre quelconque) de mots binaires, chacun de longueur quelconque, se fait en temps linéaire $O(m)$ par une variante de l'algorithme "radix-sort" (présenté par exemple dans [AHU82]) ; de ce fait, la *normalisation* d'une structure quelconque se fait aussi en temps linéaire $O(m)$. La normalisation d'une structure $\mathcal{M}$ de domaine D consiste à lui associer une structure isomorphe $\mathcal{M}_{normal}$ de domaine $D_{normal} = [d]$ où $d = \text{card}(D)$.

On remarquera que la notion de temps constant $O(1)$ n'a de sens que lorsqu'on considère le coût uniforme ; dans ce cas, on a envie de dire que la somme de deux éléments d'un tableau se fait en temps constant. De même, on dit que la recherche dichotomique dans un tableau trié de taille m se fait en temps logarithmique $O(\log m)$. Dans les deux cas, la notion naturelle et intuitive est le coût uniforme.

1.5 Complexité classique et complexité paramétrée

1.5.1 Problèmes et classes de complexité classiques

On reformule librement les notions classiques.

Définition 40 (problème, problème de décision) *Un problème est une fonction $f : \{0,1\}^+ \to \{0,1\}^+$. Si on a $f(I) = J$, on dira que J est le résultat ou sortie du problème f sur l'entrée (ou l'instance ou la donnée) I.*

Un problème de décision est une fonction $f : \{0,1\}^+ \to \{0,1\}$, souvent confondue avec l'ensemble de ses instances positives $\{I \in \{0,1\}^+ : f(I) = 1\}$.

Dans cette thèse, on s'intéressera surtout aux classes de complexité en temps.

Définition 41 (complexité en temps) *On dira qu'un problème f appartient à la classe $Time(T)$, si f est calculable par un algorithme (c'est-à-dire une machine RAM) de complexité en temps $O(T(n))$, c'est-à-dire qui calcule la fonction $I \mapsto f(I)$ en temps $O(T(|I|))$.*

Remarque 6 *Ici, n désigne la taille de l'entrée I. Dorénavant, on parlera seulement de la taille d'une donnée et plus du tout de sa longueur en nombre de bits.*

Définition 42 (P, DLIN**)** *Un problème de décision A appartient à la classe* **P** *(resp.* DLIN*) s'il existe un algorithme qui décide A en temps inférieur à $p(|I|)$ (resp. $O(|I|)$, pour toute entrée I et un polynôme fixé p.*

1.5.2 Problèmes et classes de complexité paramétrés

Nous présentons ici les définitions données par Flum et Grohe [FG06], à peine adaptées.

Définition 43 (problème de décision paramétré) *Un problème de décision paramétré est un couple $A = (Q, \kappa)$ où Q est une partie de $\{0,1\}^+$, c'est-à-dire un problème de décision, et κ est une fonction de $\{0,1\}^+$ dans $\mathbb{N}$ qui est calculable en temps linéaire.*

Définition 44 (FPT, FPLIN) *Un problème de décision paramétré $A = (Q, \kappa)$ est dit* **FPT** *(ou Fixed Parameter Tractable) (resp.* **FPLIN** *: Fixed Parameter Linear) s'il existe un algorithme qui, pour toute instance $I \in \{0,1\}^+$, décide si $I \in Q$ en temps $f(\kappa(I)) \cdot p(|I|)$ (resp. $f(\kappa(I)) \cdot |I|$), pour une fonction calculable f et un polynôme p.*

Définition 45 (FPT-réduction) *Soient $A = (Q, \kappa)$ et $B = (Q', \kappa')$ deux problèmes de décision paramétrés. Une* **FPT***-réduction de A à B est une fonction $r : \{0,1\}^+ \to \{0,1\}^+$ telle que :*

1. *Pour toute instance $I \in \{0,1\}^+$, on a $I \in Q$ si et seulement si $r(I) \in Q'$;*
2. *r est calculable par un algorithme* **FPT***, c'est-à-dire calculable en temps $f(\kappa(I)) \cdot p(|I|)$, pour une fonction calculable f et un polynôme p ;*
3. *Il existe une fonction calculable $g : \mathbb{N} \to \mathbb{N}$ telle que $\kappa'(r(I)) \leq g(\kappa(I))$, pour tout $I \in \{0,1\}^+$.*

Nous utiliserons aussi les classes usuelles de complexité paramétrée : chacune de ces classes peut être définie comme la clôture par **FPT**-réduction du problème de model-checking paramétré suivant pour une certaine logique $\mathcal{L}$.

p-MC($\mathcal{L}$)	
Entrée:	Une structure $\mathcal{M}$ et un énoncé $\varphi \in \mathcal{L}$ de même signature que $\mathcal{M}$
Paramètre:	la taille $\lvert\varphi\rvert$
Sortie:	A-t-on $\mathcal{M} \models \varphi$?

Définition 46 *On appelle classe $W[1]$ la classe des problèmes qui admettent une* **FPT***-réduction au problème p-$MC(\Sigma_1)$ ou, de façon équivalente, au problème de l'existence d'une clique de taille k dans un graphe G (avec k comme paramètre).*

On appelle classe $AW[*]$ la classe des problèmes qui admettent une **FPT***-réduction au problème p-MC(**FO**).*

1.6 Les problèmes dynamiques et leur complexité

1.6.1 Problèmes dynamiques et schémas dynamiques

Nous introduisons dans cette section les *problèmes dynamiques*. Soient deux fonctions $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N}^2 \to \mathbb{N}$. Un problème dynamique possède deux entrées : une (grande) *entrée statique* et une *entrée dynamique* qui est "petite" (dans nos applications, nous la supposerons de taille constante). Un algorithme pour un problème dynamique (qu'on appellera un *schéma dynamique*) opère en deux temps. D'abord, il fait un précalcul en n'ayant connaissance que de l'entrée statique. On lui donne ensuite l'entrée dynamique et il opère à partir de cette nouvelle entrée et du résultat du précalcul précédent : c'est la *phase dynamique*. Un schéma dynamique est intéressant (efficace) dans la mesure où le temps de la phase dynamique dépend le moins possible ou pas du tout de la taille de l'entrée statique et est "petit" (souvent constant) par rapport au temps du précalcul.

Nous donnons maintenant des définitions plus formelles.

Définition 47 *Un problème dynamique est une fonction $A : \mathcal{I} \times \mathcal{D} \to \mathcal{O}$ pour laquelle*

- *$\mathcal{I}$ est un ensemble de données appelées entrées statiques ;*
- *$\mathcal{D}$ est un ensemble de données appelées entrées dynamiques ;*
- *$\mathcal{O}$ est un ensemble de données appelées sorties ou solutions.*

Définition 48 *Un schéma dynamique pour un problème dynamique $A : \mathcal{I} \times \mathcal{D} \to \mathcal{O}$, est un couple d'algorithmes $(ext, \mathcal{E})$ tel que*

- *ext est un algorithme qui calcule une fonction $ext : \mathcal{I} \to \mathcal{I}'$ (où l'ensemble $\mathcal{I}'$ est appelé espace des entrées étendues de A), et qui, à toute entrée $I \in \mathcal{I}$, associe l'entrée étendue, notée $I' = ext(I)$ et qui vérifie $|I'| \geq |I|$;*
- *$\mathcal{E}$ est un algorithme qui prend en entrée l'entrée étendue $I' = ext(I)$ et une entrée dynamique $x \in \mathcal{D}$ et renvoie la solution $A(I, x) = \mathcal{E}(ext(I), x)$.*

Les algorithmes ext et $\mathcal{E}$ sont appelés respectivement phase statique *(ou encore prétraitement ou précalcul) et* phase dynamique *(ou calcul) du schéma dynamique.*

Remarque 7 *Par souci de simplicité, on utilise ici la même notation pour désigner l'algorithme, ext ou $\mathcal{E}$, et la fonction que celui-ci calcule.*

Nous définissons maintenant les classes de complexité en temps pour les problèmes dynamiques.

Définition 49 *Un problème dynamique A appartient à la classe $dyn(f, g)$ s'il existe un schéma dynamique $(ext, \mathcal{E})$ pour A dont la phase statique ext est exécutée en temps $O(f(|I|))$ et la phase dynamique $\mathcal{E}$ l'est en temps $O(g(|I|, |A(I, x)|))$.*
On note Constant-Time$_{lin}$ *la classe $dyn(n, 1)$, c'est-à-dire la classe des problèmes dynamiques calculables en temps constant après un précalcul linéaire.*

Remarque 8 *Dans la définition précédente, les arguments de la fonction g qui mesure le temps de la phase dynamique sont la taille de l'entrée statique et celle de la sortie. En toute rigueur, il faudrait ajouter un troisième argument : la taille de l'entrée dynamique. Par souci de simplicité, nous l'avons omise ici, car, partout où nous utilisons des problèmes (et schémas) dynamiques dans le manuscrit, chaque entrée dynamique est de taille constante, et, pour cette raison, sa taille n'intervient pas dans la complexité.*

1.6.2 Algorithmes dynamiques pour les opérations division et modulo

Le modèle RAM considéré permet d'implémenter en temps constant, non seulement l'addition et la soustraction de deux entiers (utilisant chacun un nombre fixé de registres), mais aussi le produit de deux entiers (utilisant encore un nombre fixé de registres), ceci, grâce à un prétraitement qui calcule certaines tables en temps linéaire (voir par exemple l'article de Grandjean et Schwentick [GS02]). La division entière, a div d (quotient), et l'opération modulo, a mod d (reste de la division de a par d), sont également des opérations de base et il serait intéressant de pouvoir les implémenter aussi en temps constant. Malheureusement, cela ne semble pas possible dans le cas général. Toutefois, nous allons maintenant montrer que c'est possible dans le cas particulier où le diviseur est fixé, c'est-à-dire connu à l'issue d'un précalcul.

Nous formalisons cette question sous la forme des deux problèmes dynamiques suivants où k est un entier fixé, $k \geq 1$.

Div(k)	
Entrée statique:	une donnée I, un entier d, $1 \leq d < \|I\|^k$
Entrée dynamique:	un entier $a < \|I\|^k$
Sortie:	a div d

Mod(k)	
Entrée statique:	une donnée I, un entier d, $1 \leq d < \|I\|^k$
Entrée dynamique:	un entier $a < \|I\|^k$
Sortie:	a mod d

Nous allons montrer que chacun de ces deux problèmes se calcule en temps constant après un précalcul linéaire. Cela s'obtient assez facilement pour le problème Mod(k).

Lemme 4 *Soit un entier $k \geq 1$ fixé. On a Mod(k) $\in$ CONSTANT-DELAY$_{lin}$, autrement dit le problème Mod(k) est calculable en temps constant après un précalcul en temps $O(|I|)$.*

Preuve. La RAM qui réalise le précalcul est de complexité en temps $O(|I|)$. Par définition, la capacité de la RAM est $B(|I|) = \Theta(|I|)$. Sans perte de généralité, on peut donc supposer que la base de la RAM est $B = B(|I|) = |I|$. En conséquence, tout entier $a < |I|^k$, donc inférieur à B^k, s'écrit en base B sur k registres sous la forme $(a_0, a_1, \ldots, a_{k-1})$ avec $a_i < B$ pour tout $i < k$.

Considérons une donnée I, deux entiers $a < |I|^k, b < |I|^k$ et $B = |I|$ la base de la RAM. L'entier a s'écrit donc sur k registres $(a_0, \ldots, a_{k-1})$, $a_i < B$, et est de la forme $a_0 + a_1 B + \ldots + a_{k-1}B^{k-1}$.

On les égalités suivantes.

$$\begin{aligned} a \bmod d &= (a_0 + a_1 B + \ldots + a_{k-1}B^{k-1}) \bmod d \\ a \bmod d &= (a_0 \bmod d + (a_1 B) \bmod d + \ldots + (a_{k-1}B^{k-1}) \bmod d) \bmod d \end{aligned}$$

Le problème se réduit donc à

1. savoir calculer $B[i] = B^i \bmod d$, pour $0 \leq i < k$;
2. savoir calculer $C[i, j] = jB^i \bmod d$, pour $j < B$ et $0 \leq i < k$;
3. savoir calculer $j \bmod d$, pour $j < kd$.

Examinons maintenant chacune de ces trois tâches.

(1) : la table des $B[i]$ peut facilement être calculée en temps $O(|I|)$.

(2) : la table des $C[i, j]$ peut être précalculée en temps $O(|I|)$ par l'algorithme suivant :

Algorithm 1 calcul des $C[i, j] = jB^i \bmod d$

```
1: pour i = 0 à k − 1 faire
2:     C[i, 0] = 0
3:     pour j = 1 à B − 1 faire
4:         C[i, j] ← C[i, j − 1] + B[i]
5:         si C[i, j] ≥ d alors
6:             C[i, j] ← C[i, j] − d
7:         fin si
8:     fin pour
9: fin pour
```

(3) : Tout entier $j \bmod d$, avec $j < kd$, peut être aisément calculé en temps constant en soustrayant au plus k fois l'entier d à l'entier j.

□

Avant de donner un schéma dynamique pour le problème Div(k), nous allons considérer un problème dynamique intermédiaire, toujours pour un entier fixé $k \geq 1$.

Div-aux(k)	
Entrée statique:	une donnée I, et deux entiers d, $1 \leq d < \|I\|^k$ et $\beta \in [2, \|I\| - 1]$ qui sont premiers entre eux
Entrée dynamique:	un entier $a < \beta^k$ représenté dans la base β
Sortie:	a div d

Lemme 5 *Le problème Div-aux(k) est calculable par un schéma dynamique de phase statique en temps $O(|I|)$ et de phase dynamique en temps constant.*

Preuve. Sans perte de généralité, on peut supposer que l'entier a est divisible par d. En effet, l'entier $a' = a - (a \bmod d)$ est facilement calculable, par le lemme précédent, et est divisible par d et on a bien sûr a' div $d = a$ div d. D'autre part, on peut supposer $a < \beta d$. En effet, l'algorithme usuel de division d'un entier a écrit en base β sous la forme $(a_{k-1}, a_{k-2}, \ldots, a_0)$ avec $a_i < \beta$, par un entier d, se ramène à k itérations d'une division par d d'un entier de la forme $r_i\beta + a_{i-1}$ (avec $r_i \in [0, d-1]$), donc inférieur à βd. On peut donc supposer que a appartient à l'ensemble $\mathcal{A} = \{id : i \in [0, \beta - 1]\}$.

Considérons la fonction $T : [0, \beta - 1] \to [0, \beta - 1]$ telle que $T(id \bmod \beta) = i$, pour tout $i \in [0, \beta - 1]$. Cette fonction est bien définie. En effet, considérons deux entiers $x = id \bmod \beta$ et $x' = i'd \bmod \beta$ avec $i, i' \in [0, \beta - 1], i \neq i'$ et $x \leq x'$. Alors $x' - x = (i' - i)d \bmod \beta$. Puisque l'entier β ne divise pas $i' - i$ et est premier avec d, il s'en suit que β ne divise pas le produit $(i' - i)d$. On en déduit $x' - x = (i' - i)d \bmod \beta \neq 0$ et donc $x \neq x'$. D'autre part, la fonction T peut facilement être précalculée sous forme d'un tableau en temps $O(|I|)$.

On a donc, pour tout entier $a \in \mathcal{A}$, l'égalité a div $d = T(a \bmod \beta)$. En conséquence, le quotient a div d, toujours en utilisant le lemme précédent, est calculé en temps constant par accès direct au tableau T.

□

Lemme 6 *Soit un entier fixé $k \geq 1$. Soient I une entrée de taille suffisamment grande et deux entiers β et d tels que $2 \leq \beta \leq |I|$ et $1 \leq d < |I|^k$. Alors on peut calculer en temps $O(|I|)$ un entier β' premier avec β et d et vérifiant $\sqrt{|I|} \leq \beta' \leq |I|$.*

Preuve. Considérons la liste $p_1, \ldots p_l$ des l premiers nombres premiers pour $l = (k+1)\lceil \log |I| \rceil + 1$. Comme l'entier $\beta \leq |I|$ (resp. $d < |I|^k$) admet au plus $\log |I|$ (resp. $k \log |I|$) facteurs premiers distincts et, puisque $l > (k+1) \log |I|$, il existe, pours des raisons de cardinalité, un entier premier $q \in \{p_1, \ldots p_l\}$ qui ne divise ni β ni d. Considérons l'entier j tel que $\frac{|I|}{q} < q^j \leq |I|$ et

prenons $\beta' = q^j$. Alors, comme q, l'entier β' est premier avec β et d. D'autre part, on obtient $q \leq p_l \leq C \cdot l \cdot \log l$, pour une certaine constante universelle C, d'après le théorème (classique) des nombres premiers. On en déduit $q \leq C' \log |I| \log\log |I| \leq \sqrt{|I|}$, pour une taille de l'entrée I suffisamment grande et une constante C' qui ne dépend que de k. On a donc

$$\sqrt{|I|} \leq \frac{|I|}{C' \log |I| \log\log |I|} \leq \frac{|I|}{q} \leq \beta' \leq |I|$$

On se convaincra facilement que chaque étape du calcul de β', le calcul des l premiers nombres premiers, pour $l = O_k(\log |I|)$, la recherche du nombre premier q et du résultat $\beta' = q^j$, est réalisée en temps polynomial en $\log |I|$. □

Lemme 7 *Soit un entier fixé $k \geq 1$. On a Div$(k) \in$ CONSTANT-TIME$_{lin}$, autrement dit, le problème Div(k) est calculable en temps constant après un prétraitement en temps $O(|I|)$.*

Preuve. Soit I, d, a une donnée du problème Div(k) (entrée statique et entrée dynamique). Considérons l'entier β' calculé par l'algorithme du Lemme 6. A cause de l'inégalité $\sqrt{|I|} \leq \beta'$, on obtient $a < |I|^k \leq \beta'^{2k}$.

Les étapes de la phase dynamique du schéma pour le problème Div(k) sont les suivantes :

- écrire a dans la base β' sur $2k$ registres ;
- diviser a (écrit en base β') par d.

Remarquons que la conversion de l'entier a (écrit en base $B = |I|$) dans la base β' utilise seulement un nombre fixé de divisions et de modulos par l'entier β' et donc se ramène aux problèmes Mod(k) (sur l'entrée statique I, β') et Div-aux(k) (sur l'entrée statique I, β', B). La division de a (écrit dans la base β') par d se ramène au problème Div-aux$(2k)$ (car $a < \beta'^{2k}$) sur l'entrée statique (I, d, β'). Le lemme est donc prouvé. □

1.7 Problèmes d'énumération et autres problèmes : comptage et j-ème solution

Pour un problème quelconque (qu'il soit combinatoire, logique ou autre), qui, à une instance (entrée du problème), associe un ensemble de solutions, on peut se poser une variété de questions : générer toutes les solutions du problème, compter son nombre de solutions, calculer une solution (n'importe laquelle), tester l'existence d'une solution, générer la j-ème solution du problème (pour un certain ordre des solutions), etc. Dans cette thèse, nous étudions quatre de ces questions. Dans un souci de précision et d'uniformité, nous définissons d'abord ici des problèmes types.

1.7.1 Problèmes généraux

Définition 50 (problème général) *Un problème général est une relation $A \subseteq \mathcal{I} \times \mathcal{O}$ telle que, pour tout $I \in \mathcal{I}$, on a $card(\{y \in \mathcal{O} : (I, y) \in A\}) < \infty$. Les ensembles $\mathcal{I}$ et $\mathcal{O}$ sont des ensembles de données :*

- *$\mathcal{I}$ est appelé l'espace des entrées de A ;*
- *$\mathcal{O}$ est appelé l'espace des solutions ou l'espace des sorties de A.*

Pour toute instance $I \in \mathcal{I}$, on définit l'ensemble fini $A(I) = \{y \in \mathcal{O} : (I, y) \in A\}$ qu'on appelle l'ensemble des solutions du problème A pour l'entrée I.

Nous donnons maintenant la définition d'un problème général en présence d'un *ordre total* de l'espace des solutions.

Définition 51 (problème général ordonné) *Un problème général ordonné est un couple $(A, <)$ où A est un problème général, $A \subseteq \mathcal{I} \times \mathcal{O}$, et $<$ est une relation d'ordre total de l'espace $\mathcal{O}$ des solutions.*

1.7.2 Problèmes spécifiques

A partir d'un problème général (resp. général ordonné), on définit trois problèmes spécifiques.

Définition 52 (problème d'énumération) *Soit $A \subseteq \mathcal{I} \times \mathcal{O}$ un problème général. Le problème d'énumération associé à A, noté* Enum(A)*, est le problème qui prend en entrée une donnée $I \in \mathcal{I}$ et renvoie en sortie une séquence (sans répétition) $y_1, \ldots, y_k$ telle que $A(I) = \{y_1, \ldots, y_k\}$ (ensemble des solutions du problème A sur l'entrée I).*

Définition 53 (problème d'énumération ordonné) *Si $(A, <)$ est un problème général ordonné, $A \subseteq \mathcal{I} \times \mathcal{O}$, alors* Enum$(A, <)$ *est le problème qui prend en entrée une donnée $I \in \mathcal{I}$ et renvoie en sortie la séquence $y_1, \ldots, y_k$ telle que $A(I) = \{y_1, \ldots, y_k\}$ et $y_1 < y_2 < \ldots < y_k$.* Enum$(A, <)$ *est appelé le problème d'énumération ordonnée associé au problème $(A, <)$.*

Définition 54 (problème de comptage) *A tout problème général $A \subseteq \mathcal{I} \times \mathcal{O}$, on associe le problème de comptage, noté* Count(A)*, défini de la manière suivante :*

Count(A)	
Entrée:	*une instance $I \in \mathcal{I}$*
Sortie:	*la cardinalité $card(A(I))$*

Définition 55 (problème de la j-ème solution) *A tout problème général ordonné $(A, <)$, $A \subseteq \mathcal{I} \times \mathcal{O}$, on associe le problème dynamique, noté* Jth(A)*, défini de la manière suivante :*

JTH$(A, <)$	
Entrée statique:	une instance $I \in \mathcal{I}$ et l'entier card$(A(I))$
Entrée dynamique:	un entier $j <$ card$(A(I))$
Sortie:	la j-ème solution de $A(I)$ dans l'ordre $<$

Convention 8 *Par commodité, on convient que les rangs j des éléments de $A(I)$ pour l'ordre $<$ vont de 0 (pour le premier élément) à card$(A(I)) - 1$.*

1.8 Classes de complexité pour les problèmes d'énumération

1.8.1 Définitions

Définition 56 (algorithme d'énumération) *Etant donné un problème d'énumération* ENUM(A) *(resp. problème d'énumération ordonnée* ENUM$(A, <)$*), on dit qu'un algorithme $\mathcal{A}$ calcule le problème* ENUM(A) *(resp.* ENUM$(A, <)$*) si, pour toute entrée I du problème, l'algorithme $\mathcal{A}$ écrit séquentiellement en sortie le mot $\#y_1\#y_2\#\ldots\#y_n\#$ où $y_1, \ldots y_n$ est la sortie du problème* ENUM(A) *(resp.* ENUM$(A, <)$*), pour l'entrée I, et $\#$ est un caractère spécial. Par commodité, on supposera que le premier caractère $\#$ est écrit au temps 0 et que l'algorithme $\mathcal{A}$ s'arrête à l'instant précis où il a écrit le dernier $\#$.*

Remarque 9 *Par "séquentiellement", on entend que la machine RAM écrit de gauche à droite les solutions mises en sortie et ne peut pas modifier les mots y_i déjà écrits. D'autre part, la machine n'a pas le droit de lire les mots y_i mis en sortie. Ces conditions correspondent au fonctionnement des instrictions de sortie (écriture) du modèle RAM tel qu'il a été défini précédemment.*

Définition 57 (délai) *Soient un problème d'énumération* ENUM(A) *(resp. d'énumération ordonnée* ENUM$(A, <)$*) et un algorithme $\mathcal{A}$ qui calcule le problème* ENUM(A) *(resp.* ENUM$(A, <)$*). On désigne par $time_i(I)$ l'instant où l'algorithme écrit le i-ème caractère $\#$. On appelle délai le temps qui sépare l'écriture de deux solutions successives. Plus précisément, on note $delay_i(I)$ le temps que prend $\mathcal{A}$ pour mettre en sortie la i-ème solution pour l'entrée I, c'est-à-dire $delay_i(I) = time_{i+1}(I) - time_i(I)$.*

Johnson, Papadimitriou et Yannakakis [JPY88] ont introduit les trois classes de complexité suivantes pour les problèmes d'énumération.

Délai polynomial : DelayP

On dit qu'un algorithme $\mathcal{A}$ d'énumération pour un problème A est à *délai polynomial* si le délai pour mettre en sortie chaque solution est polynomial en la taille de l'entrée de $\mathcal{A}$: on écrira alors $A \in$ DelayP.

Délai polynomial incrémental : IncP

On dit qu'un algorithme $\mathcal{A}$ est à *délai polynomial incrémental* si le délai de $\mathcal{A}$ pour générer la i-ème solution d'un problème A est polynomial en la somme des tailles de l'entrée et des $i-1$ solutions déjà générées : on écrira alors $A \in$ IncP. Contrairement à la situation DelayP où la borne du délai reste uniforme entre deux solutions successives, le délai augmente ici au cours du temps, ce qui sous-entend que les solutions peuvent devenir de plus en plus difficiles à générer.

Temps polynomial global : TotalP

On ne s'intéresse plus ici au délai entre deux solutions successives mais seulement au temps total de génération de toutes les solutions. On appelle TotalP la classe des problèmes énumérables avec un temps total polynomial en la somme cumulée des tailles de l'entrée et des solutions. Il n'y a plus ici nécessairement de contrainte sur le délai. Contrairement à la classe IncP, la classe TotalP peut contenir des problèmes dont même les premières solutions sont "difficiles" à générer.

1.8.2 Classes d'énumération polynomiales : résultats connus

Clairement, on a la hiérarchie DelayP $\subseteq$ IncP $\subseteq$ TotalP.

Johnson, Papadimitriou et Yannakakis[JPY88] ont étudié le problème de l'énumération des stables maximaux (pour l'inclusion) d'un graphe G. Ils ont montré qu'énumérer les stables de G dans l'ordre lexicographique peut se faire à délai polynomial (et espace exponentiel) mais que les énumérer dans l'ordre inverse de l'ordre lexicographique est NP-difficile. Plus précisément, ils ont établi qu'étant donnés un graphe G et un stable S de G, décider si S est le dernier stable de G dans l'ordre lexicographique est un problème NP-complet. On en déduit immédiatement le théorème suivant :

Théorème 9 *[JPY88] Si $P \neq NP$ alors on a l'inclusion stricte DelayP $\subsetneq$ TotalP.*

La question de l'égalité (sous l'hypothèse $P \neq NP$) entre les classes DelayP et IncP ou entre les classes IncP et TotalP reste ouverte, à notre connaissance.

1.8.3 Enumération avec phase de précalcul

Comme nous l'avons fait pour les problèmes dynamiques, nous nous intéressons maintenant à des algorithmes d'énumération qui opèrent en deux phases : une phase de précalcul suivie d'une phase d'énumération. Nous donnons ici des définitions plus formelles.

Définition 58 (schéma d'énumération) *Soit $A \subseteq \mathcal{I} \times \mathcal{O}$ (resp. $(A, <)$, pour $A \subseteq \mathcal{I} \times \mathcal{O}$) un problème général (resp. général ordonné). Un schéma d'énumération de* Enum(A) *(resp.* Enum$(A, <)$*) est un couple $(ext, \mathcal{E})$ défini comme suit :*

- *ext est un algorithme (appelé phase de précalcul ou prétraitement) qui calcule une fonction, encore notée ext par abus de langage, de $\mathcal{I}$ dans un ensemble $\mathcal{I}'$; $\mathcal{I}'$ est appelé espace des entrées étendues ;*
- *$\mathcal{E}$ est un algorithme (appelé phase d'énumération) qui, prenant en entrée une entrée étendue ext(I), pour $I \in \mathcal{I}$, énumère sans répétition (resp. dans l'ordre $<$) les solutions de $A(I)$.*

La complexité d'un schéma d'énumération est mesurée par deux fonctions :

- l'une donne le temps du précalcul qui, pour une entrée I de A, calcule son "extension" ext(I) ;
- l'autre mesure le délai entre deux solutions consécutives produites par la phase d'énumération $\mathcal{E}$ et aussi l'espace utilisé par $\mathcal{E}$.

Définition 59 (enum(f, g)) *Soient deux fonctions $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N}^2 \to \mathbb{N}$ et soit $A \subseteq \mathcal{I} \times \mathcal{O}$ un problème général. On écrit* Enum$(A) \in$ *enum(f, g) si le problème* Enum(A) *possède un schéma d'énumération $(ext, \mathcal{E})$ qui vérifie les deux conditions suivantes :*

1. *Le précalcul ext utilise un temps $O(f(|I|))$, pour chaque entrée $I \in \mathcal{I}$, et on a toujours $|ext(I)| \geq |I|$;*
2. *Pour chaque entrée $I \in \mathcal{I}$, l'algorithme $\mathcal{E}$, appliqué à l'entrée étendue ext(I), génère successivement chaque solution $x \in A(I)$ avec un délai $O(g(|I|, |x|))$; pour cela, l'algorithme $\mathcal{E}$ utilise un espace $O(\max_{x \in A(I)} g(|I|, |x|))$.*

On dira alors que le problème Enum(A) *est énumérable à délai $O(g(n, m))$ avec précalcul en temps $O(f(n))$.*

On définit de façon similaire la complexité Enum$(A, <) \in$ *enum(f, g) pour un problème d'énumération ordonnée.*

Nous utiliserons souvent dans cette thèse la classe de complexité "minimale" Lin-Delay$_{lin}$ et son cas particulier Constant-Delay$_{lin}$.

Définition 60 *La classe d'énumération minimale enum(n, m) est notée* Lin-Delay$_{lin}$. *On s'intéresse aussi à la classe notée* Constant-Delay$_{lin}$ = *enum$(n, 1)$. Cette dernière classe n'a de sens que pour les problèmes dont chaque sortie est de taille constante : dans ce cas, affirmer que le problème est dans* Constant-Delay$_{lin}$ *équivaut à dire qu'il est dans* Lin-Delay$_{lin}$.

Pour les problèmes de décision, la classe DLIN (temps linéaire des machines RAM pour le coût uniforme) est la classe robuste minimale (en ce qui concerne le temps de calcul) comme l'ont montré par exemple Grandjean et Schwentick [GS02]. De même, la classe Constant-Delay$_{lin}$ est la classe robuste minimale (pour le temps de précalcul et le délai). Il y a deux raisons à cela.

1. *Temps d'entrée/sortie et minimalité :* Pour la quasi-totalité des problèmes d'énumération que l'on rencontre, il est nécessaire de parcourir au moins une fois la totalité de l'entrée

avant de pouvoir donner la première solution et, pour chaque solution, il faut au moins le temps (délai) pour l'écrire.

2. *Robustesse du modèle :* Il est nécessaire que le temps de précalcul soit au moins linéaire pour pouvoir assurer une forme de robustesse au niveau des opérations autorisées par la machine RAM (addition, multiplication, etc.), opérations que l'on veut réaliser chacune en temps constant.

De plus, comme on a montré [GS02] que la classe DLIN contient (et en fait s'identifie à) la large classe des problèmes décidables en temps linéaire, au sens intuitif, nous allons montrer que la classe d'énumération Lin-Delay$_{lin}$ (avec son cas particulier Constant-Delay$_{lin}$) est, "malgré" aussi son caractère minimal et sans doute encore à cause de sa robustesse, très expressive : en effet, dans nos trois chapitres (2, 3 et 4) qui étudient les problèmes d'énumération, nous prouverons que beaucoup des problèmes de requêtes qui nous intéressent appartiennent à cette classe.

La propriété suivante est immédiate et montre que le problème de l'énumération est "plus facile", d'une certaine manière, que celui de la recherche de la i-ème solution.

Lemme 10 *Soient deux fonctions $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N}^2 \to \mathbb{N}$ et soit $(A, <)$ un problème général ordonné tel que* $\textsc{Jth}(A, <) \in dyn(f, g)$ *et* $\textsc{Count}(A) \in Time(f)$. *Alors* $\textsc{Enum}(A, <) \in enum(f, g)$.

Preuve. Sur une instance I, le schéma d'énumération de $\textsc{Enum}(A, <)$ exécute le précalcul du schéma dynamique pour $\textsc{Jth}(A, <)$, puis itère la phase dynamique de ce schéma pour l'entrée dynamique i variant de 0 à $\text{card}(A(I)) - 1$. □

Remarque 10 *Comme nous le verrons dans le chapitre 2, la réciproque de ce lemme est fausse (sous une certaine hypothèse de complexité).*

Nous introduisons maintenant une notion de réduction fine entre les problèmes généraux. Le but de cette réduction est de montrer que, si un problème général A se réduit à un problème général B, alors le problème d'énumération associé à B est, dans un sens très précis, au moins aussi difficile que le problème d'énumération associé à A . Cette réduction doit vérifier deux conditions : d'une part, elle doit être un préordre et d'autre part, les classes de complexité pour l'énumération enum(f, g) doivent être closes par cette réduction.

Définition 61 (réduction parcimonieuse) *Soient $A \subseteq \mathcal{I} \times \mathcal{O}$ et $A' \subseteq \mathcal{I}' \times \mathcal{O}'$ (resp. $(A \subseteq \mathcal{I} \times \mathcal{O}, <)$ et $(A' \subseteq \mathcal{I}' \times \mathcal{O}', <)$) deux problèmes généraux (resp. généraux ordonnés). Une réduction parcimonieuse (resp. parcimonieuse croissante) du problème général A au problème général A' (resp. du problème général ordonné $(A, <)$ au problème général ordonné $(A', <)$) est un couple de fonctions (r, s) qui vérifie les trois conditions suivantes.*

1. *On a* $r : \mathcal{I} \to \mathcal{I}' \times \mathcal{J}$ *où* $\mathcal{J}$ *est un ensemble appelé espace des extensions ;*
2. *On a* $s : \mathcal{J} \times \mathcal{O}' \to \mathcal{O}$ *;*
3. *Pour toute instance* $I \in \mathcal{I}$ *et pour* $(I', J) = r(I)$*, la fonction* $x \mapsto s(J, x)$ *est une bijection (resp. bijection croissante) de* $A'(I')$ *(resp* $(A'(I'), <)$*), l'ensemble des solutions de* A'*, pour la donnée* I'*, dans* $A(I)$ *(resp.* $(A(I), <)$*), l'ensemble des solutions de* A *pour la donnée* I*.*

Définition 62 (réduction exacte) *Une réduction parcimonieuse (resp. parcimonieuse croissante)* (r, s) *du problème général* A *au problème général* A' *(resp. du problème général ordonné* $(A, <)$ *au problème général ordonné* $(A', <)$*) est dite* exacte *si elle vérifie les conditions de complexité linéaire suivantes :*

1. *La fonction* $I \mapsto r(I)$ *est calculable en temps* $O(|I|)$ *;*
2. *La fonction bijective (resp. bijective croissante)* $x \mapsto s(J, x)$ *est calculable en temps* $O(|x|)$ *et on a* $|s(J, x)| = \Theta(|x|)$*.*

On écrira $A \leq_{exact} A'$ *(resp.* $(A, <) \leq_{exact} (A', <)$*) s'il existe une réduction exacte de* A *à* A' *(resp. de* $(A, <)$ *à* $(A', <)$*).*

Remarque 11 *Pour beaucoup de réductions exactes* (r, s) *que nous verrons, nous aurons* $r(I) = (I', J)$ *avec* $I' = J$ *et, dans nombre de cas,* s *sera même la fonction "identité"* $s(J, x) = x$*.*

Définition 63 (composition de réductions parcimonieuses) *Soient trois problèmes généraux* $A \subseteq \mathcal{I}_A \times \mathcal{O}_A$*,* $B \subseteq \mathcal{I}_B \times \mathcal{O}_B$ *et* $C \subseteq \mathcal{I}_C \times \mathcal{O}_C$*. Soient deux réductions parcimonieuses* (r, s) *de* A *à* B *et* (r', s') *de* B *à* C*,* $r : \mathcal{I}_A \to \mathcal{I}_B \times \mathcal{J}_{AB}$*,* $s : \mathcal{J}_{AB} \times \mathcal{O}_B \to \mathcal{O}_A$*,* $r' : \mathcal{I}_B \to \mathcal{I}_C \times \mathcal{J}_{BC}$*,* $s' : \mathcal{J}_{BC} \times \mathcal{O}_C \to \mathcal{O}_B$*. On définit la réduction composée* $(r'', s'') = (r', s') \circ (r, s)$ *de* A *à* C*, de la façon suivante :*

- *Pour tout* $I_A \in \mathcal{I}_A$*, si* $r(I_A) = (I_B, J_{AB})$ *et* $r'(I_B) = (I_C, J_{BC})$ *alors* $r''(I_A) = (I_C, (J_{AB}, J_{BC}))$ *;*
- *Pour tout* $x_C \in C(I_C)$*, si* $s'(J_{BC}, x_C) = x_B$ *et* $s(J_{AB}, x_B) = x_A$ *alors* $s''((J_{AB}, J_{BC}), x_C) = x_A$*.*

La composée $(r'', s'') = (r', s') \circ (r, s)$ est une réduction parcimonieuse de A à C. En effet, pour tout $I_A \in \mathcal{I}_A$ et pour $(I_C, (J_{AB}, J_{BC})) = r''(I_A)$, la fonction $x_C \mapsto s(J_{AB}, s'(J_{BC}, x_C))$ est, par composition de bijections, une bijection de $C(I_C)$ dans $A(I_A)$. De même, la composition de deux réductions parcimonieuses croissantes est encore une réduction parcimonieuse croissante.

La réduction exacte a les propriétés que l'on attend comme l'exprime la proposition suivante :

Proposition 11 *La relation* $\leq_{exact}$ *est une relation de préordre pour les problèmes généraux (resp. problèmes généraux ordonnés).*

Preuve. La relation $\leq_{exact}$ est évidemment réflexive : considérer la réduction parcimonieuse triviale "identité". Pour démontrer que cette relation est transitive, il suffit de prouver que la composée de deux réductions exactes est encore une réduction exacte. Soient A, B, C trois problèmes généraux (ou généraux ordonnés) et soient (r, s) et (r', s') des réductions exactes de A à B et de B à C, respectivement. Il suffit de démontrer que la réduction composée $(r'', s'') = (r', s') \circ (r, s)$ vérifie les conditions (1) et (2) d'une réduction exacte.

1) Puisque, pour toute instance $I_A \in \mathcal{I}_A$, on a $r''(I_A) = (I_C, (J_{AB}, J_{BC}))$ avec $r(I_A) = (I_B, J_{AB})$ calculable en temps $O(|I_A|)$ et $r'(I_B) = (I_C, J_{BC})$ calculable en temps $O(|I_B|)$, on conclut que la fonction $I_A \mapsto r''(I_A)$ est calculable en temps $O(|I_A| + |I_B|) = O(|I_A|)$.

2) Puisque les bijections $x_C \mapsto s'(J_{BC}, x_C)$ et $x_B \mapsto s(J_{AB}, x_B)$ sont calculables en temps $O(|x_C|)$ et $O(|x_B|)$ respectivement, la bijection composée $x_C \mapsto s(J_{AB}, s'(J_{BC}, x_C))$ est calculable en temps $O(|x_C| + |s'(J_{BC}, x_C)|) = O(|x_C|)$. Par ailleurs, on a

$$|s(J_{AB}, s'(J_{BC}, x_C))| = \Theta(|s'(J_{BC}, x_C)|) = \Theta(|x_C|).$$

□

Nous allons maintenant démontrer que la complexité des problèmes d'énumération est préservée par réduction exacte. Pour cela, il faut imposer certaines conditions, toujours vérifiées dans la pratique, sur les fonctions f et g autorisées pour les classes enum(f, g). Nous introduisons pour cela la notion de *couple honnête de fonctions.*

Définition 64 (couple honnête de fonctions) *Un couple de fonctions (f, g) avec $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N}^2 \to \mathbb{N}$ est dit* ***honnête*** *si elles vérifient les quatre conditions suivantes :*

- $f(O(n)) = O(f(n))$;
- $g(O(n), O(m)) = O(g(n, m))$;
- $f(n) = \Omega(n)$;
- $g(n, m) = \Omega(m)$.

Remarque 12 *Les deux propriétés de f qui sont énoncées dans la définition précédente impliquent pour cette fonction l'encadrement suivant : $c.n \leq f(n) \leq p(n)$, pour tout entier n, pour une constante $c > 0$ et un polynôme fixé p.*

Convention 12 *Dorénavant, nous ne parlerons d'une classe enum(f, g) ou dyn(f, g) que sous l'hypothèse que le couple de fonctions (f, g) est honnête. Cette hypothèse sera implicite (sauf dans les deux lemmes suivants où nous la rappelons).*

A cause de cette convention et compte tenu de la remarque précédente et du fait qu'on a, par définition, pour tout $I \in \mathcal{I}$, l'inégalité $|\text{ext}(I)| \geq |I|$, pour tout schéma d'énumération (resp. schéma dynamique) $(\text{ext}, \mathcal{E})$ d'espace des entrées $\mathcal{I}$, on a l'implication suivante : si le schéma

$(\text{ext}, \mathcal{E})$ appartient à la classe $\text{enum}(f, g)$ (resp. $\text{dyn}(f, g)$), alors, pour toute entrée $I \in \mathcal{I}$, on a $|I| \leq |\text{ext}(I)| \leq p(|I|)$, pour un polynôme fixé p.

Lemme 13 *Soient A, B deux problèmes généraux (ou généraux ordonnés) avec $A \leq_{exact} B$. Si on a $\text{ENUM}(B) \in enum(f, g)$ pour un couple honnête de fonctions (f, g), alors on a aussi $\text{ENUM}(A) \in enum(f, g)$.*

Soit (r, s) une réduction exacte de A à B et soit $(\text{ext}, \mathcal{E})$ un schéma d'énumération pour B de complexité $\text{enum}(f, g)$. On doit construire pour A un schéma d'énumération $(\text{ext}', \mathcal{E}')$ de même complexité. On définit ext' comme la fonction qui, à tout $I_A \in \mathcal{I}_A$, associe le couple $(\text{ext}(I_B), J_{AB})$ pour $r(I_A) = (I_B, J_{AB})$. Enfin, on définit ci-dessous la phase d'énumération $\mathcal{E}'$ de A, agissant sur l'entrée étendue $\text{ext}'(I_A) = (\text{ext}(I_B), J_{AB})$.

Algorithm 2 Phase d'énumération $\mathcal{E}'(\text{ext}(I_B), J_{AB})$

1: faire fonctionner l'algorithme $\mathcal{E}$ sur l'entrée étendue $\text{ext}(I_B)$
2: **pour** tout x_B mis en sortie par $\mathcal{E}$ (c'est-à-dire pour tout $x_B \in B(I_B)$) **faire**
3: calculer $x_A = s(J_{AB}, x_B)$
4: mettre en sortie x_A
5: **fin pour**

Par construction, le schéma d'énumération $(\text{ext}', \mathcal{E}')$ pour A est correct : typiquement, l'algorithme $\mathcal{E}'$ énumère sans répétition (resp. dans l'ordre croissant) les solutions $x_A \in A(I_A)$ puisque l'algorithme $\mathcal{E}$, appliqué à la donnée $\text{ext}(I_B)$, énumère sans répétition (resp. dans l'ordre croissant) les solutions $x_B \in B(I_B)$ (où $r(I_A) = (I_B, J_{AB})$) et que la fonction $x_B \mapsto s(J_{AB}, x_B)$ est une bijection (resp. bijection croissante) de $B(I_B)$ dans $A(I_A)$.

Il reste à vérifier que les complexités du précalcul et du délai sont bien celles qu'on attend. Puisque les temps de calcul des fonctions $r : I_A \mapsto (I_B, J_{AB})$ et $\text{ext} : I_B \mapsto \text{ext}(I_B)$ sont $O(|I_A|)$ et $O(f(|I_B|))$, respectivement, le temps de calcul de la fonction composée $\text{ext}' : I_A \mapsto (\text{ext}(I_B), J_{AB})$ est $O(|I_A| + f(O(|I_A|))) = O(f(|I_A|))$ à cause des propriétés $f(O(n)) = O(f(n))$ et $f(n) = \Omega(n)$. Le délai de l'algorithme $\mathcal{E}'$ pour mettre en sortie une solution x_A, délai noté $\text{delay}(\mathcal{E}', x_A)$, est celui de $\mathcal{E}$ pour produire la solution x_B telle que $x_A = s(J_{AB}, x_B)$, délai qu'on note $\text{delay}(\mathcal{E}, x_B)$, auquel on ajoute le temps de calcul de la solution x_A à partir de (J_{AB}, x_B).

On a donc

$$\text{delay}(\mathcal{E}', x_A) = \text{delay}(\mathcal{E}, x_B) + O(|x_B|)$$

Par hypothèse, on a

$$\text{delay}(\mathcal{E}, x_B) = O(g(|I_B|, |x_B|)).$$

Or on a aussi $I_B = O(|I_A|)$, $|x_B| = \Theta(|x_A|)$ et $g(O(n), O(m)) = O(g(n, m))$. D'où on tire $\text{delay}(\mathcal{E}, x_B) = O(g(|I_A|, |x_A|))$ et $\text{delay}(\mathcal{E}', x_A) = O(g(|I_A|, |x_A|) + |x_A|)$. Comme on a aussi

$g(n,m) = \Omega(m)$, on en déduit $\text{delay}(\mathcal{E}', x_A) = O(g(|I_A|, |x_A|))$.

Pour les mêmes raisons et avec les mêmes calculs, on établit que l'algorithme $\mathcal{E}'$ utilise pour tout $I_A \in \mathcal{I}_A$ un espace $O(\max_{x \in A(I_A)} g(|I_A|, |x_A|))$. On a donc prouvé que, comme $\text{ENUM}(B)$, le problème $\text{ENUM}(A)$ est de complexité $\text{enum}(f,g)$. □

Notation 14 *Par abus de langage, on écrira souvent* $\text{ENUM}(A) \leq_{exact} \text{ENUM}(B)$ *au lieu d'écrire* $A \leq_{exact} B$.

Nous obtenons aussi la propriété similaire suivante pour le problème de la j-ème solution.

Lemme 15 *Soient* A, B *deux problèmes généraux (ou généraux ordonnés) avec* $A \leq_{exact} B$. *Si on a* $\text{JTH}(B) \in dyn(f,g)$ *pour un couple honnête de fonctions* (f,g), *alors on a aussi* $\text{JTH}(A) \in dyn(f,g)$.

Preuve. La preuve est similaire à celle du lemme précédent. □

1.8.4 D'autres propriétés de clôture pour l'énumération

Nous allons examiner maintenant comment se comportent les classes $\text{enum}(f,g)$ pour les opérations d'union et de produit cartésien.

Produit cartésien

Définition 65 *Soient* $A \subseteq \mathcal{I} \times \mathcal{O}$ *et* $B \subseteq \mathcal{I} \times \mathcal{O}'$ *deux problèmes généraux ayant le même espace d'entrées* $\mathcal{I}$. *On appelle produit cartésien des problèmes* A *et* B, *qu'on note* $A \times B$, *le problème général* $A \times B \subseteq \mathcal{I} \times (\mathcal{O} \times \mathcal{O}')$ *tel que* $(I,(x,y)) \in A \times B$ *si et seulement si* $(I,x) \in A$ *et* $(I,y) \in B$.

Lemme 16
- *Soient* A *et* B *deux problèmes généraux tels que* $\text{ENUM}(A) \in enum(f,g)$ *et* $\text{ENUM}(B) \in enum(f,g)$. *Alors on a aussi* $ENUM(A \times B) \in enum(f,g)$.
- *Soient* $(A, <_A)$ *et* $(B, <_B)$ *deux problèmes généraux ordonnés tels que* $\text{ENUM}(A, <_A) \in enum(f,g)$ *et* $\text{ENUM}(B, <_B) \in enum(f,g)$. *Alors on a aussi* $\text{ENUM}(A \times B, <_A \times_{lex} <_B) \in enum(f,g)$.

Preuve.

Soient $\mathcal{A}$ et $\mathcal{B}$ deux schémas d'énumération respectifs pour $\text{ENUM}(A)$ et $\text{ENUM}(B)$, tous deux de complexité $\text{enum}(f,g)$. Nous présentons ci-dessous un schéma d'énumération pour $\text{ENUM}(A \times B)$. Clairement, sa complexité est la même que celle de $\mathcal{A}$ et $\mathcal{B}$. Le schéma d'énumération est le même dans le cas où les problèmes sont ordonnés.

□

Algorithm 3 schéma d'énumération de $\textsc{Enum}(A \times B)$ sur l'entrée $I \in \mathcal{I}$

```
1: exécuter la phase de précalcul de 𝒜 sur I
2: exécuter la phase de précalcul de ℬ sur I
3: si B(I) n'est pas vide alors
4:    commencer la phase d'énumération de 𝒜
5:    tant que non fin(𝒜) faire
6:       x ← prochain(𝒜)
7:       commencer la phase d'énumération de ℬ
8:       tant que non fin(ℬ) faire
9:          y ← prochain(ℬ)
10:         mettre en sortie (x, y)
11:      fin tant que
12:   fin tant que
13: fin si
```

Union

La question que l'on se pose est de savoir si, étant donnés deux problèmes d'énumération A et B, ayant le même espace d'entrées et le même espace de solutions et appartenant tous deux à $\text{enum}(f, g)$, on peut dire quelque chose sur la complexité du problème réunion $A \cup B$. La question est ouverte dans le cas général mais nous pouvons y répondre facilement dans deux cas particuliers : l'union disjointe et l'union de deux problèmes utilisant le même ordre d'énumération.

Lemme 17 *Soient $A \subseteq \mathcal{I} \times \mathcal{O}$ et $B \subseteq \mathcal{I} \times \mathcal{O}$ deux problèmes généraux disjoints (c'est-à-dire vérifiant $A \cap B = \emptyset$) et tels que $\textsc{Enum}(A) \in enum(f, g)$ et $\textsc{Enum}(B) \in enum(f, g)$. Alors on a aussi $\textsc{Enum}(A \uplus B) \in enum(f, g)$.*

Preuve. Triviale. □

Lemme 18 *Soient $(A, <)$ et $(B, <)$ deux problèmes généraux ordonnés avec $A \subseteq \mathcal{I} \times \mathcal{O}$ et $B \subseteq \mathcal{I} \times \mathcal{O}$ et l'ordre $<$ sur $\mathcal{O}$, et tels que $\textsc{Enum}(A) \in enum(f, 1)$ et $\textsc{Enum}(B) \in enum(f, 1)$. Alors on a aussi $\textsc{Enum}(A \cup B, <) \in enum(f, 1)$.*

Preuve. Considérons un schéma d'énumération $\mathcal{A}$ (resp. $\mathcal{B}$) pour $\textsc{Enum}(A, <)$ (resp. $\textsc{Enum}(B, <$)), avec le précalcul et le délai demandés. Nous en déduisons le schéma d'énumération ci-dessous pour le problème $\textsc{Enum}(A \cup B, <)$. Sa phase d'énumération n'est rien d'autre que l'algorithme classique qui réalise la fusion de deux listes triées.

□

Algorithm 4 algorithme pour calculer ENUM$(A \cup B, <)$

exécuter la phase de précalcul de $\mathcal{A}$
exécuter la phase de précalcul de $\mathcal{B}$
commencer la phase d'énumération de $\mathcal{A}$ et la phase d'énumération de $\mathcal{B}$
$x \leftarrow$ prochain$(\mathcal{A})$
$y \leftarrow$ prochain$(\mathcal{B})$
tant que non fin$(\mathcal{A})$ et non fin$(\mathcal{B})$ **faire**
    si $x = y$ **alors**
        mettre en sortie x
        $x \leftarrow$ prochain$(\mathcal{A})$
        $y \leftarrow$ prochain$(\mathcal{B})$
    sinon si $x < y$ **alors**
        mettre en sortie x
        $x \leftarrow$ prochain$(\mathcal{A})$
    sinon
        mettre en sortie y
        $y \leftarrow$ prochain$(\mathcal{B})$
    fin si
fin tant que
si fin$(\mathcal{A})$ **alors**
    mettre en sortie tous les éléments restants de $\mathcal{B}$
sinon
    mettre en sortie tous les éléments restants de $\mathcal{A}$
fin si

Remarquons que le résultat ne se généralise apparemment pas pour $\textsc{Enum}(A, <) \in \text{enum}(f, g)$ et $\textsc{Enum}(B, <) \in \text{enum}(f, g)$ avec une fonction g quelconque. La difficulté vient du fait que, pour une même entrée I, deux solutions $x \in A(I)$ et $y \in B(I)$ peuvent être de tailles très différentes.

1.8.5 Propriétés de clôture des classes de complexité dynamique pour le problème de la j-ème solution

Nous établissons maintenant que, pour le problème de la j-ème solution, les classes $\text{dyn}(f, g)$ sont closes par produit cartésien et union disjointe.

Notre résultat de clôture pour le produit cartésien nécessite une restriction naturelle sur les problèmes étudiés.

Définition 66 (problème polynomialement borné) *Un problème général A (resp. général ordonné $(A, <)$), $A \subseteq \mathcal{I} \times \mathcal{O}$, est dit polynomialement borné si, pour toute entrée $I \in \mathcal{I}$, le nombre de solutions du problème A sur l'entrée I est polynomialement borné, autrement dit, on a $card(A(I)) \leq p(|I|)$, pour un polynôme p fixé.*

Lemme 19 *Soit $(A, <)$, $A \subseteq \mathcal{I} \times \mathcal{O}$, un problème général ordonné, polynomialement borné, et soit $\mathcal{A} = (ext, \mathcal{E})$ un schéma dynamique de complexité $dyn(f, g)$ pour le problème $\textsc{Jth}(A, <)$. Alors, toute entrée dynamique i du schéma $\mathcal{A}$ (et donc du problème $\textsc{Jth}(A, <)$) est de taille bornée par une constante ; autrement dit, il existe un entier fixé k tel que, pour toute entrée $I \in \mathcal{I}$, tout entier $i \leq card(A(I))$ s'écrit sur au plus k registres.*

Preuve. Puisqu'on a $|I| \leq |\text{ext}(I)|$, on en déduit $\text{card}(A(I)) \leq p(|I|) \leq p(|\text{ext}(I)|)$. Comme on a par ailleurs $|\text{ext}(I)| = O(B)$ pour la base B de la RAM $\mathcal{E}$ (phase dynamique) sur l'entrée $\text{ext}(I)$, on obtient finalement $\text{card}(A(I)) = O(p(B))$ □

Notre premier résultat de clôture pour le problème de la j-ème solution est le suivant.

Lemme 20 (clôture pour le produit cartésien) *Soient $(A, <_A)$ et $(B, <_B)$ deux problèmes généraux ordonnés, de même espace d'entrée $\mathcal{I}$ et polynomialement bornés. Supposons qu'on a les quatre points suivants :*

- $\textsc{Count}(A) \in Time(f)$ *et* $\textsc{Count}(B) \in Time(f)$;
- $\textsc{Jth}(A, <_A) \in dyn(f, g)$ *et* $\textsc{Jth}(B, <_B) \in dyn(f, g)$.

Alors on a aussi $\textsc{Jth}(A \times B, <_A \times_{lex} <_B) \in dyn(f, g)$.

Preuve.

Soit une entrée $I \in \mathcal{I}$ et soit un entier $i < \text{card}(A(I)) \times \text{card}(B(I))$ On fait d'abord les deux constatations suivantes :

- Il existe un entier fixé k tel que l'entier i (entrée dynamique) s'écrit sur (au plus) k registres de toute RAM qui réalise la phase dynamique d'un schéma dynamique pour le problème $\textsc{Jth}(A \times B, <_A \times_{lex} <_B)$: c'est une conséquence immédiate du lemme précédent puisque le problème $A \times B$ est, comme les problèmes A et B, polynomialement borné ;
- si le couple (x, y) est la i-ème solution de $(A \times B)(I)$ dans l'ordre $<_A \times_{lex} <_B$, alors, pour l'entier $n = \text{card}(B(I))$, l'élément x est la $(i \text{ div } n)$-ème solution de $A(I)$ dans l'ordre $<_A$ et l'élément y est la $(i \bmod n)$-ème solution de $B(I)$ dans l'ordre $<_B$;

Considérons des schémas dynamiques $\mathcal{A}$ et $\mathcal{B}$ pour les problèmes respectifs $\textsc{Jth}(A, <_A)$ et $\textsc{Jth}(B, <_B)$. On en déduit le schéma dynamique suivant $\mathcal{C}$ pour le problème "produit" $\textsc{Jth}(A \times B, <_A \times_{lex} <_B)$.

Phase statique de $\mathcal{C}$ prenant pour entrée statique I :

- exécuter la phase de précalcul de $\mathcal{A}$ avec I comme entrée statique ;
- exécuter la phase de précalcul de $\mathcal{B}$ avec I comme entrée statique ;
- calculer $n = \text{card}(B(I))$.

Phase dynamique de $\mathcal{C}$ prenant i comme entrée dynamique :

- calculer x la $(i \text{ div } n)$-ème solution de $A(I)$;
- calculer y la $(i \bmod n)$-ème solution de $B(I)$;
- renvoyer (x, y).

□

Lemme 21 (union disjointe) *Soient $(A, <)$ et $(B, <)$ deux problèmes généraux ordonnés disjoints, $A \subseteq \mathcal{I} \times \mathcal{O}$ et $B \subseteq \mathcal{I} \times \mathcal{O}$, vérifiant les conditions suivantes :*

- *pour toute instance $I \in \mathcal{I}$ et toutes solutions $y \in A(I)$ et $y' \in B(I)'$, on a $y < y'$ (toutes les solutions de $A(I)$ sont avant toutes celles de $B(I)$) ;*
- $\textsc{Count}(A) \in \mathit{Time}(f)$ *et* $\textsc{Count}(B) \in \mathit{Time}(f)$ *;*
- $\textsc{Jth}(A) \in \mathit{dyn}(f, g)$ *et* $\textsc{Jth}(B) \in \mathit{dyn}(f, g)$.

Alors, on a encore $\textsc{Jth}(A \uplus B, <) \in \mathit{dyn}(f, g)$.

Preuve. Soient $\mathcal{A}$ et $\mathcal{B}$ des schémas dynamiques pour les problèmes respectifs $\textsc{Jth}(A, <)$ et $\textsc{Jth}(B, <)$. Nous donnons un schéma dynamique $\mathcal{C}$ pour le problème $\textsc{Jth}(A \uplus B, <)$.

Phase statique de $\mathcal{C}$ prenant pour entrée statique I.

- calculer $\text{card}(A(I))$;
- exécuter la phase de précalcul de $\mathcal{A}$ avec I comme entrée statique ;
- exécuter la phase de précalcul de $\mathcal{B}$ avec I comme entrée statique.

Phase dynamique de $\mathcal{C}$ prenant l'entier i comme entrée dynamique :

- si $i < \text{card}(A(I))$ alors renvoyer la i-ème solution de $A(I)$;
- sinon renvoyer la $(i - \text{card}(A(I)))$-ème solution de $B(I)$.

□

1.9 Requêtes logiques

1.9.1 Problèmes logiques étudiés

Soient φ une formule et $\mathcal{S}$ une classe de structures. Nous nous intéressons aux cinq problèmes génériques suivants.

Model-checking

$\textsc{mc}(\varphi, \mathcal{S})$	
Entrée:	une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ
Sortie:	l'ensemble $\varphi(\mathcal{M})$ admet-il au moins un élément $(\varphi(\mathcal{M}) \neq \emptyset)$?

Comptage

$\textsc{Count}(\varphi, \mathcal{S})$	
Entrée:	une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ
Sortie:	le nombre de solutions $\text{card}(\varphi(\mathcal{M}))$

Enumération

$\textsc{Enum}(\varphi, \mathcal{S})$	
Entrée:	une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ
Sortie:	l'ensemble des solutions $\varphi(\mathcal{M})$

Enumération ordonnée, pour un certain ordre $<$ des solutions

$\textsc{Enum}(\varphi, \mathcal{S}, <)$	
Entrée:	une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ
Sortie:	la liste des solutions de $\varphi(\mathcal{M})$, présentée en ordre croissant pour l'ordre $<$

j-ème solution, pour un certain ordre $<$ des solutions

$\textsc{Jth}(\varphi, \mathcal{S}, <)$	
Entrée statique:	une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ et la cardinalité $\text{card}(\varphi(\mathcal{M}))$
Entrée dynamique:	un entier $i < \text{card}(\varphi(\mathcal{M}))$
Sortie:	la i-ème solution de $\varphi(\mathcal{M})$ dans l'ordre $<$

Nous utilisons aussi quatre variantes du problème du model-checking. Soient $\mathcal{L}$ un ensemble de formules et $\mathcal{S}$ une classe de structures. On définit les deux problèmes suivants.

MC($\mathcal{L}, \mathcal{S}$)	
Entrée:	une formule $\varphi \in \mathcal{L}$ et une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ
Sortie:	a-t-on $\varphi(\mathcal{M}) \neq \emptyset$?

p-MC($\mathcal{L}, \mathcal{S}$)	
Entrée:	une formule $\varphi \in \mathcal{L}$ et une structure $\mathcal{M} \in \mathcal{S}$ de même signature que φ
Paramètre:	$\lvert\varphi\rvert$
Sortie:	a-t-on $\varphi(\mathcal{M}) \neq \emptyset$?

Si on prend pour $\mathcal{S}$ la classe de toutes les structures, on note simplement ces problèmes MC($\mathcal{L}$) et p-MC($\mathcal{L}$), respectivement. On définit des variantes similaires des quatre autres problèmes : comptage, énumération, énumération ordonnée et j-ème solution.

1.9.2 Résultats connus sur la complexité du model-checking

Dans le cas général, le problème du model-checking est difficile.

Théorème 22 *[Var82] Les problèmes* MC(**FO**) *et* MC(**MSO**) *sont PSPACE-complets.*

A partir du résultat classique précédent, une question naturelle est de déterminer la complexité paramétrée de ces deux problèmes. Ils restent difficiles. En effet, on rappelle que le problème paramétré p-MC(**FO**) est $AW[*]$-complet, par définition, et on sait que la classe de complexité paramétrée $AW[*]$ contient toute la hiérarchie des classes $A[i]$, pour tout $i \geq 1$.

On va donc faire des restrictions.

Résultats positifs pour les formules FO ou MSO

Les résultats suivants établissent que, pour plusieurs restrictions sur les structures, le problème du "model-checking" devient **FPT** et même linéaire ou quasi-linéaire.

Pour **MSO**, on a les deux résultats suivants de Courcelle.

Théorème 23 *[Cou90b, Cou92] Soient φ une σ-formule* **MSO** *et $\mathcal{S}$ une classe de σ-structures de largeur arborescente bornée. Alors le problème* MC($\varphi, \mathcal{S}$) *peut être décidé en temps linéaire.*

Dans le cas des graphes, le théorème se généralise aux graphes de largeur de clique bornée.

Théorème 24 *[CMR00] Soient k un entier, φ une $\{E\}$-formule* **MSO** *et $\mathcal{S}$ une classe de graphes de largeur de clique au plus k, chacun représenté par une clique-décomposition de largeur k. Alors le problème $\text{MC}(\varphi, \mathcal{S})$ peut être décidé en temps linéaire.*

Dans le cas où les graphes ne sont pas donnés par leur représentation en clique-décomposition, nous pouvons appliquer le théorème suivant dû à Hlineny et Oum.

Théorème 25 *[HO07] Il existe une fonction f et un algorithme qui, pour tout graphe G et tout entier $k > 0$, vérifie les conditions suivantes :*

- *Cet algorithme fonctionne en temps $O_k(n^3)$ où n est le nombre de sommets de G ;*
- *Soit l'algorithme calcule une clique-décomposition de largeur $f(k)$, soit il certifie que l'on a $cw(G) > k$.*

De ces deux théorèmes, on déduit le théorème suivant.

Théorème 26 *Soient φ une $\{E\}$-formule* **MSO** *et $\mathcal{S}$ une classe de graphes de largeur de clique bornée. Alors le problème $\text{MC}(\varphi, \mathcal{S})$ peut être décidé en temps $O(n^3)$ où n est le nombre de sommets du graphe.*

Pour **FO**, on a les deux résultats suivants dus respectivement à Seese et à Frick et Grohe.

Théorème 27 *[See96] Soient φ une σ-formule* **FO** *et $\mathcal{S}$ une classe de σ-structures de degré borné. Alors, le problème $\text{MC}(\varphi, \mathcal{S})$ peut être décidé en temps linéaire.*

Toute classe de σ-structures de degré borné est trivialement de largeur arborescente localement bornée. Dans ce cas plus général, on obtient la complexité suivante.

Théorème 28 *[FG01b] Soient φ une σ-formule* **FO** *et $\mathcal{S}$ une classe de σ-structures de largeur arborescente localement bornée. Alors, pour tout $\epsilon > 0$, il existe un algorithme qui décide le problème $\text{MC}(\varphi, \mathcal{S})$ en temps $O_\epsilon(n^{1+\epsilon})$.*

1.9.3 Traduction et interprétation

Nous définissons ici la notion de traduction et son cas particulier, la notion d'interprétation. Cette dernière notion remonte aux travaux de Rabin [Rab64].

Définition 67 (traduction) *Soient L et L' deux classes de formules et $\mathcal{S}$ et $\mathcal{S}'$ deux classes de structures. Une traduction de $(L, \mathcal{S})$ dans $(L', \mathcal{S}')$ est une fonction $T : L \times \mathcal{S} \to L' \times \mathcal{S}'$ telle que les conditions suivantes sont réalisées, pour chaque couple $(\varphi, \mathcal{M}) \in L \times \mathcal{S}$ de même signature σ et son couple traduit $(\varphi', \mathcal{M}') = T(\varphi, \mathcal{M})$:*

- *φ' a les mêmes variables libres (FO ou MSO) que φ et la même signature σ' que $\mathcal{M}'$;*

- $\varphi'(\mathcal{M}') = \varphi(\mathcal{M})$;
- $|\varphi| \leq c|\varphi'|$, *pour une constante* c *(ne dépendant que de* T*).*

Une telle traduction est dite polynomiale (resp. linéaire) si elle est calculable en temps polynomial (resp. linéaire). On note alors $(L, \mathcal{S}) \leq_P (L', \mathcal{S}')$ *(resp.* $(L, \mathcal{S}) \leq_{lin} (L', \mathcal{S}')$*).*

Définition 68 (interprétation) *Soient* σ *une signature relationnelle et* σ' *une signature quelconque. Soient* $\mathcal{S}$ *une classe de* σ*-structures et* $\mathcal{S}'$ *une classe de* σ'*-structures. On appelle* **FO***-interprétation (resp.* **MSO***-interprétation) de* $\mathcal{S}$ *dans* $\mathcal{S}'$ *un tuple* $I = (t_I, \Delta, (\Theta_R)_{R\in\sigma}))$ *constitué par :*

- *une fonction* $t_I : \mathcal{S} \to \mathcal{S}'$ *qui transforme toute* σ*-structure* $\mathcal{M}$ *en une* σ'*-structure* $\mathcal{M}'$;
- *des* σ'*-formules* **FO** *(resp.* **MSO***) :*
  - *pour le domaine, une formule* $\Delta(x)$ *à une variable libre du premier ordre et,*
  - *pour chaque symbole relationnel* $R \in \sigma$ *d'arité* k*, une formule* $\Theta_R(x_1, \ldots, x_k)$ *à* k *variables libres du premier ordre.*

Ces objets doivent satisfaire les conditions suivantes : pour toute structure $\mathcal{M} \in \mathcal{S}$ *et sa structure associée* $\mathcal{M}' = t_I(\mathcal{M})$*, les formules introduites permettent de décrire le domaine et les relations de* $\mathcal{M}$ *à l'intérieur de la structure* $\mathcal{M}'$ *:*

1. *pour le domaine :* $Dom(\mathcal{M}) = \Delta(\mathcal{M}')$*, et*
2. *chaque prédicat* k*-aire* $R \in \sigma$ *a l'interprétation* $R^{\mathcal{M}} = \Theta_R(\mathcal{M}') \cap Dom(\mathcal{M})^k$*.*

Une interprétation I *de* $\mathcal{S}$ *dans* $\mathcal{S}'$ *est dite polynomiale (resp. linéaire) si sa fonction de transformation* $t_I : \mathcal{S} \to \mathcal{S}'$ *est calculable en temps polynomial (resp. linéaire).*

Si une telle interprétation existe, on note $\mathcal{S} \leq_{FO,lin} \mathcal{S}'$ *(resp.* $\mathcal{S} \leq_{FO,P} \mathcal{S}'$*,* $\mathcal{S} \leq_{MSO,lin} \mathcal{S}'$*,* $\mathcal{S} \leq_{MSO,P} \mathcal{S}'$*).*

Le lemme suivant exprime que toute interprétation définit une traduction de même complexité.

Lemme 29 *Soient* $\mathcal{S}$ *et* $\mathcal{S}'$ *deux classes de structures. Toute* **FO***-interprétation (resp.* **MSO***-interprétation)* I *de* $\mathcal{S}$ *dans* $\mathcal{S}'$ *définit une traduction* T *de* $(\mathbf{FO}, \mathcal{S})$ *dans* $(\mathbf{FO}, \mathcal{S}')$ *(resp. de* $(\mathbf{MSO}, \mathcal{S})$ *dans* $(\mathbf{MSO}, \mathcal{S}')$*).*

De plus, si l'interprétation I *est polynomiale (resp. linéaire), alors la traduction associée* T *est aussi polynomiale (resp. linéaire).*

Preuve. Supposons que I est une **MSO**-interprétation de $\mathcal{S}$ dans $\mathcal{S}'$. (Le cas où I est une **FO**-interprétation est similaire mais plus simple.) I est donc de la forme $I = (t_I, \Delta, (\Theta_R)_{R\in\sigma})$. On associe à I la traduction $T : \mathbf{MSO} \times \mathcal{S} \to \mathbf{MSO} \times \mathcal{S}'$ suivante. Pour chaque σ-formule **MSO** $\varphi(X_1, \ldots, X_p)$ de variables libres $X_1, \ldots, X_p$, toutes du second ordre, et chaque σ-structure $\mathcal{M} \in \mathcal{S}$, on définit la traduction $T(\varphi, \mathcal{M}) = (\varphi', \mathcal{M}')$ comme suit :

- $\mathcal{M}' = t_I(\mathcal{M})$;
- $\varphi'(X_1, \ldots, X_p) \equiv \varphi''(X_1, \ldots, X_p) \wedge \bigwedge_{i \in [p]} \forall x(X_i(x) \to \Delta(x))$.

Ici, $\varphi''(\bar{X})$ est la formule obtenue à partir de $\varphi(\bar{X})$ en faisant les substitutions suivantes :

- chaque atome $R(x_1, \ldots, x_k)$ de φ, $R \in \sigma$, est remplacé par la formule $\Theta_R(x_1, \ldots, x_k)$;
- chaque quantification du premier ordre $\exists x$ de φ est remplacée par $\exists x(\Delta(x) \wedge \ldots)$ (formulation intuitive : $\exists x \in \Delta$) ;
- chaque quantification du second ordre $\exists X$ de φ est remplacée par

$$\exists X(\forall x(X(x) \to \Delta(x)) \wedge \ldots)$$

(formulation intuitive : $\exists X \subseteq \Delta$).

A partir des conditions (1) $\mathrm{Dom}(\mathcal{M}) = \Delta(\mathcal{M}')$ et (2) $R^{\mathcal{M}} = \Theta_R(\mathcal{M}') \cap \mathrm{Dom}(\mathcal{M})^k$ vérifiées par l'interprétation I, on en déduit, par induction sur la structure de la formule φ, l'égalité $\varphi(\mathcal{M}) = \varphi''(\mathcal{M}') \cap \mathcal{P}(\mathrm{Dom}(\mathcal{M}))^p$, d'où on tire finalement $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$.

Complexité : on vérifie facilement que l'on a $|\varphi'| \leq c|\varphi|$, pour une constante c (dépendant seulement de I), et que φ' est calculable à partir de φ en temps $O(|\varphi|)$. De plus, si I est linéaire (resp. polynomiale), c'est-à-dire si t_I est calculable en temps linéaire (resp. polynomiale), alors T l'est aussi. □

Le lemme suivant est immédiat.

Lemme 30 *Soient L et L' deux classes de formules et $\mathcal{S}$ et $\mathcal{S}'$ deux classes de structures. Supposons qu'il existe une traduction linéaire T de $(L, \mathcal{S})$ dans $(L', \mathcal{S}')$.*

Alors, on peut calculer, pour toute formule $\varphi \in L$, une formule $\varphi' \in L'$ et une réduction exacte du problème $\textsc{Enum}(\varphi, \mathcal{S})$ *au problème* $\textsc{Enum}(\varphi', \mathcal{S}')$ *avec, pour toute structure $\mathcal{M} \in \mathcal{S}$ et pour $T(\varphi, \mathcal{M}) = (\varphi', \mathcal{M}')$, $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$.*

En conséquence, si on a $\textsc{Enum}(\varphi', \mathcal{S}') \in enum(f, g)$*, alors on a aussi* $\textsc{Enum}(\varphi, \mathcal{S}) \in enum(f, g)$.

Ces deux derniers lemmes vont nous être essentiels, dans toute la thèse, pour transférer la complexité d'énumération $\mathrm{enum}(f, g)$ d'un problème de requêtes à un autre problème de requêtes.

Chapitre 2

Evaluation de requêtes conjonctives

Sommaire

2.1 Introduction

Ce chapitre étudie la complexité du calcul et de l'énumération des requêtes *conjonctives* et, tout particulièrement, des requêtes *conjonctives acycliques* et de leurs généralisations. Une requête conjonctive est acyclique (**ACQ**) si son hypergraphe est acyclique. Les requêtes conjonctives acycliques ont été introduites dès 1981 par Yannakakis dans [Yan81] et ont aussi été étudiées, entre autres, dans [BFMY83] et [GKS06].

Yannakakis a donné un algorithme d'évaluation d'une requête conjonctive acyclique φ sur une structure quelconque $\mathcal{M}$, dont le temps est $O(|\varphi| \cdot |\mathcal{M}| \cdot |\varphi(\mathcal{M})|$. La classe des requêtes **ACQ** (conjonctives acycliques) a été généralisée par Papadimitriou et Yannakakis [PY99] en autorisant les inégalités $\neq$: on définit ainsi la classe $\mathbf{ACQ}^{\neq}$. Ils ont démontré que le problème de l'évaluation d'une requête $\mathbf{ACQ}^{\neq}$, même s'il devient **NP**-difficile, est **FPT** et, plus précisément, est calculable en temps $f(|\varphi|) \cdot |\mathcal{M}| \cdot |\varphi(\mathcal{M})| \cdot (\log |\mathcal{M}|)^2$.

Dans ce chapitre comme dans notre article [BDG07], nous améliorons d'abord cette borne en temps. Nous démontrons que le même problème d'évaluation est de complexité en temps $f(|\varphi|) \cdot |\mathcal{M}| \cdot |\varphi(\mathcal{M})|$. Pour cela, nous introduisons une technique d'élimination des quantificateurs qui est basée sur des outils combinatoires où la notion de couverture joue un rôle central. Ces outils s'apparentent à des notions similaires déjà utilisées par Monien [Mon85] pour tester l'existence d'un chemin de longueur k dans un graphe. Ce problème est d'ailleurs un cas particulier de requêtes $\mathbf{ACQ}^{\neq}$.

Notre seconde contribution concerne le problème de l'énumération des solutions d'une requête conjonctive acyclique ou acyclique généralisée, c'est-à-dire la question du délai entre deux solutions produites. Dans le cas général, on démontre qu'il est possible d'énumérer les solutions d'une requête $\mathbf{ACQ}^{\neq}$ fixée, à délai linéaire en la taille de la structure. On se pose ensuite la question de déterminer pour quelles requêtes conjonctives les solutions sont énumérables à *délai constant*. Nous établissons d'abord qu'on peut énumérer à délai constant après précalcul linéaire les solutions de toute requête $\mathbf{ACQ}^{\neq}$ sans quantificateur. On se ramène pour cela à un problème combinatoire basé encore sur la notion de couverture. Nous cherchons ensuite à étendre au maximum ces résultats d'énumération. Pour cela, nous introduisons la classe des requêtes

conjonctives connexe-acycliques, notée $\mathbf{CCQ}^{\neq}$, qui est basée sur une notion de connexité qui fait intervenir les variables libres de la requête. Nous établissons pour $\mathbf{CCQ}^{\neq}$ le même résultat de complexité d'énumération optimal (CONSTANT-DELAY$_{lin}$) que précédemment. Nous démontrons aussi que la classe des formules $\mathbf{CCQ}^{\neq}$ est, d'une certaine manière, maximale pour ce résultat. Si l'on fait l'hypothèse que le produit de deux matrices booléennes $n \times n$ n'est pas calculable en temps $O(n^2)$, alors on en déduit que toute requête *simple* (c'est-à-dire où aucun symbole de relation n'est repété) appartenant à $\mathbf{ACQ}^{\neq} - \mathbf{CCQ}^{\neq}$ (ou à $\mathbf{ACQ} - \mathbf{CCQ}$) n'est pas calculable à délai constant après précalcul linéaire. Ces résultats permettent de classifier la complexité des requêtes conjonctives acycliques (avec ou sans inégalités) : nous obtenons un résultat de dichotomie. Par ailleurs, nous donnons un algorithme linéaire opérant sur l'hypergraphe d'une formule conjonctive quelconque et qui permet de déterminer si une telle formule est $\mathbf{CCQ}^{\neq}$ ou non. Ceci montre que notre classification dichotomique est effective. Notons au passage que les résultats précédents sont nouveaux et pertinents, même pour des formules sans inégalité.

Par ailleurs, on s'intéresse à une classe particulière de requêtes $\mathbf{ACQ}^{\neq}$, appelées $\mathbf{ACQ}^{all\neq}$, pour lesquelles on demande que les inégalités forment une clique. Cette classe est intéressante car elle permet d'exprimer au plus juste plusieurs problèmes **NP**-complets naturels : génération des chemins de longueur k dans un graphe, "multi-dimensional matching", etc. A partir d'une procédure d'élimination des quantificateurs adaptée, nous déduisons, pour ces problèmes, des bornes de complexité en temps plus fines que celles résultant de la méthode générale pour $\mathbf{ACQ}^{\neq}$.

Enfin, on cherche à déterminer des classes de requêtes énumérables à délai constant après prétraitement polynomial. Pour cela, nous introduisons la notion de *largeur arborescente connexe*. Nous démontrons que les solutions d'une requête conjonctive de largeur arborescente connexe k peuvent être énumérées à délai constant après prétraitement en temps $O(\text{card}(\text{Dom}(\mathcal{M}))^{k+1})$ et donc en temps total $O(\text{card}(\text{Dom}(\mathcal{M}))^{k+1} + |\varphi(\mathcal{M})|)$. Ce résultat est à comparer avec le résultat de [CR00] et [FFG02] énonçant que, pour une formule conjonctive fixée φ, de largeur arborescente k, l'évaluation de la requête $\varphi(\mathcal{M})$ est réalisée en temps $O(\text{card}(\text{Dom}(\mathcal{M}))^{k+1} \cdot |\varphi(\mathcal{M})|)$. Là encore, pour tout entier k fixé, nous donnons un algorithme polynomial qui détermine si une formule conjonctive est de largeur aborescente connexe k.

2.2 Préliminaires

Nous présentons maintenant les notions de base étudiées dans ce chapitre, notamment les formules conjonctives, avec ou sans inégalités, acycliques ou non.

Définition 69 (formule conjonctive, formule conjonctive avec inégalités) *Soit σ une signature relationnelle. Une σ-formule est dite conjonctive ou **CQ** (resp. conjonctive avec inégali-*

tés ou $\boldsymbol{CQ}^{\neq}$*) si elle est de la forme* $\varphi(\bar{x}) \equiv \exists\bar{y}\psi(\bar{x},\bar{y})$ *où* ψ *est une conjonction d'atomes* $R(\bar{v})$ *pour* $\bar{v} \subseteq \{\bar{x},\bar{y}\}$ *et* $R \in \sigma$ *(resp. d'atomes* $R(\bar{v})$ *pour* $\bar{v} \subseteq \{\bar{x},\bar{y}\}$ *et d'inégalités* $u \neq v$ *pour* $u, v \in \{\bar{x},\bar{y}\}$*)).*

Il est classique d'associer à chaque formule conjonctive son hypergraphe, outil essentiel dans l'analyse de la formule.

Définition 70 (hypergraphe associé à une formule conjonctive) *Soit* φ *une formule conjonctive (avec ou sans inégalités). On appelle hypergraphe associé à* φ*, l'hypergraphe* $H_\varphi = (V_\varphi, E_\varphi)$ *dont l'ensemble des sommets est* $V_\varphi = var(\varphi)$ *et l'ensemble des hyperarêtes est* $E_\varphi = \{\{v_1,\ldots,v_p\} : R(v_1,\ldots,v_p)$ *est un atome de* $\varphi\}$*.*

La notion principale de ce chapitre est celle de requête acyclique.

Définition 71 (formule conjonctive acyclique, arbre de jointure) *Une formule conjonctive* φ *(avec ou sans inégalités) est* acyclique *ou* $\boldsymbol{ACQ}$ *(resp.* $\boldsymbol{ACQ}^{\neq}$*) si son hypergraphe* H_φ *est acyclique. Une structure arborescente de* H_φ *est appelée un* arbre de jointure *de* φ*.*

Exemple 31 $\varphi(x,y) \equiv \exists z\exists t\exists u(Rxyz \wedge Ryzt \wedge Rztu \wedge Syt \wedge x \neq u)$ *est une formule* $\boldsymbol{ACQ}^{\neq}$ *dont un arbre de jointure* T_φ *est représenté par la figure ci-dessous.*

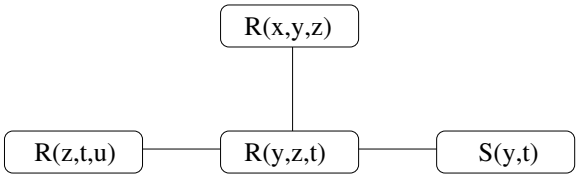


FIG. 2.1 – un arbre de jointure de φ

Dès 1984, Tarjan et Yannakakis [TY84, TY85] ont établi que cette propriété d'acyclicité est décidable en temps linéaire.

Théorème 32 *[TY84, TY85] Il existe un algorithme linéaire qui, pour tout hypergraphe* H *(resp. toute formule conjonctive* φ*), décide si* H *(resp.* φ*) est acyclique et, si oui, calcule une structure arborescente de* H *(resp. un arbre de jointure de* φ*).*

2.3 Des requêtes relationnelles aux requêtes fonctionnelles

Dans le souci de simplifier les notations et les preuves qui vont suivre, tant pour l'élimination des quantificateurs que pour l'énumération des solutions d'une requête, nous allons nous placer

dans un cadre fonctionnel, c'est-à-dire un cadre logique qui n'utilise que des fonctions et des prédicats unaires. Nous introduisons pour cela les requêtes fonctionnelles conjonctives et leur version acyclique.

2.3.1 Formules fonctionnelles conjonctives acycliques

Définition 72 (formule fonctionnelle conjonctive) *Soit σ une signature fonctionnelle, c'est-à-dire une signature composée uniquement de symboles de fonctions et de prédicats unaires. Une σ-formule est dite fonctionnelle conjonctive ou* ***F-CQ*** *(resp.* ***F-CQ$^{\neq}$****) si elle est de la forme $\varphi = \exists \bar{x} \psi$ où ψ est une conjonction d'atomes $U(v)$ et d'égalités $\tau = \tau'$ (resp. d'égalités $\tau = \tau'$ et d'inégalités $\tau \neq \tau'$) et chaque terme τ ou τ' est de la forme $f(v)$ où v est une variable, U un prédicat unaire et f un symbole de fonction unaire.*

Comme on a associé à toute formule conjonctive son hypergraphe, on associe maintenant à toute formule fonctionnelle conjonctive son graphe.

Définition 73 (graphe de structure) *A chaque formule* ***F-CQ*** *ou* ***F-CQ$^{\neq}$*** *φ, on associe le graphe de structure (non orienté) $G_\varphi = (V_\varphi, E_\varphi)$ défini de la manière suivante :*

- *$V_\varphi = var(\varphi)$ et,*
- *pour chaque paire de variables distinctes v, v', on a l'arête $\{v, v'\} \in E_\varphi$ si et seulement si une égalité [2] faisant invervenir à la fois v et v', donc de la forme $f(v) = f'(v')$, apparaît dans φ.*

Définition 74 (formule fonctionnelle conjonctive acyclique) *Une formule* ***F-CQ*** *(resp.* ***F-CQ$^{\neq}$****) φ est acyclique ou* ***F-ACQ*** *(resp.* ***F-ACQ$^{\neq}$****) si son graphe de structure G_φ est acyclique. Sans perte de généralité, on supposera que chaque graphe G_φ est également connexe ; le graphe G_φ est alors appelé l'arbre de la formule φ qu'on notera donc T_φ.*

2.3.2 Traduction des requêtes relationnelles en requêtes fonctionnelles

La transformation des requêtes acycliques que nous allons décrire dans cette section est très semblable à la traduction du calcul relationnel sur domaines (domain relational calculus) dans le calcul relationnel sur n-uplets (tuple relational calculus) dans le cadre classique de la théorie des bases de données. L'idée est très simple, même s'il est nécessaire d'examiner soigneusement tous les détails.

[2] Les inégalités de φ n'interviennent pas dans la définition de G_φ

Traduction de la structure

L'idée de base consiste, étant donnée une structure relationnelle $\mathcal{M}$, à construire une structure fonctionnelle $\mathcal{M}'$ où chaque sommet de $\mathcal{M}'$ représente un tuple d'une relation de $\mathcal{M}$. De façon plus précise, chaque σ-structure relationnelle $\mathcal{M} = (D; R_1, \ldots, R_s)$, où $R_1, \ldots, R_s$ sont des relations, est traduite par la σ'-structure fonctionnelle $\mathcal{M}' = (D'; D, D_{R_1}, \ldots, D_{R_s}, f_1, \ldots, f_p, \mathrm{Id})$ ainsi décrite :

- $D, D_{R_1}, \ldots, D_{R_s}$ sont des relations unaires disjointes, dont le domaine D est la réunion $D' = D \uplus D_{R_1} \uplus \ldots \uplus D_{R_s} \uplus \{\perp\}$ avec un élément additionnel $\perp$; pour chaque $i \in [s]$, on prend $D_{R_i} = R_i$, i.e. D_{R_i} est l'ensemble des tuples de R_i.
- Chaque f_j ($j \in [p]$) est une fonction unaire : $D' \to D$ telle que, pour chaque $t = (a_1, \ldots, a_{p_i}) \in D_{R_i}$, on a $f_j(t) = a_j$ pour $j \in [p_i]$ et $f_j(t) = \perp$ sinon.
- Id représente la fonction identité.

On voit facilement que la structure fonctionnelle $\mathcal{M}'$ contient toute l'information portée par la structure $\mathcal{M}$. On décrit maintenant comment traduire une formule en accord avec la traduction de la structure.

Traduction d'une formule acyclique sans quantificateur (traduction 1)

Le principe de la traduction est d'associer à une formule (relationnelle) conjonctive acyclique ψ sans quantificateur une formule fonctionnelle acyclique sans quantificateur, dont les variables sont les atomes de ψ et dont l'arbre associé est l'arbre de jointure de ψ.

Soit $\psi(\bar{x}) \equiv \bigwedge_{i \in [q]} A_i$ une conjonction d'atomes A_i qui est une σ-formule **ACQ**. On associe à ψ la σ'-formule sans quantificateur suivante :

$$\psi'(\alpha_1, \ldots, \alpha_q) \equiv \bigwedge_{i \in [q]} D_{R_i}(\alpha_i) \wedge \bigwedge_{(i,j) \in T_\psi} \bigwedge_{\substack{h,k \in [p] \\ \mathrm{var}_h(A_i) = \mathrm{var}_k(A_j)}} f_h(\alpha_i) = f_k(\alpha_j)$$

où $\sigma' = \{D_R : R \in \sigma\} \cup \{\mathrm{Id}, f_1, \ldots, f_p\}, p = \mathrm{arity}(\sigma) = \max_{R \in \sigma} \mathrm{arity}(R)$, T_ψ est l'arbre de jointure de ψ, $A_i \equiv R_i(v_i^1, \ldots, v_i^h, \ldots, v_i^{p_i})$, p_i est l'arité de R_i et $\mathrm{var}_h(A_i)$ désigne la variable v_i^h qui apparaît à l'indice h dans l'atome A_i.

Exemple 33 *La traduction 1 transforme la σ-formule **ACQ** sans quantificateur $\psi(x, y, z) \equiv R(x, y) \wedge S(y, z)$ en la σ'-formule **F-ACQ** sans quantificateur : $\psi'(\alpha_1, \alpha_2) \equiv D_R(\alpha_1) \wedge D_S(\alpha_2) \wedge f_2(\alpha_1) = f_1(\alpha_2)$.*

Traduction d'une formule acyclique (traduction 2)

Considérons une formule **ACQ** $\varphi(\bar{x}) \equiv \exists \bar{y} \psi(\bar{x}, \bar{y})$ où ψ est une formule sans quantificateur et $\bar{x} = (x_1, \ldots, x_m)$. On associe à φ la σ'-formule fonctionnelle suivante :

$$\varphi''(\bar{x}) \equiv \exists \alpha_1 \ldots \exists \alpha_q (\psi'(\bar{\alpha}) \wedge \bigwedge_{i \in [m]} f_{j_i}(\alpha_{k_i}) = \mathrm{Id}(x_i))$$

où ψ' est la traduction (1) de la matrice ψ de φ et A_{k_i} (représenté par la variable α_{k_i}) est un atome, arbitrairement choisi, de φ dans lequel x_i apparaît (à l'indice j_i).

Exemple 34 *La traduction 2 transforme la σ-formule $\varphi(x, z) \equiv \exists y (R(x, y) \wedge S(y, z))$ de **ACQ** en la σ'-formule **F-ACQ** suivante, avec les mêmes variables libres que φ et la sous-formule ψ' ci-dessus : $\varphi''(x, z) \equiv \exists \alpha_1 \exists \alpha_2 (\psi'(\alpha_1, \alpha_2) \wedge f_1(\alpha_1) = x \wedge f_2(\alpha_2) = z)$.*

Remarque 13 *Les traductions (1) et (2) peuvent être aisément étendues aux $\mathbf{ACQ}^{\neq}$ formules.*

Le lemme suivant exprime que les traductions 1 et 2 ont les propriétés attendues.

Lemme 35 *1. Préservation de l'acyclicité : Soit φ une formule $\mathbf{CQ}^{\neq}$. Si φ est acyclique ($\mathbf{ACQ}^{\neq}$) alors ses traductions fonctionnelles 1 (dans le cas où φ est sans quantificateur) et 2 sont également acycliques, c'est-à-dire sont $\mathbf{F\text{-}ACQ}^{\neq}$.*

2. Linéarité des transformations :
- *Pour les formules : les tailles des formules transformées 1 et 2 sont linéaires en la taille de φ.*
- *Pour les structures : la transformation de la σ-structure relationnelle $\mathcal{M}$ en la σ'-structure fonctionnelle $\mathcal{M}'$ est calculable en temps $O(|\mathcal{M}|)$.*

3. Correction des réductions :
- *Traduction 1 : Soit ψ' la formule $\mathbf{F\text{-}ACQ}^{\neq}$ sans quantificateur associée à la formule sans quantificateur ψ de $\mathbf{ACQ}^{\neq}$. Il existe une réduction exacte du problème* Enum(ψ) *au problème* Enum(ψ').
- *Traduction 2 : On a $\varphi(\mathcal{M}) = \varphi''(\mathcal{M}')$.*

Preuve. 1) Dans le cas de la traduction 1, on constate que si ψ' contient une égalité $f_h(\alpha_i) = f_k(\alpha_j)$ qui fait intervenir deux variables α_i et α_j (associées aux atomes A_i et A_j), alors, par construction, les deux noeuds A_i et A_j sont adjacents dans l'arbre de jointure T_ψ. Par conséquent, le graphe $G_{\psi'}$ de ψ' est isomorphe à un graphe partiel de T_ψ et est donc aussi acyclique. La traduction 2 contient, par rapport à la traduction 1, les variables supplémentaires $x_1, \ldots x_m$, chacune ajoutée comme feuille au graphe acyclique précédent. L'acyclicité est donc préservée.

2) Ce point est facile à vérifier.

3) Traduction 1 : On définit la réduction (r, s) suivante du problème ENUM(ψ) au problème ENUM(ψ') pour la formule $\mathbf{ACQ}^{\neq}$ $\psi(x_1, \ldots, x_m)$ sans quantificateur et sa traduction $\psi'(\alpha_1, \ldots, \alpha_q)$ dans $\mathbf{F\text{-}ACQ}^{\neq}$:

- fonction r : on prend $r(\mathcal{M}) = \mathcal{M}'$ où $\mathcal{M}'$ est la structure fonctionnelle qui traduit la structure relationnelle $\mathcal{M}$;
- fonction s : on prend $s(\mathcal{M}', (\alpha_1, \ldots, \alpha_q)) = (f_{j_1}(\alpha_{k_1}), \ldots, f_{j_m}(\alpha_{k_m}))$ où, comme on l'a déjà écrit, la variable x_i ($i \in [m]$) apparaît à l'indice j_i dans l'atome A_{k_i} de ψ.

Le couple (r, s) est une réduction parcimonieuse du problème ENUM(ψ) au problème ENUM(ψ'), c'est-à-dire la transformation

$$(\alpha_1, \ldots, \alpha_q) \mapsto (f_{j_1}(\alpha_{k_1}), \ldots, f_{j_m}(\alpha_{k_m}))$$

est une bijection de $\psi'(\mathcal{M}')$ dans $\psi(\mathcal{M})$. Par souci de simplicité et pour simplifier les notations, vérifions le seulement sur l'exemple déjà vu, exemple qu'on peut facilement généraliser. Soit une σ-structure $\mathcal{M} = (D, R, S)$ et soit $\mathcal{M}' = (D'; D, D_R, D_S, f_1, f_2, \mathrm{Id})$ sa σ'-structure associée. Vérifions donc que la correspondance

$$(\alpha_1, \alpha_2) \mapsto s(\mathcal{M}', (\alpha_1, \alpha_2)) = (f_1(\alpha_1), f_2(\alpha_1), f_2(\alpha_2))$$

est une bijection de l'ensemble $\psi'(\mathcal{M}')$ dans l'ensemble $\psi(\mathcal{M})$:

- en effet, c'est une fonction de $\psi'(\mathcal{M}')$ dans l'ensemble $\psi(\mathcal{M})$: pour toute solution $(\alpha_1, \alpha_2) \in \psi'(\mathcal{M}')$ et pour $(x, y, z) = (f_1(\alpha_1), f_2(\alpha_1), f_2(\alpha_2))$, on a $\alpha_1 = (x, y) \in R$ et $\alpha_2 = (f_1(\alpha_2), f_2(\alpha_2)) \in S$; or, comme on a par ailleurs $f_2(\alpha_1) = f_1(\alpha_2) = y$, on peut écrire $(y, z) \in S$; d'où on tire $(x, y, z) \in \psi(\mathcal{M})$;
- la fonction est bijective : chaque solution $(x, y, z) \in \psi(\mathcal{M})$ a pour antécédent unique la solution $((x, y), (y, z)) \in \psi'(\mathcal{M}')$.

Comme les conditions de linéarité sont aussi vérifiées, le couple (r, s) est donc une réduction exacte de ENUM(ψ) à ENUM(ψ').

Traduction 2 : On déduit $\varphi(\mathcal{M}) = \varphi''(\mathcal{M}')$ en s'appuyant sur les mêmes arguments que pour la traduction 1. □

2.4 Elimination des quantificateurs dans les formules acycliques

Nous allons évaluer chaque requête conjonctive acyclique (avec ou sans inégalités) par une procédure inductive d'élimination des quantificateurs. L'induction est réalisée sur l'arbre de la formule en éliminant successivement chaque variable en position de feuille. Nous décrivons d'abord une telle procédure pour les formules **F-ACQ** avant de la donner dans le cas plus élaboré des formules $\mathbf{F\text{-}ACQ}^{\neq}$.

2.4.1 Evaluation des formules acycliques sans inégalité

Afin de présenter au mieux la procédure inductive d'élimination des quantificateurs et surtout d'en justifier précisément la complexité combinée, par rapport à la structure et à la formule, nous introduisons maintenant la notion d'arbre à égalités d'une formule. Cet arbre représente à la fois la structure de la formule et sa complexité, c'est-à-dire sa taille.

Définition 75 (arbre à égalités) *L'arbre à égalités d'une formule* ***F-ACQ*** φ *est un couple* (T, λ) *où* T *est l'arbre de* φ *et* λ *est une fonction qui, à toute arête* $\{x, y\} \in E(T)$*, associe l'étiquette constituée de l'ensemble des égalités de* φ *de la forme* $f(x) = g(y)$*. S'il n'y a pas d'ambiguïté, on notera souvent* $T = (T, \lambda)$ *un arbre à égalités.*

Remarque 14 *L'arbre à égalités* $T = (T, \lambda)$ *de* φ *contient toute l'information sur* φ*, à l'exception de ses quantificateurs et de ses atomes unaires* $U(x)$*. Sans perte de généralité, on peut supposer que les tailles de* φ *et de* T *sont du même ordre :* $|\varphi| = \Theta(|T|)$*. En effet, on peut supposer que chaque variable* x *n'apparaît que dans (au plus) un atome unaire* $U(x)$ *de* φ*, ceci en amalgamant en un seul atome unaire toute conjonction d'atomes unaires sur* x *(même chose pour les égalités* $f(x) = g(x)$ *ou les inégalités* $f(x) \neq g(x)$*). En conséquence, le nombre d'atomes unaires de* φ *est au plus* $card(var(\varphi)) \leq |T|$*. Le nombre de quantificateurs de* φ *est évidemment borné de la même façon.*

A l'élimination d'une variable y d'une formule, en position de feuille, correspond évidemment l'élagage de la feuille y dans l'arbre à égalités de la formule.

Définition 76 (arbre élagué) *Soit* T *un arbre à égalités où* y *est une feuille et* x *est le parent de* y*. L'arbre* y*-élagué de* T*, noté* $T - \{y\}$*, est l'arbre à égalités* T *où l'on supprime le noeud* y *et l'arête étiquetée* $\{x, y\}$*.*

Le lemme suivant décrit l'élimination d'un quantificateur $\exists z$, dans une formule φ d'arbre à égalités T pour une variable-feuille z de l'arbre T. L'arbre à égalités de la formule obtenue par élimination de z est l'arbre élagué $T - \{z\}$.

Lemme 36 *Soit* $\varphi(\bar{x})$ *une* σ*-formule* ***F-ACQ*** *d'arbre à égalités* T *et de la forme* $\varphi(\bar{x}) = \exists z \psi(\bar{x}, z)$*, où* ψ *est sans quantificateur et* z *est une feuille de* T *ayant pour parent* y*. Soit* m_{yz} *le nombre d'égalités de la forme* $f(y) = g(z)$ *dans* $\varphi(\bar{x})$ *et soit* $\mathcal{M}$ *une* σ*-structure de domaine* D*. Alors on peut calculer en temps* $O(m_{yz} \cdot card(D))$ *une* σ'*-formule sans quantificateur* $\varphi'(\bar{x})$*, avec* $\sigma \subseteq \sigma'$*, et une* σ'*-structure* $\mathcal{M}'$ *qui étend* $\mathcal{M}$*, telles que*

- $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ *;*
- φ' *est une formule* ***F-ACQ*** *ayant pour arbre à égalités* $T - \{z\}$*.*

Preuve. Considérons une σ-formule $\varphi(\bar{x})$ dans **F-ACQ** qui satisfait les hypothèses de l'énoncé. On note $m = m_{yz}$. La formule φ peut être mise sous la forme suivante :

$$\varphi(\bar{x}) \equiv \psi_0(\bar{x}) \wedge \exists z(P(z) \wedge \bigwedge_{i\in[m]} g_i(z) = g'_i(y))$$

où $\psi_0(\bar{x})$ est une formule $\mathbf{F\text{-}ACQ}^{\neq}$ sans quantificateur, y est le parent de z dans l'arbre T, $y \in \bar{x}$ et $P(z)$ est un atome unaire sur z. On note $\bar{g}(z) = (g_1(z), \dots g_m(z))$, $\bar{g'}(y) = (g'_1(y), \dots g'_m(y))$. Soit $P = P(\mathcal{M})$. Considérons l'ensemble $Q = \bar{g'}^{-1}(\bar{g}(P))$. Cet ensemble Q, qu'on va considérer comme une nouvelle relation unaire, peut aisément être calculé en temps $O(m_{yz} \cdot \text{card}(D))$. Considérons maintenant la σ'-formule

$$\varphi'(\bar{x}) \equiv \psi_0(\bar{x}) \wedge Q(y)$$

où $\sigma' = \sigma \cup \{Q\}$ et $\mathcal{M}'$ est la σ'-expansion de $\mathcal{M}$ avec la relation unaire Q. Il est facile de constater que φ' et $\mathcal{M}'$ vérifient tous les points demandés. □

Remarque 15 *Si la conjonction $\psi_0(\bar{x})$ contient un atome de la forme $U(y)$, il est facile d'amalgamer les prédicats unaires U et Q en un seul prédicat unaire U'. La structure $\mathcal{M}'$ est modifiée en conséquence en prenant $U' = U \cap Q$.*

Proposition 37 *Soit $\varphi(x)$ une σ-formule **F-ACQ** de seule variable libre x et soit $\mathcal{M}$ une σ-structure. Alors on peut évaluer la requête $\varphi(\mathcal{M})$ en temps total $O(m \cdot card(D))$ où D est le domaine de $\mathcal{M}$ et m est le nombre d'égalités de φ.*

Preuve. La première étape consiste à éliminer itérativement chaque feuille de φ, hormis x, en utilisant le lemme 36, jusqu'à aboutir à une formule $\varphi'(x)$ sans quantificateur et à une structure $\mathcal{M}'$ telles que $\varphi'(\mathcal{M}') = \varphi(\mathcal{M})$. Le temps pour supprimer une feuille z de parent y dans T est $O(m_{yz}\text{card}(D))$ où m_{yz} est le nombre d'égalités de la forme $f(y) = g(z)$. En conséquence, le temps total pour supprimer toutes les variables sauf x est borné par $O(\sum_{\{y,z\}\in E(T)} m_{yz}\text{card}(D)) = O(m \cdot \text{card}(D))$. Puisque la formule sans quantificateur φ' se réduit à un atome $U(x)$, la requête $\varphi'(\mathcal{M}')$ peut donc aisément être évaluée en temps $O(\text{card}(D))$. □

Corollaire 38 *Soient φ une σ-formule **F-ACQ** et $\mathcal{M}$ une σ-structure. Alors, on peut décider si on a $\varphi(\mathcal{M}) \neq \emptyset$ en temps $O(m \cdot card(D))$ où D est le domaine de $\mathcal{M}$ et m le nombre d'égalités de φ.*

2.4.2 Couvertures et ensembles représentants

Nous voulons maintenant effectuer l'élimination des quantificateurs dans les formules $\mathbf{F\text{-}ACQ}^{\neq}$, c'est-à-dire en présence d'inégalités. Pour cela, nous avons besoin d'introduire de nouveaux ou-

tils combinatoires : les couvertures et les ensembles représentants. Nous introduisons d'abord quelques conventions de notations.

Notations

- Dans toutes les définitions et résultats qui vont suivre, on désigne par E et F deux ensembles finis et $\bar{f} = (f_1, \ldots, f_k)$ est un k-tuple de fonctions de E dans F. Le couple $(E, \bar{f})$ est appelé une *table*.
- λ désigne le k-tuple vide et $\perp$ est un symbole spécial qui n'appartient pas à F.
- Pour tout entier $i \in [k]$ et tout élément $a \in E$, on désigne par E_a^i l'ensemble $\{x \in E : f_i(x) \neq f_i(a)\}$.
- Pour tout k-tuple $\bar{x} = (x_1, \ldots, x_k)$ et tout $i \in [k]$, l'expression x_{-i}^{-} désigne le $(k-1)$-tuple $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$.

Couvertures

Définition 77 (couverture) *Une couverture $\bar{c}$ d'une table $(E, \bar{f})$ est un tuple $(c_1, \ldots, c_k) \in (F \cup \{\perp\})^k$ tel que, pour tout élément $y \in E$, il existe un indice $i \in [k]$ tel que $c_i = f_i(y)$. On désigne par covers$(E, \bar{f})$ l'ensemble des couvertures de la table $(E, \bar{f})$.*

Définition 78 (couverture plus générale, couverture minimale) *Une couverture $\bar{c'} = (c'_1, \ldots c'_k)$ est dite plus générale qu'une couverture $\bar{c} = (c_1, \ldots c_k)$, ce qu'on note $\bar{c'} \leq \bar{c}$, si, pour tout $i \in [k]$, soit $c'_i = c_i$, soit $c'_i = \perp$.*

Une couverture $\bar{c}$ d'une table $(E, \bar{f})$ est dite minimale s'il n'existe pas de couverture $\bar{c'}$ de la table $(E, \bar{f})$ strictement plus générale que $\bar{c}$.

Définition 79 (ensemble complet de couvertures, ensemble minimal de couvertures) *Un ensemble complet de couvertures d'une table $(E, \bar{f})$ est un ensemble $\mathcal{C}$ de couvertures de $(E, \bar{f})$ tel que, pour chaque couverture $\bar{c}$ de $(E, \bar{f})$, il existe une couverture $\bar{c'} \in \mathcal{C}$ plus générale (au sens large) que $\bar{c}$.*

Si, de plus, chaque élément de $\mathcal{C}$ est une couverture minimale de $(E, \bar{f})$ alors $\mathcal{C}$ est appelé ensemble minimal de couvertures de la table $(E, \bar{f})$.

Lemme 39 *Il existe un et un seul ensemble minimal de couvertures d'une table $(E, \bar{f})$ qu'on appelle mincovers$(E, \bar{f})$. C'est l'ensemble des couvertures minimales de $(E, \bar{f})$.*

Preuve. Pour l'existence, il suffit de noter que pour chaque $\bar{c} \in$ covers$(E, \bar{f})$, il existe $\bar{c'} \in$ mincovers$(E, \bar{f})$ qui est plus général que $\bar{c}$. Pour l'unicité, considérons $\mathcal{C}$ et $\mathcal{C}'$ deux ensembles minimaux de couvertures de $(E, \bar{f})$. Par complétude de $\mathcal{C}'$, il existe, pour chaque $\bar{c} \in \mathcal{C}$, un $\bar{c'} \in \mathcal{C}'$ tel que $\bar{c'} \leq \bar{c}$. Par minimalité de $\bar{c}$, cela implique que $\bar{c} = \bar{c'}$. Par conséquent, on a prouvé

que $\mathcal{C} \subseteq \mathcal{C}'$. Par symétrie, on obtient $\mathcal{C}' \subseteq \mathcal{C}$ et donc $\mathcal{C} = \mathcal{C}'$. Il est aussi facile de vérifier que mincovers$(E, \bar{f})$ est l'ensemble des couvertures minimales de $(E, \bar{f})$. □

Exemple 40 *On considère la table $(E, \bar{f})$ ci-dessous où $\bar{f} = (f_1, f_2, f_3)$, $f_i : E \to F$, $E = \{a, b, c, d, e\}$ et $F = \{1, 2, 3, 4, 5\}$.*

	f_1	f_2	f_3
a	1	2	4
b	2	5	1
c	3	2	4
d	3	5	3
e	5	2	4

On constate facilement que les triplets $(1, 5, 4), (2, 2, 3), (2, 5, 4), (3, 2, 1)$ et $(\perp, 5, 4)$ sont des couvertures de la table $(E, \bar{f})$ dont ils forment un ensemble complet de couvertures et que l'on a mincovers$(E, \bar{f}) = \{(\perp, 5, 4), (2, 2, 3), (3, 2, 1)\}$.

Représentants

La notion principale utilisée pour l'élimination de quantificateurs est celle de représentant.

Définition 80 (ensemble représentant) *Un représentant ou ensemble représentant d'une table $(E, \bar{f})$ est un sous-ensemble $E' \subseteq E$ tel que covers$(E, \bar{f}) =$ covers$(E', \bar{f})$* [3].

Dans l'exemple donné ci-dessus, $E' = \{a, b, c, d\}$ est un représentant de la table $(E, \bar{f})$.

Remarque 16 *La notion de représentant est à comparer avec la notion de "full kernel" d'un problème dans le cadre de la complexité paramétrée.*

Caractérisation des couvertures et des représentants

Le but de cette section est de montrer que pour une table $(E, \bar{f})$ donnée, $\bar{f} = (f_1, \ldots, f_k)$, il existe un représentant et un ensemble complet de couvertures de $(E, \bar{f})$ dont les cardinalités respectives ne dépendent pas de la cardinalité de l'ensemble E mais seulement de k.

Le lemme suivant donne une définition inductive de l'ensemble des couvertures d'une table.

Lemme 41 *Les trois propriétés suivantes sont vérifiées.*

1. *covers$(\emptyset, \bar{f}) = F^k$;*
2. *Si $E \neq \emptyset$ alors covers$(E, \lambda) = \emptyset$;*

[3] par abus de langage, on note encore $\bar{f} = (f_1, \ldots, f_k)$ la restriction de $\bar{f}$ à la partie $E' \subseteq E$.

3. *Supposons que* $E \neq \emptyset$, $a \in E$ *et* $\bar{f} \neq \lambda$. *Alors l'équivalence suivante est vraie, pour tout tuple* $\bar{c}$: $\bar{c} \in covers(E, \bar{f})$ *si et seulement si il existe un indice* $i \in [k]$ *tel que* $c_i = f_i(a)$ *et* $\bar{c_{-i}} \in covers(E_i^a, \bar{f_{-i}})$.

Preuve. Evidente et laissée au lecteur. □

Le lemme suivant va nous permettre de construire, de façon inductive, pour chaque table, un "petit" ensemble de représentants de cette table.

Lemme 42 *Les trois propriétés suivantes sont vérifiées.*

1. $\emptyset$ *est un représentant de la table* $(\emptyset, \bar{f})$;
2. *Si* $E \neq \emptyset$ *et* $a \in E$ *alors* $\{a\}$ *est un représentant de la table* (E, λ) ;
3. *Supposons* $E \neq \emptyset$, $a \in E$, $\bar{f} \neq \lambda$ *et, pour tout* $i \in [k]$, R_i *est un représentant de la table* $(E_i^a, \bar{f_{-i}})$. *Alors,* $\{a\} \cup \bigcup_{i \in [k]} R_i$ *est un représentant de la table* $(E, \bar{f})$.

Preuve. (1) et (2) sont évidents. Nous allons maintenant prouver (3). Par hypothèse, $a \in E$ et, pour tout $i \in [k]$, $R_i \subseteq E_i^a$ et $\text{covers}(R_i, \bar{f_{-i}}) = \text{covers}(E_i^a, \bar{f_{-i}})$. Nous avons seulement à prouver l'inclusion

$$\text{covers}(\{a\} \cup \bigcup_{i \in [k]} R_i, \bar{f}) \subseteq \text{covers}(E, \bar{f})$$

puisque l'inclusion inverse est évidente. Soit $\bar{c} \in \text{covers}(\{a\} \cup \bigcup_{i \leq k} R_i, \bar{f})$. En particulier, il existe un entier $j \in [k]$ tel que $f_j(a) = c_j$. Soit b un élément de E. Il y a deux cas possibles :

- soit $f_j(b) = f_j(a) = c_j$;
- soit $f_j(b) \neq f_j(a) = c_j$. Dans ce cas, $b \in E_j^a$. Par hypothèse, $\bar{c} \in \text{covers}(R_j, \bar{f})$ et $R_j \subseteq E_j^a$. Par conséquent, pour tout $x \in R_j$, $f_j(x) \neq f_j(a) = c_j$. On obtient donc $\bar{c} \in \text{covers}(R_j, \bar{f_{-j}})$. Donc $\bar{c} \in \text{covers}(E_j^a, \bar{f_{-j}})$ par hypothèse. Comme $b \in E_j^a$, il existe $i \neq j$ tel que $f_i(b) = c_i$.

Finalement, on a prouvé dans les deux cas que, pour chaque $b \in E$, il existe $i \in [k]$ tel que $f_i(b) = c_i$. Cela signifie que $\bar{c} \in \text{covers}(E, \bar{f})$.

□

La proposition suivante exprime le fait que la cardinalité d'un ensemble minimal de couvertures de toute table $(E, \bar{f})$ est bornée par un nombre indépendant de $card(E)$ et que toute table $(E, \bar{f})$ possède un ensemble représentant de cardinalité bornée de la même façon.

Proposition 43 1. *Pour chaque table* $(E, \bar{f})$, $\bar{f} : E \to F^k$, *on a* $card(mincovers(E, \bar{f})) \leq k!$;

2. *Il existe une fonction* $h(k) = O(k!)$ *telle que chaque table* $(E, \bar{f})$, $\bar{f} : E \to F^k$, *possède un ensemble représentant* E' *de cardinalité bornée par* $h(k)$ *et qu'on appellera un petit (ensemble) représentant de la table* $(E, \bar{f})$.

Preuve. Il est assez facile de vérifier que le nombre de couvertures minimales est borné par la fonction $g(k)$ définie par la récurrence $g(0) = 1$ et $g(i) = g(i-1) \times i$, pour $i \in [k]$, donc $g(k) = k!$. De manière similaire, à partir du lemme 42, on obtient un représentant E' d'une table $(E, \bar{f})$ de cardinalité bornée par la fonction $h(k)$ définie par la récurrence $h(0) = 1$ et $h(i) = i \times h(i-1)+1$, pour $i \in [k]$. D'où on déduit $h(k) = O(k!)$. □

On s'intéresse maintenant à la complexité du calcul d'un petit représentant. On définit donc le problème suivant.

SMALL-REPRESENTATIVE-SET	
Entrée:	deux ensembles finis E et F, un entier k et un k-tuple de fonctions unaires $\bar{f} = (f_1, \dots f_k)$ de E dans F
Paramètre:	l'entier k
Sortie:	un petit représentant E' de la table $(E, \bar{f})$

Lemme 44 *Le problème* SMALL-REPRESENTATIVE-SET *est calculable en temps* $O(k! \cdot card(E))$.

L'algorithme RepSet donné ci-dessous pour ce problème se déduit du lemme 42.

Algorithm 5 RepSet$(E, F, k, \bar{f})$

1: **si** $E = \emptyset$ **alors**
2: **renvoyer** $\emptyset$
3: **sinon**
4: choisir a dans E
5: **renvoyer** $\{a\} \cup \bigcup_{i \in [k]} \text{RepSet}(E_a^i, F, k-1, \bar{f_{-i}})$
6: **fin si**

On voit facilement que chaque ensemble E_i^a peut être calculé en temps $O(\text{card}(E))$. Par conséquent, la complexité globale de l'algorithme RepSet est $O(k! \cdot \text{card}(E))$.

2.4.3 Une première méthode d'élimination des quantificateurs

L'arbre à inégalités d'une formule **F-ACQ**$^{\neq}$ va jouer un rôle analogue à celui de l'arbre à égalités d'une formule **F-ACQ**.

Définition 81 (arbre à inégalités) *L'arbre à inégalités d'une formule* ***F-ACQ***$^{\neq}$ *est son arbre à égalités* T *complété par des liens d'inégalité. Plus précisément, si* φ *contient* k *inégalités de la forme* $f(x) \neq g(y)$ *alors* k *liens d'inégalité* $\{x, y\}$ *sont ajoutés à* T.

Nous adaptons la notion d'arbre élagué d'un arbre à inégalités. Pour chaque feuille supprimée, on raccroche ses liens d'inégalités à son parent. Cette forme d'élagage reflète le traitement des inégalités dans l'élimination des quantificateurs telle qu'on l'effectuera ci-dessous.

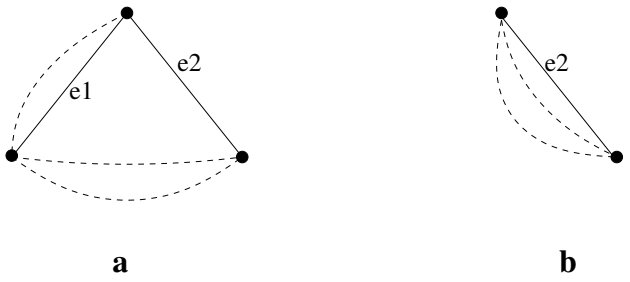


FIG. 2.2 – Arbre à inégalités T de racine x et son arbre élagué $T - \{y\}$

Définition 82 (arbre élagué) *Soit T un arbre à inégalités où y est une feuille et x est le parent de y. L'arbre y-élagué de T, noté $T - \{y\}$, est l'arbre à inégalités T modifié de la manière suivante :*

- *On supprime le noeud y, l'arête étiquetée $\{x, y\}$ et les liens d'inégalité entre x et y ;*
- *On remplace chaque lien d'inégalité $\{y, z\}$, pour $z \neq x$, par un lien d'inégalité $\{x, z\}$.*

De manière plus générale, un arbre élagué de T est un arbre à inégalités construit à partir de T en y-élaguant successivement des feuilles y.

Exemple 45 *Soit φ la formule **F-ACQ**$^{\neq}$ suivante :*

$$\varphi(x, y, z) \equiv f(x) = g(y) \wedge f'(x) = h(z) \wedge h(x) \neq f'(y) \wedge f(y) \neq g(z) \wedge g(y) \neq h(z)$$

L'arbre à inégalités T de φ, de racine x est représenté sur la figure 2.2.a.

Les étiquettes sont respectivement l'ensemble d'inégalités $e_1 = \{f(x) = g(y)\}$ et $e_2 = \{f'(x) = h(z)\}$. L'arbre y-élagué $T - \{y\}$ est représenté sur la figure 2.2.b. Les deux liens d'inégalité $\{y, z\}$ sont remplacés par deux liens d'inégalité $\{x, z\}$ dans $T - \{y\}$.

Définition 83 *Dans un arbre à inégalités T, le degré d'inégalité d'un noeud x, noté $deg^{\neq}_T(x)$, est le nombre de liens d'inégalité adjacents à x. Le degré d'inégalité d'un sous-arbre T' de T, noté $deg^{\neq}_T(T')$, est la somme des degrés d'inégalité de ses noeuds :*

$$deg^{\neq}_T(T') = \sum_{x \in V(T')} deg^{\neq}_T(x)$$

Lemme 46 *Soit k le nombre de liens d'inégalité de l'arbre à inégalités T.*

1. *Pour chaque sous-arbre T' de T, on a $deg^{\neq}_T(T') \leq 2k$;*
2. *Pour chaque arbre élagué T' de T et chaque noeud x de T', on a $deg^{\neq}_{T'}(x) \leq k$.*

Preuve.

1. C'est trivial puisque chaque lien d'inégalité est adjacent à exactement deux noeuds.

2. Cela résulte du fait que l'arbre élagué $T - \{y\}$ a au plus le même nombre de liens d'inégalité que T.

□

Nous énonçons maintenant notre premier résultat d'élimination des quantificateurs pour les formules **F-ACQ**$^{\neq}$.

Lemme 47 *Soit $\varphi(\bar{x})$ une σ-formule **F-ACQ**$^{\neq}$ dont l'arbre à inégalités est T et qui est de la forme $\varphi(\bar{x}) = \exists z \psi(\bar{x}, z)$ où ψ est sans quantificateur et z est une feuille de T. Soit k (resp. m) le nombre d'inégalités (resp. égalités) de $\varphi(\bar{x})$ qui font intervenir la variable z et soit $\mathcal{M}$ une σ-structure de domaine D. Alors on peut calculer en temps $O(k! \cdot card(D) + m \cdot card(D))$ une σ'-formule sans quantificateur $\varphi'(\bar{x})$ avec $\sigma \subseteq \sigma'$ et une σ'-structure $\mathcal{M}'$ qui est une expansion de $\mathcal{M}$ telles que*

- *$\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$;*
- *φ' est une disjonction de $h(k) = O(k!)$ formules ψ_i de **F-ACQ**$^{\neq}$ et d'arbre à inégalités $T - \{z\}$.*

Preuve. La formule $\varphi(\bar{x})$ peut être mise sous la forme suivante :

$$\varphi(\bar{x}) \equiv \psi_0(\bar{x}) \wedge \exists z (P(z) \wedge \bigwedge_{i \in [m]} g_i(z) = g'_i(y) \wedge \bigwedge_{i \in [k]} f_i(z) \neq f'_i(\bar{x}))$$

où $\psi_0(\bar{x})$ est une formule **F-ACQ**$^{\neq}$ sans quantificateur, y est le parent de z dans l'arbre T et $P(z)$ est un atome unaire sur z et l'expression $f'_i(\bar{x})$ désigne un terme de la forme $f'_i(v_i)$ pour une variable $v_i \in \bar{x}$. Soient $\bar{g}(z) = (g_1(z), \dots g_m(z))$, $\bar{g'}(y) = (g'_1(y), \dots g'_m(y))$. On partitionne, en temps $O(m \cdot \text{card}(D))$, l'ensemble $P = P(\mathcal{M})$ selon les valeurs de $\bar{g}$. Plus précisément, pour chaque $\bar{\alpha} \in \bar{g}(D)$, on définit $P_{\bar{\alpha}} = P \cap \bar{g}^{-1}(\bar{\alpha})$ et on calcule un petit représentant de la table $(P_{\bar{\alpha}}, \bar{f})$, représentant noté $P'_{\bar{\alpha}}$ (de cardinalité au plus $h(k) = O(k!)$). Cela peut être fait globalement en temps $\sum_{\bar{\alpha} \in \bar{g}(D)} O(k! \cdot \text{card}(P_{\bar{\alpha}})) = O(k! \cdot \text{card}(P)) = O(k! \cdot \text{card}(D))$. Clairement, la formule $\varphi(\bar{x})$ peut être reformulée en

$$\psi_0(\bar{x}) \wedge \exists z \in P_{\bar{g'}(y)} \bigwedge_{i \in [k]} f_i(z) \neq f'_i(\bar{x})$$

Par définition d'un ensemble représentant, elle est également équivalente à la formule suivante sur $\mathcal{M}$.

$$\psi_0(\bar{x}) \wedge \exists z \in P'_{\bar{g'}(y)} \bigwedge_{i \in [k]} f_i(z) \neq f'_i(\bar{x})$$

Soit $\mathcal{M}'$ une σ'-expansion de $\mathcal{M}$ obtenue en introduisant les relations unaires E_j et les fonctions unaires v_j et $f_{i,j}$, pour $i \in [k]$ et $j \in [h(k)]$, qui sont définis de la manière suivante :

- $E_j(y) \Leftrightarrow \mathrm{card}(P'_{\bar{g}'(y)}) \geq j$;
- $v_j(y)$ est le j-ème élément du petit représentant $P'_{\bar{g}'(y)}$ si $1 \leq j \leq \mathrm{card}(P'_{\bar{g}'(y)})$ et n'est pas défini sinon ;
- $f_{i,j} = f_i \circ v_j$.

Finalement, on obtient $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ pour la σ'-formule $\varphi' \equiv \bigvee_{j\in[h(k)]} \psi_i(\bar{x})$ où

$$\psi_j(\bar{x}) \equiv \psi_0(\bar{x}) \wedge E_j(y) \wedge \bigwedge_{i\in[k]} f_{i,j}(y) \neq f'_i(\bar{x})$$

On notera que chaque ψ_j est une formule **F-ACQ**$^{\neq}$ qui a pour arbre à inégalités $T - \{z\}$ comme demandé.

On voit facilement que l'expansion $\mathcal{M}'$ de $\mathcal{M}$ par les relations et fonctions unaires E_j, v_j et $f_{i,j}$ peut être calculée en temps $O(k!\cdot\mathrm{card}(D))$. Finalement, la complexité globale de la transformation $(\mathcal{M}, \varphi) \mapsto (\mathcal{M}', \varphi')$ est $O(m \cdot \mathrm{card}(D) + k! \cdot \mathrm{card}(D))$ comme demandé. □

En utilisant le résultat précédent, on peut déduire la proposition suivante.

Proposition 48 *Soit $\varphi(x)$ une σ-formule de **F-ACQ**$^{\neq}$ avec une seule variable libre x. Soit $\mathcal{M}$ une σ-structure. On peut calculer $\varphi(\mathcal{M})$ en temps $f(|\varphi|) \cdot |\mathcal{M}|$ pour une certaine fonction calculable f.*

2.4.4 Une borne de complexité plus précise

Dans cette section, nous cherchons à améliorer les bornes de complexité vis-à-vis des paramètres liés à la formule : par exemple, son nombre d'inégalités. Pour cela, on adopte une stratégie modifiée : notre élimination d'un quantificateur, au lieu de dupliquer, par disjonction, la formule, duplique, d'une certaine manière, le domaine de la structure.

Le lemme suivant est une variante du lemme 47. Il est commode de supposer que chaque variable x de la formule φ possède son domaine propre D_x. Cette convention autorisera à dupliquer le domaine D_x d'une variable x sans dupliquer les domaines des autres variables. La définition d'une σ-structure $\mathcal{M}$ est modifiée en conséquence. Plus précisément, on suppose, sans perte de généralité, que, pour chaque σ-formule φ de **F-ACQ**$^{\neq}$ et chaque σ-structure $\mathcal{M}$, les conditions suivantes sont vérifiées :

1. chaque symbole de fonction unaire f (resp. prédicat unaire U) apparaît une seule fois dans φ ;
2. si $f(x)$ (resp. $U(x)$) est l'unique terme (resp. atome) qui fait intervenir f (resp. U) dans φ alors f (resp. U) est appelé *symbole de x-fonction* (resp. *x-prédicat*) ou plus simplement *x-symbole*, et l'interprétation du symbole de fonction f (resp. du prédicat U) dans la structure $\mathcal{M}$ est la fonction $f : D_x \to D$ (resp. sous-ensemble $U \subseteq D_x$) où D est le domaine de $\mathcal{M}$ qui est l'union $\bigcup_{x\in\mathrm{var}(\varphi)} D_x$ des domaines des variables de φ.

On note σ_x l'ensemble des symboles de x-fonctions et des x-prédicats de σ. Clairement, σ est l'union disjointe des ensembles σ_x, pour les variables $x \in \mathrm{var}(\varphi)$.

Lemme 49 *Soit $\varphi(\bar{x})$ une σ-formule **F-ACQ**$^{\neq}$ d'arbre à inégalités T et de la forme $\varphi(\bar{x}) \equiv \exists y \exists z \psi(\bar{x}, y, z)$, où ψ est une formule sans quantificateur, z est une feuille de T et y le parent de z dans l'arbre, $y, z \notin \bar{x}$. Soit k (resp. m) le nombre d'inégalités (resp. d'égalités) dans ψ qui font intervenir la variable z, et soit $\mathcal{M}$ une σ-structure. Alors on peut calculer une formule sans quantificateur $\psi'(\bar{x}, y)$ et une σ'-structure $\mathcal{M}'$ telles que*

- *$\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ où $\varphi'(\bar{x}) = \exists y \psi'(\bar{x}, y)$;*
- *$\psi'(\bar{x}, y)$ est une formule **F-ACQ**$^{\neq}$ avec pour arbre à inégalités $T - \{z\}$ et ainsi $|\psi'| \leq |\psi|$;*
- *$card(\sigma') = card(\sigma) + k$; plus précisément, $card(\sigma'_y) = card(\sigma_y) + k$;*
- *dans la transformation $\mathcal{M} \mapsto \mathcal{M}'$, les domaines des variables ne sont pas modifiés à l'exception du domaine de y dont la cardinalité est multipliée par $h(k) = O(k!)$;*
- *la transformation $(\varphi, \mathcal{M}) \mapsto (\varphi', \mathcal{M}')$ est calculable en temps*

$$O(m \cdot card(D_z) + k!(card(\sigma_y) \cdot card(D_y) + card(D_z)))$$

où D_y et D_z sont les domaines respectifs de y et z dans $\mathcal{M}$.

Preuve. On décrit les différences avec la preuve du lemme 47. La première partie de la preuve est similaire avec la formule

$$\psi(\bar{x}, y, z) \equiv \psi_0(\bar{x}, y) \wedge P(z) \wedge \bigwedge_{i \in [m]} g_i(z) = g'_i(y) \wedge \bigwedge_{i \in [k]} f_i(z) \neq f'_i(\bar{x}, y)$$

où $\psi_0(\bar{x}, y)$ est une formule sans quantificateur, $P(z)$ est un atome unaire et $f'_i(\bar{x}, y)$ désigne un terme de la forme $f'_i(v_i)$ pour une variable $v_i \in \{\bar{x}, y\}$. La formule $\exists z \psi(\bar{x}, y, z)$ est équivalente à la formule suivante sur la même structure $\mathcal{M}$:

$$\psi_0(\bar{x}, y) \wedge \exists z \in P'_{g'(y)} \bigwedge_{i \in |k]} f_i(z) \neq f'_i(\bar{x}, y)$$

Rappelons que $P'_{\bar{g'}(y)}$ est un petit représentant de la table $(P_{\bar{g'}(y)}, \bar{f})$ où $P_{\bar{g'}(y)} = P(\mathcal{M}) \cap \bar{g}^{-1}(\bar{g'}(y))$. Soit maintenant $\mathcal{M}'$ une σ'-structure, $\sigma \subseteq \sigma'$, définie de la manière suivante à partir de $\mathcal{M}$:

- Premièrement, on remplace le domaine D_y de la variable y par le domaine suivant :

$$D'_y = \{(b, j) \in D_y \times [h(k)] : j \leq \mathrm{card}(P'_{g'(b)})\}$$

  Clairement on a, $\mathrm{card}(D'_y) \leq h(k)\mathrm{card}(D_y) = O(k!\mathrm{card}(D_y))$.
- Ensuite, on construit la signature étendue $\sigma' = \sigma \cup \{f''_i : i \in [k]\}$, où les f''_i sont k nouveaux symboles de y-fonctions unaires, et on définit leurs interprétations de la manière suivante

sur le domaine D'_y : pour tout élément $(b,j) \in D'_y$, $f''_i((b,j)) = f_i(u)$ où u est le j-ème élément du petit représentant $P'_{g'(\bar{b})}$.

– Pour finir, pour chaque symbole de y-fonction f (resp. chaque y-prédicat U) de σ et chaque élément $(b,j) \in D'_y$, on fixe $f^{\mathcal{M}'}((b,j)) = f^{\mathcal{M}}(b)$ (resp. $U^{\mathcal{M}'}((b,j)) \equiv U^{\mathcal{M}}(b)$).

Soit ψ' la formule suivante :

$$\psi'(\bar{x},y) \equiv \psi_0(\bar{x},y) \wedge \bigwedge_{i\in[k]} f''_i(y) \neq f'_i(\bar{x},y)$$

On peut conclure de manière analogue que $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ pour $\varphi(\bar{x}) \equiv \exists z \exists y \psi(\bar{x},y,z)$ et $\varphi'(\bar{x}) \equiv \exists y \psi'(\bar{x},y)$. La complexité de la transformation $(\mathcal{M},\varphi) \mapsto (\mathcal{M}',\varphi')$ est $O(m \cdot \mathrm{card}(D) + k!(\mathrm{card}(D_y) + \mathrm{card}(D_z)))$. □

2.4.5 Applications de l'élimination des quantificateurs

Nous allons procéder à l'élimination des quantificateurs d'une formule en éliminant successivement les variables-feuilles de son arbre.

Définition 84 (ordre d'élimination) *Soit φ une formule **F-ACQ**$^{\neq}$ d'arbre T. Un ordre d'élimination (ou T-ordre) des noeuds de T (variables de φ) est une liste ordonnée (éventuellement vide) $x_1,\ldots,x_p$ des noeuds de T telle que x_p est une feuille de T et $x_1,\ldots,x_{p-1}$ est un ordre d'élimination des noeuds de $T - \{x_p\}$.*

Remarque 17 *Un ordre d'élimination de T est en fait un ordre topologique des noeuds de T.*

Dans le résultat qui va suivre, nous allons appliquer l'ensemble de nos méthodes d'élimination des quantificateurs à l'évaluation d'une requête. Nous allons pour cela nous ramener à une formule sans quantificateur en utilisant successivement la méthode du lemme 49 (duplication du domaine d'une variable) et la méthode du lemme 47 (duplication de la formule). Notons qu'il ne semble pas possible de prouver la borne à laquelle nous aboutissons en n'utilisant qu'une seule de ces deux méthodes. Le résultat concerné et qui généralise le lemme 37 est le suivant.

Proposition 50 *Soient $\varphi(x)$ une σ-formula **F-ACQ**$^{\neq}$ possédant une seule variable libre x et $\mathcal{M}$ une σ-structure. Soit k le nombre d'inégalités de φ. La requête $\varphi(\mathcal{M})$ est calculable en temps $O((2k)!^2|\varphi| \cdot |\mathcal{M}|)$ et donc en temps $2^{O(k\log k)} \cdot |\varphi| \cdot |\mathcal{M}|$.*

Preuve.

Soit T l'arbre à inégalités de φ. On considère que x est la racine de T. Sans perte de généralité, on supposera que φ est de la forme

$$\varphi(x) \equiv \exists \bar{y} \exists \bar{z} \psi(x,\bar{y},\bar{z})$$

où la formule ψ est sans quantificateur, $\bar{y} = y_1, \dots y_p$ est la liste des enfants de la racine x dans T, $\bar{z} = z_1, \dots, z_q$ et $x, y_1, \dots, y_p, z_1, \dots, z_q$ est un T-ordre des noeuds de T. L'algorithme qui calcule $\varphi(\mathcal{M})$ procède en trois phases :

- La phase 1 élimine successivement les variables $\bar{z}$ en utilisant le lemme 49 ;
- La phase 2 élimine successivement les variables $\bar{y}$ en utilisant le lemme 47 ;
- La phase 3 calcule $\varphi(\mathcal{M})$.

Phase 1

Les "feuilles" $z_q, \dots, z_2, z_1$ sont supprimées successivement en appliquant itérativement le lemme 49. La σ-structure initiale $\mathcal{M}$ est transformée en les σ'-structures $\mathcal{M}'$ successives en dupliquant, à chaque fois, le domaine de chaque variable parent de la variable éliminée. On note que le domaine D_x de la racine x qui, par construction, n'est parent d'aucun z_i n'est pas modifié.

Le fait suivant décrit comment, pour chaque variable v, distincte de x, le nombre de v-symboles de la signature et la cardinalité du domaine de v augmentent dans la transformation $(\sigma, \varphi, \mathcal{M}) \mapsto (\sigma', \varphi', \mathcal{M}')$. Pour une variable v, on note T_v le sous-arbre de T de racine v.

Fait 51 *Soit une variable $v \in \bar{y} \cup \bar{z}$, racine du sous-arbre T_v de T. A tout moment de la phase 1, les bornes supérieures suivantes sont respectées :*

- *$card(\sigma'_v) \leq card(\sigma_v) + deg^{\neq}_T(T_v)$;*
- *$card(D'_v) = O(card(D_v) \cdot (deg^{\neq}_T(T_v))!)$ où D_v (resp. D'_v) est le domaine de v dans $\mathcal{M}$ (resp. $\mathcal{M}'$) et σ_v (resp. σ'_v) est l'ensemble des v-symboles de σ (resp. σ').*

Preuve. On note $u_1, \dots, u_r \in \bar{z}$ la liste des enfants de v dans T. Par le lemme 49, quand une variable u_i est éliminée, la cardinalité de la signature associée à son parent v est augmentée de k_i où k_i est le nombre de liens d'inégalité adjacents à u_i. Similairement, la cardinalité du domaine de v est multipliée par $h(k_i) = O(k_i!)$. Par le lemme 46, on a $k_i \leq d_i$ pour $d_i = \deg^{\neq}_T(T_{u_i})$. Par conséquent, on obtient :

$$\text{card}(\sigma'_v) = \text{card}(\sigma_v) + \sum_{i \in [r]} d_i$$

et

$$\text{card}(D'_v) = O(\text{card}(D_v) \cdot \prod_{i \in [r]} d_i!)$$

Notons les inégalités :

$$\prod_{i \in [r]} d_i! \leq (\sum_{i \in [r]} d_i)!$$

et

$$\sum_{i \in [r]} \deg^{\neq}_T(T_{u_i}) \leq \deg^{\neq}_T(T_v)$$

Ce qui prouve le fait. □

Nous analysons maintenant la complexité en temps (noté Time(u)) de l'élimination d'une variable u de parent v dans la phase 1. Par le lemme 49,

$$\text{Time}(u) = O(m_u \cdot \text{card}(D'_u) + k_u!(\text{card}(D'_u) + \text{card}(\sigma'_v) \cdot \text{card}(D'_v)))$$

où m_u est le nombre d'égalités de φ faisant intervenir u et v et k_u est le nombre d'inégalités faisant intervenir u dans φ'. Par le fait 51, on a $\text{card}(D'_u) = O(\text{card}(D_u) \cdot d_u!)$ et $\text{card}(D'_v) = O(\text{card}(D_v) \cdot d_v!)$ où $d_u = \deg^{\neq}_T(T_u)$ et $d_v = \deg^{\neq}_T(T_v)$. On a aussi $\text{card}(\sigma'_v) \leq \text{card}(\sigma_v) + d_v$.

Par le lemme 46, on a $d_u \leq 2k$ et $d_v \leq 2k$. Par conséquent, $\text{card}(\sigma'_v) \leq \text{card}(\sigma_v) + 2k$, $\text{card}(D'_u) = O((2k)! \cdot \text{card}(D))$ et $\text{card}(D'_v) = O((2k)! \cdot \text{card}(D))$. Donc on a $\text{Time}(u) = O((m_u + k!)(2k)! \cdot |\mathcal{M}|)$. Notons que $\sum_{u \in \bar{z}} m_u \leq m$, $\text{card}(\sigma) \cdot \text{card}(D) \leq |\mathcal{M}|$ et $|\bar{z}| \leq |\varphi|$.

Le temps total de la phase 1 est donc :

$$\begin{aligned} \sum_{u \in \bar{z}} \text{Time}(u) &= O((2k)!(m \cdot \text{card}(D) + |\varphi|(k! \cdot \text{card}(D) + k!|\mathcal{M}|))) \\ &= O(k! \cdot (2k)!|\varphi| \cdot |\mathcal{M}|) \end{aligned}$$

Phase 2

Rappelons qu'à la fin de la phase 1, la σ'-formule obtenue admet un arbre à inégalités $T - \{\bar{z}\}$ de profondeur 1, qui consiste en la racine x et ses enfants $y_1, \ldots, y_p$. La σ'-structure correspondante $\mathcal{M}'$ a des domaines $D'_x = D_x = D$ et, pour tout $i \in [p]$, D'_{y_i} avec $\text{card}(D'_{y_i}) = O(d_i!\text{card}(D_{y_i}))$ où $d_i = \deg^{\neq}_T(T_{y_i})$.

On supprime successivement chaque variable y_i en utilisant l'algorithme du lemme 47. Soit $k_i = \deg^{\neq}_{T-\{\bar{z}\}}(y_i)$. Clairement, $\sum_{i=1}^{p} k_i \leq 2k$, et, après élimination de $y_1, \ldots, y_{i-1}$, la formule φ' est transformée en une disjonction de $O(k_1! \cdots k_{i-1}!)$ formules de **F-ACQ**$^{\neq}$ d'arbre à inégalités $T - \{\bar{z}, y_1, \ldots, y_{i-1}\}$. Donc, le temps requis pour éliminer la variable y_i dans toutes les disjonctions est

$$\text{Time}(y_i) = O(k_1! \cdot \ldots \cdot k_{i-1}!(m_i \text{card}(D'_{y_i}) + k_i! \cdot (\text{card}(D'_{y_i}) + \text{card}(D_x))))$$

où m_i est le nombre d'égalités dans φ faisant intervenir les variables y_i et x. Notons les inégalités

$$\prod_{i=1}^{p} k_i! \leq (\sum_{i=1}^{p} k_i)! \leq (2k)!$$

et

$$\text{card}(D'_{y_i}) \leq (2k)! \cdot \text{card}(D)$$

Donc, par le lemme 47

$$\text{Time}(y_i) = O(((2k)!m_i + (2k)!^2)|\mathcal{M}|)$$

Notons la borne

$$\sum_{i\in[p]} m_i = O(|\varphi|)$$

Le temps total de l'élimination de $y_1, \ldots, y_p$ est donc

$$\sum_{i\in|p]} Time(y_i) = O((2k)!^2|\varphi| \cdot |\mathcal{M}|)$$

Phase 3

A la fin de la phase 2, on obtient une disjonction $\varphi''(x) \equiv \bigvee_j P_j(x)$ de $O(\prod_{i=1}^{p} k_i!) = O((2k)!)$ σ''-formules sans quantificateur de **F-ACQ**$^{\neq}$, avec une seule variable x, donc des atomes unaires $P_j(x)$, et une σ''-structure $\mathcal{M}''$ de domaine $D_x = D$, telles que $\varphi''(\mathcal{M}'') = \varphi(\mathcal{M})$. Clairement, chaque requête $P_j(\mathcal{M}'')$ peut être calculée en temps $O(\text{card}(D))$ et la requête $\varphi(\mathcal{M}) = \bigcup_i P_i(\mathcal{M}'')$ est calculée en temps $O((2k)!\text{card}(D))$.

Le temps total, dominé par la phase 2, pour calculer $\varphi(\mathcal{M})$ est $O((2k)!^2|\varphi|\cdot|\mathcal{M}|)$ Cela prouve la proposition.

□

2.5 Enumération

Notre premier résultat de complexité d'énumération pour les formules acycliques avec ou sans inégalités est le suivant.

Proposition 52 *Soit $\varphi(x_1, \ldots, x_p)$ une σ-formule fixée de **F-ACQ**$^{\neq}$ (ou **ACQ**$^{\neq}$). Alors le problème* Enum(φ) *peut être évalué, sur une structure $\mathcal{M}$, avec un délai $O_\varphi(|\mathcal{M}|)$. Plus précisément, le délai est $O(p(2k)!^2|\varphi| \cdot |\mathcal{M}|)$ où k est le nombre d'inégalités de φ.*

Preuve. Nous décrivons un algorithme et sa complexité pour les formules **F-ACQ**$^{\neq}$. Le même résultat pour les formules **ACQ**$^{\neq}$ est déduit en utilisant la réduction exacte mise en évidence dans le lemme 35 (traduction 2). Soit $\mathcal{M}$ une σ-structure fonctionnelle et $\varphi(x_1, \ldots, x_p)$ une σ-formule **F-ACQ**$^{\neq}$.

L'algorithme 6 énumère les éléments de $\varphi(\mathcal{M})$. Nous analysons maintenant la complexité de cet algorithme d'énumération. Clairement, le délai pour mettre en sortie chaque solution $(a_1, \ldots, a_p) \in \varphi(\mathcal{M})$ est $O(\sum_{i\in[p]} \text{Time}(a_1, \ldots, a_{i-1}))$ où $\text{Time}(a_1, \ldots, a_{i-1})$ désigne le temps de l'évaluation de la requête $\varphi_{a_1,\ldots a_{i-1}}(\mathcal{M})$ où

$$\varphi_{a_1,\ldots a_{i-1}}(x_i) \equiv \exists x_{i+1} \ldots \exists x_p \varphi(a_1, \ldots a_{i-1}, x_i, x_{i+1}, \ldots, x_p)$$

Algorithm 6 Enum$(\varphi(x_1, \ldots, x_p), \mathcal{M})$

si $p = 1$ **alors**
  calculer $\varphi(\mathcal{M})$
  pour $a \in \varphi(\mathcal{M})$ **faire**
    mettre en sortie a
  fin pour
sinon
  évaluer $\varphi_1(x_1) \equiv \exists x_2 \ldots \exists x_p \varphi(x_1, \ldots x_p)$ sur $\mathcal{M}$
  pour $a \in \varphi_1(\mathcal{M})$ **faire**
    soit $\varphi_a \equiv \varphi(a, x_2, \ldots, x_p)$
    pour $\bar{b} \in$ Enum$(\varphi_a(x_2, \ldots, x_p), \mathcal{M})$ **faire**
      mettre en sortie $(a, \bar{b})$
    fin pour
  fin pour
fin si

Par la proposition 50, Time$(a_1, \ldots, a_{i-1}) = O((2k)!^2|\varphi| \cdot |\mathcal{M}|)$ Par conséquent, le délai pour la sortie $(a_1, \ldots, a_p)$ est

$$O(p(2k)!^2|\varphi| \cdot |\mathcal{M}|)$$

Ce qui donne la borne désirée pour le délai. □

2.5.1 Outils combinatoires

Afin d'énumérer à délai constant les solutions d'une requête **ACQ**$^{\neq}$, il sera utile de calculer efficacement le problème dynamique et paramétré suivant.

LeastNonCover

Entrée statique :	Un ensemble ordonné $(E, <)$, un ensemble F, un entier k et un k-tuple de fonctions unaires $\bar{f} = (f_1, \ldots f_k)$ de E dans F
Entrée dynamique :	un vecteur $\bar{u} = (u_1, \ldots, u_k) \in F^k$ et un élément $x \in E$
Paramètre :	l'entier k
Sortie :	le plus petit entier $y \geq x$ dans E tel que $f_i(y) \neq u_i$, pour tout $i \in [k]$, si un tel élément y existe, et $\perp$ sinon

Pour résoudre le problème LeastNonCover dans Constant-Time$_{lin}$, on précalculera un ensemble d'arbres de choix.

Définition 85 *Soit $(E, <), F, k, \bar{f}$ une entrée statique de* LeastNonCover *et soit x un élément de E. L'arbre de choix pour x est un arbre étiqueté, noté $T_x = (T, (label_\alpha(x))_{\alpha \in V(T)})$. Son*

ensemble de noeuds est

$$V(T) = \{(a_1, \ldots, a_l) \in [k]^l : l \in [0,k] \text{ et } a_i \neq a_j \text{ pour tous } i,j \text{ tels que } 1 \leq i < j \leq l\}.$$

La racine de T est le noeud ϵ (liste vide). Tout noeud $(a_1, \ldots, a_{l-1}, a_l)$ est enfant du noeud $(a_1, \ldots, a_{l-1})$, autrement dit, les ancêtres d'un noeud sont tous ses préfixes.
La fonction d'étiquetage $\alpha \mapsto label_\alpha(x)$ associe à chaque noeud $\alpha \in V(T)$ un élément de $E \cup \{\perp\}$ et est définie inductivement dans l'ordre préfixe de T.

- *étiquette de la racine : $label_\epsilon(x) = x$;*
- *étiquette d'un noeud $\alpha = (a_1, \ldots, a_l) \in V(T)$ avec $l \geq 1$: on distingue deux cas selon l'étiquette du parent de α :*
  - *soit $label_{a_1,\ldots,a_{l-1}}(x) = \perp$: on prend encore $label_{a_1,\ldots,a_l}(x) = \perp$;*
  - *soit $label_{a_1,\ldots,a_{l-1}}(x) \neq \perp$: l'étiquette du noeud $\alpha = (a_1, \ldots, a_l)$ est calculé à partir des étiquettes de ses ancêtres que l'on note $y_j = label_{a_1,\ldots,a_j}(x)$ pour $j \in [0, l-1]$; on prend*

$$label_{a_1,\ldots,a_l}(x) = \min\{y \in E : y \geq x \wedge \forall j \in [l]\ f_{a_j}(y) \neq f_{a_j}(y_{j-1})\}$$

si cet ensemble n'est pas vide, et sinon, on prend $label_{a_1,\ldots,a_l}(x) = \perp$.

L'objet essentiel pour calculer le problème LeastNonCover et précalculer les arbres de choix T_x, pour $x \in E$, est la fonction FIND.

Définition 86 (fonction FIND) *Supposons fixée une entrée statique du problème* LeastNonCover *: $(E, <), F, k, \bar{f}$. Soient un élément $x \in E$, une partie $S \subseteq [k]$ et une fonction $u : S \to F$. On définit* FIND(x, S, u) *comme le plus petit élément $y \geq x$ appartenant à E et tel que $f_i(y) \neq u(i)$, pour tout $i \in S$, si un tel élément y existe, et comme $\perp$ sinon.*

Remarque 18 *De façon évidente, la fonction* FIND *généralise le problème* LeastNonCover *: le résultat de* LeastNonCover *est* FIND$(x, [k], u)$ *où $u(i) = u_i$ pour tout $i \in [k]$.*

On calcule aisément FIND(x, S, u) en utilisant la procédure Find ci-contre qui parcourt une branche de l'arbre de choix T_x précalculé.

Algorithm 7 Find(x, S, u)

1: $\alpha \leftarrow \epsilon$
2: $y \leftarrow x$
3: $S' \leftarrow S$
4: **tant que** $y \neq \perp$ **faire**
5: **si** pour tout $i \in S'$ on a $f_i(y) \neq u(i)$ **alors**
6: **renvoyer** y
7: **sinon**
8: choisir $i \in S'$ avec $f_i(y) = u(i)$
9: $\alpha \leftarrow (\alpha, i)$
10: $y \leftarrow \text{label}_\alpha(x)$
11: $S' \leftarrow S' - \{i\}$
12: **fin si**
13: **fin tant que**
14: renvoyer $\perp$

Complexité de la procédure Find : Le nombre d'itérations de la boucle "tant que" est au plus $k+1$ et le temps d'un pas de boucle est $O(k)$. Le temps de calcul de Find est donc $O(k^2)$.

Correction de la procédure Find : Soit $y_0, y_1, \ldots, y_l$ la liste des valeurs prises successivement par la variable y au cours de l'exécution de la procédure Find(x, S, u). De l'examen de la procédure, on déduit les points suivants :

1. Clairement, on a $y_j = \text{label}_{a_1,\ldots,a_j}(x)$, pour tout $j \in [0, l]$, pour une certaine liste de l valeurs distinctes $(a_1, \ldots, a_l) \in S^l$.
2. On a $f_{a_j}(y_{j-1}) = u(a_j)$, pour tout $j \in [l]$: car, dans le cas contraire, on n'aurait pas calculé la nouvelle valeur y_j de y.

Le résultat de la procédure Find(x, S, u) est, par construction, $y_l = \text{label}_{a_1,\ldots,a_l}(x)$. Prouvons qu'il est correct. On a deux cas.

Premier cas : $y_l = \text{label}_{a_1,\ldots,a_l}(x) = \perp$. Par définition de l'étiquette $\text{label}_{a_1,\ldots,a_l}(x) = \perp$ du noeud $(a_1, \ldots, a_l)$ dans l'arbre T_x, l'ensemble $\{y \in E : y \geq x \wedge \forall j \in [l]\ f_{a_j}(y) \neq f_{a_j}(y_{j-1})\}$ est vide. A cause du point 2, cela signifie que l'ensemble $\{y \in E : y \geq x \wedge \forall j \in [l]\ f_{a_j}(y) \neq u(a_j)\}$ est vide. A cause de l'inclusion $\{a_1, \ldots, a_l\} \subseteq S$, l'ensemble $\{y \in E : y \geq x \wedge \forall i \in S\ f_i(y) \neq u(i)\}$ est aussi vide.

Second cas : $y_l = \text{label}_{a_1,\ldots,a_l}(x) \neq \perp$. Par définition de l'étiquette $\text{label}_{a_1,\ldots,a_l}(x)$ dans T_x, on a donc

$$y_l = \min\{y \in E : y \geq x \wedge \forall j \in [l]\ f_{a_j}(y) \neq f_{a_j}(y_{j-1})\}.$$

A cause du point 2, ceci se réécrit :

$$y_l = \min\{y \in E : y \geq x \wedge \forall j \in [l]\ f_{a_j}(y) \neq u(a_j)\}.$$

Par définition de la procédure Find, puisque son résultat est $y_l \neq \bot$, on a $f_i(y_l) \neq u(i)$, pour tout $i \in S'$. Par construction de l'ensemble S', on a $S = S' \cup \{a_1, \ldots, a_l\}$.

On en tire donc :

$$y_l = \min\{y \in E : y \geq x \wedge \forall i \in S\ f_i(y) \neq u(i)\}.$$

En conséquence, dans chacun des deux cas, le résultat y_l de la procédure $\text{Find}(x, S, u)$ est la valeur spécifiée $\text{FIND}(x, S, u)$. La procédure est donc correcte.

Il reste à décrire le précalcul des arbres T_x, pour tout $x \in E$. On peut supposer que l'ensemble ordonné $(E, <)$ est l'intervalle $[n]$ avec son ordre naturel $<$. On utilise les propriétés suivantes.

Lemme 53 *Soit un noeud $(a_1, \ldots, a_l) \in [k]^l$ de T_x qui n'est pas racine, c'est-à-dire avec $l \geq 1$. Examinons ce que vaut son étiquette.*

1. *Si $x = n$ alors $label_{a_1,\ldots,a_l}(x) = \bot$;*
2. *si $x < n$ alors on a $label_{a_1,\ldots,a_l}(x) = \text{FIND}(x, S, u) = \text{FIND}(x+1, S, u)$ avec l'ensemble $S = \{a_1, \ldots, a_l\}$ et la fonction $u : S \to F$ définie par $u(a_j) = f_{a_j}(label_{a_1,\ldots,a_{j-1}}(x))$.*

Preuve. On note d'abord que si on a $\text{label}_{a_1,\ldots,a_l}(x) \neq \bot$ alors, puisqu'on a par définition $\text{label}_{a_1,\ldots,a_l}(x) \geq x$ et $f_{a_1}(\text{label}_{a_1,\ldots,a_l}(x)) \neq f_{a_1}(x)$, il s'en déduit $\text{label}_{a_1,\ldots,a_l}(x) \geq x + 1$.

Prouvons le (1) : Si on a $x = n$ et $\text{label}_{a_1,\ldots,a_l}(x) \neq \bot$ alors on en déduit $\text{label}_{a_1,\ldots,a_l}(x) \geq n+1$: c'est absurde.

Prouvons le (2) : En examinant la définition de l'étiquette $\text{label}_{a_1,\ldots,a_l}(x)$ d'une part, et les définitions de S, u et FIND d'autre part, on vérifie immédiatement l'égalité $\text{label}_{a_1,\ldots,a_l}(x) = \text{FIND}(x, S, u)$. On a deux cas :

- soit on a $\text{FIND}(x, S, u) = \bot$ et on en déduit immédiatement $\text{FIND}(x+1, S, u) = \bot$;
- soit on a $\text{FIND}(x, S, u) \neq \bot$ et on en tire $\text{FIND}(x, S, u) = \text{label}_{a_1,\ldots,a_l}(x) \geq x + 1$. D'où on déduit $\text{label}_{a_1,\ldots,a_l}(x) = \text{FIND}(x+1, S, u)$.

L'assertion (2) est prouvée dans les deux cas. □

Les points (1) et (2) du lemme précédent établissent la correction du précalcul ci-dessous : ce précalcul calcule les étiquettes des arbres de choix T_x dans l'ordre T_n, $T_{n-1} \ldots T_1$.

Algorithm 8 Phase de précalcul

pour x de n à 1 en décroissant **faire**
  $\text{label}_\epsilon(x) \leftarrow x$
  pour l de 1 à k **faire**
    pour chaque liste $\alpha = (a_1, \ldots, a_l) \in [k]^l$ formée de l valeurs distinctes **faire**
      si $x = n$ ou $\text{label}_{a_1,\ldots,a_{l-1}}(x) = \perp$ **alors**
        $\text{label}_\alpha[x] \leftarrow \perp$
      sinon
        soit $S = \{a_1, \ldots, a_l\}$
        soit la fonction $u : S \to F$ définie pour tout $j \in [l]$ par $u(a_j) = f_{a_j}(\text{label}_{a_1,\ldots,a_{j-1}}(x))$
        $\text{label}_\alpha(x) \leftarrow \text{Find}(x+1, S, u)$
      fin si
    fin pour
  fin pour
fin pour

Complexité du précalcul : L'instruction la plus coûteuse est l'appel de la procédure $\text{Find}(x+1, S, u)$ dont le temps est $O(k^2)$. Cette procédure est appelée (sur des arguments divers) un nombre de fois borné par $n \cdot \sum_{l \in [k]} l! = O(n \cdot k!)$. Le temps du précalcul est donc $O(k^2 k! \text{card}(E))$. □

Les éléments précédents, algorithmes et preuves, justifient le résultat suivant.

Proposition 54 *Le problème dynamique et paramétré* LeastNonCover *est* Constant-Time$_{lin}$. *Plus précisément, ce problème est calculé en temps $O(k^2)$ après un précalcul en temps $O(k^2 k! card(E))$.*

L'intérêt principal des objets et algorithmes qu'on vient d'introduire est qu'ils vont nous permettre de calculer efficacement le problème d'énumération paramétré suivant qui possède à la fois une entrée statique et une entrée dynamique.

EnumNonCover

Entrée statique :	Un ensemble ordonné $(E, <)$, un ensemble F, un entier k et un k-uplet de fonctions unaires $\bar{f} = (f_1, \ldots f_k)$ de E dans F
Entrée dynamique :	un vecteur $\bar{u} = (u_1, \ldots, u_k) \in F^k$
Paramètre :	l'entier k
Sortie :	l'ensemble $\{x \in E : \bigwedge_{i \in [k]} f_i(x) \neq u_i\}$ énuméré dans l'ordre croissant de $(E, <)$

La proposition suivante énonce que ce problème a la complexité désirée.

Proposition 55 *Le problème d'énumération* EnumNonCover *est dans* Constant-Delay$_{lin}$. *Plus précisément, ce problème est énumérable à délai $O(k^2)$ après un précalcul en temps $O(k^2 k! card(E))$.*

Preuve. Le schéma d'énumération EnumNonCover du problème EnumNonCover itère la phase dynamique, notée LeastNonCover, du schéma dynamique du problème LeastNonCover et les phases statiques des deux schémas sont identiques. Nous ne donnons donc que la phase d'énumération d'EnumNonCover.

Algorithm 9 EnumNonCover$(E, F, k, \bar{f}, \bar{u})$

$x \leftarrow 0$
tant que $x \neq \perp$ **faire**
  $x \leftarrow$ LeastNonCover$(E, F, k, \bar{f}, \bar{u}, x+1)$
  mettre en sortie x
fin tant que

□

Remarque 19 *Le schéma d'énumération EnumNonCover est aussi un schéma dynamique dans le sens suivant : son précalcul est indépendant de la donnée $\bar{u} \in F^k$. Il ne dépend que de $(E, <)$, F, k et $\bar{f}$.*

2.5.2 Enumération des solutions d'une requête acyclique sans quantificateur

Dans ce qui va suivre, on supposera implicitement que les variables de la formule **F-ACQ**$^{\neq}$ sans quantificateur sont numérotées dans un certain T-ordre fixé.

Théorème 56 *Soit $\varphi(x_1, \ldots, x_p)$ une σ-formule de **F-ACQ**$^{\neq}$ d'arbre T, sans quantificateur. Alors on a* Enum$(\varphi, <_{lex}) \in$ Constant-Delay$_{lin}$ *pour l'ordre lexicographique $<_{lex}$.*

Preuve. Le théorème est prouvé par induction sur le nombre de variables de φ. Sans perte de généralité, on supposera que φ est de la forme $\varphi(\bar{x}, y) \equiv \varphi_0(\bar{x}) \wedge \Theta(\bar{x}, y)$ où

$$\Theta(\bar{x}, y) \equiv P(y) \wedge \bigwedge_{i \in [m]} f_i(y) = g_i(z) \wedge \bigwedge_{i \in [k]} h_i(y) \neq h'_i(\bar{x})$$

Dans cette formule, $P(y)$ est un atome unaire, la variable $z \in \bar{x}$ est un parent de y dans l'arbre T de φ et $h'_i(\bar{x})$ désigne un terme $h'_i(v_i)$ pour une certaine variable $v_i \in \bar{x}$. Trivialement, la formule φ est logiquement équivalente à la formule suivante $\varphi_1(\bar{x}, y) \equiv \psi(\bar{x}) \wedge \Theta(\bar{x}, y)$ où $\psi(\bar{x}) \equiv \varphi_0(\bar{x}) \wedge \exists y \Theta(\bar{x}, y)$. Par le lemme 47, on peut calculer en temps linéaire, pour chaque σ-structure $\mathcal{M}$, une nouvelle σ'-structure $\mathcal{M}'$, σ'-expansion of $\mathcal{M}$, $\sigma \subseteq \sigma'$, et une disjonction $\psi'(\bar{x}) \equiv \bigvee_{i \in [N]} \psi_i(\bar{x})$

de formules ψ_i sans quantificateur dans **F-ACQ**$^{\neq}$ (chacune d'arbre à inégalités $T - \{y\}$) telles que $\psi(\mathcal{M}) = \psi'(\mathcal{M}')$. Cela donne $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ pour la formule sans quantificateur suivante : $\varphi'(\bar{x}, y) \equiv \psi'(\bar{x}) \wedge \Theta(\bar{x}, y)$. φ' est équivalente à la disjonction $\bigvee_{i\in[N]} \varphi_i(\bar{x}, y)$ où φ_i est la formule

$$\varphi_i(\bar{x}, y) \equiv \psi_i(\bar{x}) \wedge P(y) \wedge \bar{f}(y) = \bar{g}(z) \wedge \bigwedge_{i\in[k]} h_i(y) \neq h'_i(\bar{x})$$

Par l'hypothèse d'induction, on a, pour chaque $i \in [N]$, $\textsc{Enum}(\psi_i, <_{lex}) \in \textsc{Constant-Delay}_{lin}$. Par le lemme 18, il est suffisant de prouver le même résultat de complexité pour le problème $\textsc{Enum}(\varphi_i, <_{lex}), i \in [N]$. Notons le fait essentiel que l'ordre linéaire $<_{lex}$ est le même pour chaque $i \in [N]$.

On décrit ci-dessous l'algorithme pour $\textsc{Enum}(\varphi_i, <_{lex})$.

Entrée : Une σ'-structure $\mathcal{M}'$ de domaine D ordonné par $<$

Phase de précalcul

- Effectuer la phase de précalcul de $\textsc{Enum}(\psi_i, <_{lex})$;
- Soit $P = \{y \in D : \mathcal{M}' \models P(y)\}$;
- En triant les éléments $y \in P$ en fonction des valeurs $\bar{f}(y)$, calculer la partition de l'ensemble P en ensembles non vides $P_{\bar{\alpha}} = \{y \in P : \bar{f}(y) = \bar{\alpha}\}$ ainsi que l'ensemble $A = \{\bar{\alpha} \in D^m : P_{\bar{\alpha}} \neq \emptyset\}$;
- Pour chaque $\bar{\alpha} \in A$, exécuter la phase de précalcul du schéma EnumNonCover sur l'entrée (statique) $(P_{\bar{\alpha}}, D, k, (h_i)_{i\in[k]})$.

Phase d'énumération

Pour chaque solution $\bar{a}$ énumérée par $\textsc{Enum}(\psi_i, <_{lex})$ faire

- Soit $\bar{\alpha} = \bar{g}(c)$ où c est la valeur de la variable z dans le tuple $\bar{a}$ (si z est la variable x_i, alors $c = a_i$) ;
- Soit $\bar{u} = (h'_i(\bar{a}))_{i\in[k]}$;
- Pour chaque élément b énuméré par $\text{EnumNonCover}(P_{\bar{\alpha}}, D, k, (h_i)_{i\in[k]}, \bar{u})$, renvoyer $(\bar{a}, b)$.

Correction de l'algorithme : Cet algorithme est correct par la structure de la formule φ_i et la spécification de l'algorithme EnumNonCover.

Complexité de l'algorithme : On vérifie facilement que chacun des quatre points de la phase de précalcul est exécuté en temps $O(|\mathcal{M}'|)$. Le lecteur peut également facilement constater que la phase d'énumération est à délai constant. La *propriété d'extension* suivante est essentielle pour cela : par construction des formules ψ, ψ' et ψ_i, chaque solution $\bar{a} \in \psi_i(\mathcal{M}')$ peut être étendue en (au moins) une solution $(\bar{a}, b) \in \varphi_i(\mathcal{M}')$. Donc, $\textsc{Enum}(\varphi_i, <_{lex}) \in \textsc{Constant-Delay}_{lin}$ comme demandé. Cela complète la preuve inductive du théorème. □

2.5.3 Formules connexe-acycliques : énumération et comptage

Nous introduisons maintenant une classe de formules acycliques particulières, pour lesquelles nous allons voir que les deux problèmes de l'énumération et du comptage sont de complexité optimale.

Définition 87 *Une formule de* ***F-ACQ*** *(resp.* ***F-ACQ***$^{\neq}$*) est dite connexe-acyclique si elle est acyclique et si l'ensemble de ses variables libres est une partie connexe de l'arbre de la formule. On désigne par* ***F-CCQ*** *(resp.* ***F-CCQ***$^{\neq}$*) la classe des formules connexe-acycliques de* ***F-ACQ*** *(resp.* ***F-ACQ***$^{\neq}$*).*

Nous nous intéressons d'abord au problème de l'énumération pour une formule connexe-acyclique sur une structure quelconque.

Théorème 57 *Pour toute formule φ de* ***F-CCQ***$^{\neq}$*, on a* $\textsc{Enum}(\varphi) \in \textsc{Constant-Delay}_{lin}$.

Preuve. Il suffit d'éliminer inductivement toutes les "feuilles" de φ qui sont des variables quantifiées en utilisant le lemme 47 jusqu'à aboutir à une formule φ' sans quantificateur. On applique ensuite le théorème 56. □

Nous nous intéressons maintenant au problème du calcul du nombre de solutions d'une requête **F-CCQ**, donc sans inégalité, sur une structure quelconque.

Théorème 58 *Pour toute formule φ de* ***F-CCQ****, le problème* $\textsc{Count}(\varphi)$ *est calculable en temps linéaire.*

Preuve. Considérons $\varphi(x_1, \ldots, x_k)$ une formule **F-CCQ** à k variables libres et $\mathcal{M}$ une structure de domaine D et de même signature que φ. Par le lemme 36, on peut supposer, sans perte de généralité, que φ est une formule de **F-ACQ** sans quantificateur. Comme pour l'énumération, nous allons procéder par élimination successive des "feuilles" de l'arbre de φ. Pour cela, nous considérons le problème plus général suivant : Etant donné un tuple (de pondération) $w : (w_v)_{v \in \text{free}(\varphi)}$ où chaque w_v est une fonction (de poids) de D dans $\mathbb{N}$, le *comptage pondéré* (notion déjà utilisée par Frick [Fri04]) par w de $\varphi(\mathcal{M})$ est défini de la manière suivante :

$$w(\varphi(\mathcal{M})) = \sum_{(a_1,\ldots,a_k) \in \varphi(\mathcal{M})} \prod_{i \in [k]} w_{x_i}(a_i)$$

Evidemment, dans le cas particulier où on pose $w_x(a) = 1$, pour toute variable x et tout élément $a \in D$, on obtient $\text{card}(\varphi(\mathcal{M})) = w(\varphi(\mathcal{M}))$ comme demandé.

Comme dans la preuve du lemme 36, on peut écrire notre formule φ (**F-ACQ** sans quantificateur) sous la forme :

$$\varphi(\bar{x}, z) = \psi(\bar{x}) \wedge P(z) \wedge \bar{f}(z) = \bar{g}(y)$$

où ψ est une formule **F-ACQ** sans quantificateur, $P(z)$ est un atome unaire, z est une feuille de l'arbre de φ, y le parent de z, $y \in \bar{x}$, et $\bar{f}$ et $\bar{g}$ sont des tuples de fonctions.

Considérons un nouveau tuple de pondération $w' = (w'_v)_{v \in \text{free}(\psi)}$ où chacune des fonctions de poids w'_v est égale à w_v, sauf la fonction de poids w'_y :

$$w'_y(a) = w_y(a) \cdot \sum_{b \in D : \bar{f}(b) = \bar{g}(a)} w_z(b) \quad \text{pour tout } a \in D$$

Intuitivement, $w'_y(a)$ est le nombre de solutions de la sous-formule de φ correspondant au sous-arbre T_y, de racine y, de l'arbre T (de φ) et dans lesquelles la valeur de y est a.

On se convaincra aisément que l'on a $w(\varphi(\mathcal{M})) = w'(\psi(\mathcal{M}))$ et que w' peut être calculé en temps $O_\varphi(|\mathcal{M}|)$. Ainsi, en itérant, on peut calculer, en temps linéaire, une formule φ' de la forme $Q(y)$, pour un prédicat unaire Q, et une fonction de poids w' telles que $w(\varphi(\mathcal{M})) = w'(\varphi'(\mathcal{M})) = \sum_{a \in Q(\mathcal{M})} w'_y(a)$. □

Remarque 20 *Comme on va le voir par la suite, le théorème précédent ne peut pas se généraliser aux formules connexe-acycliques avec inégalités.*

2.6 Une classe particulière de requêtes acycliques avec inégalités

2.6.1 Problèmes combinatoires paramétrés exprimés par des requêtes acycliques généralisées

Un autre intérêt de nos formules conjonctives acycliques généralisées (**ACQ**$^{\neq}$ ou **F-ACQ**$^{\neq}$) est qu'elles peuvent exprimer un certain nombre de problèmes naturels **NP**-complets dont la version paramétrée est **FPT**. En conséquence, notre procédure d'élimination des quantificateurs permet d'obtenir une méthode générale et uniforme pour déduire des bornes supérieures pour la complexité paramétrée de chacun de ces problèmes, à la fois dans sa version de décision et sa version d'énumération.

Un exemple typique est le problème k-PATH dont la version d'énumération est la suivante :

k-PATH	
Entrée:	un graphe G
Paramètre:	un entier k
Sortie:	l'ensemble des chemins simples de longueur k de G

Le problème k-PATH peut être exprimé par la formule **ACQ**$^{\neq}$

$$\psi_k(x_0, \ldots, x_k) \equiv \bigwedge_{i \in [0,k-1]} E(x_i, x_{i+1}) \wedge \bigwedge_{0 \leq i < j \leq k} x_i \neq x_j$$

Plus précisément, $\psi_k(G)$ est l'ensemble des k-chemins $(a_0, \ldots, a_k) \in V^{k+1}$ de G. Notons que les inégalités de ψ_k forment une "clique" (graphe complet) $\{x_0, \ldots, x_k\}$.

Il s'avère que cette propriété de clique est vérifiée pour plusieurs problèmes combinatoires que nous avons étudiés. Cela justifie la définition suivante.

Définition 88 *Une formule* $\boldsymbol{ACQ}^{all\neq}$ *(resp.* $\boldsymbol{F\text{-}ACQ}^{all\neq}$*) est une formule* $\boldsymbol{ACQ}^{\neq}$ *(resp.* $\boldsymbol{F\text{-}ACQ}^{\neq}$*) de la forme*

$$\exists x_1 \dots \exists x_p(\psi \wedge All^{\neq}(\tau_1, \dots, \tau_k))$$

où ψ *est une formule* $\boldsymbol{ACQ}$ *(resp.* $\boldsymbol{F\text{-}ACQ}$*) sans quantificateur et* $All^{\neq}(\tau_1, \dots, \tau_k)$ *est l'abréviation de la conjonction*

$$\bigwedge_{i,j\in[k],i<j} \tau_i \neq \tau_j$$

et chaque terme τ_i *est une variable (resp. un terme de la forme* $f(x)$*). L'ensemble de variables (resp. termes)* $\{\tau_1, \dots, \tau_k\}$ *est appelé la clique d'inégalités de* φ*.*

Par exemple, la formule ψ_k ci-dessus peut être réécrite sous la forme $\mathbf{ACQ}^{all\neq}$ suivante :

$$\psi_k(x_0, \dots, x_k) \equiv \bigwedge_{i\in[0,k-1]} E(x_i, x_{i+1}) \wedge \mathrm{All}^{\neq}(x_0, \dots, x_k)$$

Un autre exemple de problème est le suivant.

MultidimensionalMatching	
Entrée:	Des ensembles E, $F_1 \dots F_k$, k fonctions $f_i : E \to F_i$, pour $i \in [k]$, et un entier l
Paramètre:	les entiers k et l
Sortie:	les ensembles M de l éléments $M \in \mathcal{P}_l(E)$ tels que, pour tout $i \in [k]$ et toute paire d'éléments distincts $x, y \in M$, on a $f_i(x) \neq f_i(y)$

Sans perte de généralité, on peut supposer que les ensembles $F_1, \dots, F_k$ sont deux à deux disjoints. Dans ces conditions, le problème s'exprime par la formule $\mathbf{F\text{-}ACQ}^{all\neq}$ très simple :

$$\mathrm{All}^{\neq}_{(i,j)\in[k]\times[l]}(f_i(x_j))$$

L'intérêt de ce type de restriction est qu'il donnera des bornes de complexité plus précises : de manière informelle, le nombre k de termes de la formule $\mathrm{All}^{\neq}(\tau_1, \dots, \tau_k)$ remplacera le nombre $\Theta(k^2)$ d'inégalités $\tau_i \neq \tau_j$, $1 \leq i < j \leq k$. Ces bornes pourront être obtenues par une variante de notre procédure d'élimination des quantificateurs qui utilise une version non ordonnée de couverture : au lieu de couvrir un ensemble de tuples par un tuple, on couvre un ensemble d'ensembles par un ensemble.

Remarquons qu'en appliquant le lemme 56, on prouve que les problèmes MultidimensionalMatching et k-Path sont énunérables à délai constant après prétraitement linéaire si leurs paramètres respectifs sont fixés.

Dans les sous-sections suivantes, on adapte la procédure d'élimination des quantificateurs des formules **F-ACQ**$^{\neq}$ au cas particulier des formules **F-ACQ**$^{all\neq}$; on modifie d'abord les notions de couverture et de représentant, puis celle d'arbre à inégalités qu'on appellera alors arbre pondéré.

2.6.2 Couvertures et représentants

Dans cette sous-section, on considérera un entier fixé $k \geq 0$, deux ensembles finis E et F et une fonction $f : E \to \mathcal{P}_k(F)$. Le couple (E, f) est encore appelé une *table*. Pour chaque élément $x \in F$, la notation E_x désigne le sous-ensemble $\{y \in E : x \notin f(y)\}$.

Remarque 21 *L'entier k et l'ensemble F seront implicites par la suite.*

Définition 89 (l-couverture) *Une l-couverture S d'une table (E, f) est une partie de F telle que $card(S) \leq l$ et pour tout $x \in E$, on a $f(x) \cap S \neq \emptyset$.*

On désigne par l-COVERS(E, f) l'ensemble des l-couvertures de la table (E, f).

Définition 90 (l-couverture minimale) *Une l-couverture S d'une table (E, f) est dite minimale s'il n'existe pas de l-couverture S' strictement incluse dans S.*

Définition 91 (ensemble complet de couvertures, ensemble minimal de couvertures) *Un ensemble complet de l-couvertures d'une table (E, f) est un ensemble $\mathcal{C}$ de l-couvertures de E tel que, pour chaque l-couverture S de (E, f), il existe un ensemble $S' \in \mathcal{C}$ inclus dans S.*

Si, de plus, chaque élément de $\mathcal{C}$ est une couverture minimale de (E, f) alors $\mathcal{C}$ est appelé ensemble minimal de l-couvertures de (E, f).

Définition 92 (l-représentant) *Un l-représentant d'une table (E, f) est une partie $E' \subseteq E$ telle que l-COVERS$(E, f) = l$-COVERS(E', f).* [4]

Le lemme suivant construit récursivement, pour chaque table (E, f) un l représentant de cette table.

Lemme 59 *Les trois propriétés suivantes sont vérifiées.*

1. *Pour tout $l \geq 0$, $\emptyset$ est un l-représentant de la table $(\emptyset, f)$;*
2. *Si $E \neq \emptyset$ et $a \in E$ alors $\{a\}$ est un 0-représentant de la table (E, f) ;*
3. *Supposons que $E \neq \emptyset$, $a \in E$, $l > 0$ et, pour tout $x \in f(a)$, R_a est un $(l-1)$-représentant de la table (E_x, f). Alors l'ensemble $\{a\} \cup \bigcup_{x \in f(a)} R_x$ est un l-représentant de la table (E, f).*

[4] Par abus de notation, la restriction de f sur E' est également notée f

Preuve. Nous laissons au lecteur le soin de prouver ce lemme (preuve analogue à celle du lemme 41). □

Ce lemme permet d'obtenir un algorithme récursif RepSet qui calcule un "petit" l-représentant d'une table (E, f).

Lemme 60 *Il existe un algorithme qui étant, donnés une table (E, f) avec $f : E \to \mathcal{P}_k(F)$ et un entier l, calcule un l-représentant de la table (E, f) de cardinalité au plus $h(k, l) = O(k^l)$ en temps $O(k^l \cdot card(E))$.*

Preuve.

Il s'agit de l'algorithme RepSet donné ci-dessous et qui s'appuie sur le lemme 59. La complexité de cet algorithme se justifie de la même façon que pour l'algorithme, noté aussi RepSet, qui calcule un petit représentant d'une table $(E, \bar{f})$ pour $\bar{f} : E \to F^k$. Ici, la cardinalité d'un "petit" représentant de la table (E, f) est bornée par la fonction $h(k, l)$ définie par l'induction : $h(k, 0) = 1$ et, pour $l \geq 1$, $h(k, l) = 1 + k \cdot h(k, l-1)$. D'où $h(k, l) = O(k^l)$.

Algorithm 10 RepSet

1: **Entrée :** une table (E, f) et un entier l
2: **si** $E = \emptyset$ **alors**
3: **renvoyer** $\emptyset$
4: **sinon**
5: choisir a dans E
6: **si** $l = 0$ **alors**
7: **renvoyer** $\{a\}$
8: **sinon**
9: **renvoyer** $\{a\} \cup \bigcup_{x \in f(a)} \text{RepSet}(E_x, f, l-1)$
10: **fin si**
11: **fin si**

□

2.6.3 Arbre pondéré

Comme il est commode d'associer à une formule **F-ACQ**$^{\neq}$ son arbre à inégalités, il est pratique d'associer à une formule **F-ACQ**$^{all\neq}$ son arbre pondéré.

Définition 93 (arbre pondéré) *L'arbre pondéré d'une formule **F-ACQ**$^{all\neq}$*

$$\varphi \equiv \exists \bar{x}(\psi \wedge All^{\neq}(\tau_1, \ldots, \tau_k))$$

est son arbre à égalités T complété par une fonction de pondération w_T : $var(\varphi) \to \mathbb{N}$: $w_T(x)$ est le nombre de termes τ_i, $i \in [k]$, de la forme $f(x)$, présents dans la clique d'inégalités de φ.

La notion d'arbre élagué est modifiée en conséquence.

Définition 94 (arbre pondéré élagué) *Soit T un arbre pondéré où y est une feuille et x le parent de y. L'arbre y-élagué de T, noté $T - \{y\}$, est l'arbre pondéré T modifié de la manière suivante :*

- *On supprime le noeud y et l'arête étiquetée $\{x, y\}$;*
- *Le poids $w_T(x)$ de la variable x est remplacé par $w_{T-\{y\}}(x) = w_T(x) + w_T(y)$.*

De manière plus générale, un arbre élagué de T est un arbre pondéré T' construit à partir de T en supprimant itérativement des feuilles.

Il est commode d'introduire la notion de poids d'un sous-arbre.

Définition 95 (poids d'un sous-arbre) *Dans un arbre pondéré T, le poids d'un sous-arbre T' est la somme des poids de ses noeuds : $w_T(T') = \sum_{x \in V(T')} w_T(x)$.*

Lemme 61 *Pour tout arbre élagué T' d'un arbre pondéré T et chaque noeud x de T', on a $w_{T'}(x) \leq w_T(T_x)$ pour le sous-arbre T_x de racine x de T.*

Preuve. Notons que, pour chaque feuille $y \neq x$ de T_x, on a $w_{T-\{y\}}(T_x - \{y\}) = w_T(T_x)$. L'égalité $w_{T'}(T'_x) = w_T(T_x)$ s'en déduit inductivement, pour chaque arbre élagué T' qui contient x et le sous-arbre T'_x de racine x dans T'. D'où le résultat puisque $w_{T'}(x) \leq w_{T'}(T'_x)$. □

2.6.4 Elimination des quantificateurs

Les deux lemmes suivants sont des variantes des lemmes 47 et 49, et, les preuves étant analogues, nous en donnons juste des esquisses.

Lemme 62 *Soit $\varphi(\bar{x})$ une σ-formule $\boldsymbol{F\text{-}ACQ}^{all\neq}$ ayant pour arbre pondéré T et de la forme $\varphi(\bar{x}) = \exists z \psi(\bar{x}, z)$ où ψ est sans quantificateur et z est une feuille de T. Soient les entiers $k = w_T(z)$, $l = w_T(T)$ et m le nombre d'égalités de φ qui font intervenir la variable z. Soit $\mathcal{M}$ une σ-structure. Alors, on peut calculer en temps $O(k^l card(D) + m \cdot card(D))$ une σ'-formule sans quantificateur $\varphi'(\bar{x})$, avec $\sigma \subseteq \sigma'$, et une σ'-structure $\mathcal{M}'$ qui est une expansion de $\mathcal{M}$, telles que*

- *$\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$;*
- *φ' est une disjonction de $h(k, l) = O(k^l)$ formules ψ_i de $\boldsymbol{F\text{-}ACQ}^{all\neq}$ et d'arbre pondéré $T - \{z\}$, et donc $|\psi_i| \leq |\varphi|$.*

Preuve. La formule $\varphi(\bar{x})$ peut être mise sous la forme suivante :

$$\varphi(\bar{x}) \equiv \psi_0(\bar{x}) \wedge \exists z(P(z) \wedge \bigwedge_{i\in[m]} g_i(z) = g'_i(y) \wedge \mathrm{All}^{\neq}(f_1(z),\ldots,f_k(z),f'_1(\bar{x}),\ldots,f'_q(\bar{x})))$$

où $\psi_0(\bar{x})$ est une formule **F-ACQ** sans quantificateur d'arbre $T-\{z\}$, y est le parent de z dans l'arbre T, $y \in \bar{x}$, $P(z)$ est un atome unaire et $w_T(T) = l = k+q$ avec $q = w_T(T-\{z\})$. La preuve est similaire à celle du lemme 47 : on remplace les couvertures (resp. représentants) par des q-couvertures (resp. q-représentants) et on remplace chaque borne $h = h(k) = O(k!)$ par la borne $h = h(k,q) \leq h(k,l) = O(k^l)$ (rappelons l'inégalité $q+k=l$). Pour conclure la preuve, on obtient $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ pour la σ'-formule sans quantificateur $\varphi'(\bar{x}) \equiv \bigvee_{j\in[h]} \psi_j(\bar{x})$ où

$$\psi_j(\bar{x}) \equiv \psi_0(\bar{x}) \wedge E_j(y) \wedge \mathrm{All}^{\neq}(f_{1,j}(y),\ldots,f_{k,j}(y),f'_1(\bar{x}),\ldots,f'_q(\bar{x}))$$

est une formule **F-ACQ**$^{all\neq}$ d'arbre pondéré $T-\{z\}$: en particulier, notons que $w_{T-\{z\}}(y) = w_T(y) + w_T(z)$. □

Le lemme suivant est l'analogue du lemme 49.

Lemme 63 *Soit $\varphi(\bar{x})$ une σ-formule **F-ACQ**$^{all\neq}$ dont l'arbre pondéré est T et de la forme $\varphi(\bar{x}) = \exists y \exists z \psi(\bar{x},y,z)$ où ψ est une formule sans quantificateur, z est une feuille de T et y est le parent de z dans l'arbre, $y,z \notin \bar{x}$. Soient les entiers $k = w_T(z)$, $l = w_T(T)$ et m le nombre d'égalités dans ψ faisant intervenir la variable z. Alors, on peut calculer, pour toute σ-structure $\mathcal{M}$, une σ'-formule sans quantificateur $\psi'(\bar{x},y)$ et une σ'-structure $\mathcal{M}'$, telles que*

- *$\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$ où $\varphi'(\bar{x}) = \exists y \psi'(\bar{x},y)$;*
- *$\psi'(\bar{x},y)$ est une formule **F-ACQ**$^{all\neq}$ avec pour arbre pondéré $T-\{z\}$ et donc $|\psi'| \leq |\psi|$;*
- *$card(\sigma') = card(\sigma) + k$; plus précisément $card(\sigma'_y) = card(\sigma_y) + k$;*
- *dans la transformation $\mathcal{M} \mapsto \mathcal{M}'$, les domaines des variables ne sont pas modifiés à l'exception du domaine de y dont la cardinalité est multipliée par un nombre inférieur ou égal à $h(k,l) = O(k^l)$;*
- *la transformation $(\varphi,\mathcal{M}) \mapsto (\varphi',\mathcal{M}')$ est calculable en temps $O(m \cdot card(D_z) + k^l(card(D_y) + card(D_z)))$ où D_y et D_z sont les domaines respectifs des variables y et z dans $\mathcal{M}$.*

Preuve. La preuve est complètement analogue à celle du lemme 49 et le lecteur pourra facilement adapter et vérifier tous les détails. □

Nous analysons maintenant la complexité du model-checking pour les formules **F-ACQ**$^{all\neq}$, complexité meilleure que celle donnée pour les formules **F-ACQ**$^{\neq}$ quelconques.

Proposition 64 *Soit φ un σ-énoncé de **F-ACQ**$^{all\neq}$ (resp. **ACQ**$^{all\neq}$), ayant une clique d'inégalités à k éléments. Pour chaque σ-structure $\mathcal{M}$, la question $\mathcal{M} \models \varphi$ peut être décidée en temps $O(2^{O(k\log k)} \cdot |\varphi| \cdot |\mathcal{M}|)$.*

Preuve. Sans perte de généralité, on peut supposer que l'arbre pondéré T de φ est de branchements au plus binaires. (Pour cela, on introduit, par les techniques classiques sur les arbres, des variables-noeuds intermédiaires.) Soit $\bar{x} = (x_1, \ldots, x_p)$ un T-ordre des variables de $\varphi \equiv \exists x_1 \ldots \exists x_p \psi(x_1, \ldots, x_p)$ où ψ est une σ-formule **F-ACQ**$^{all\neq}$ sans quantificateur. L'algorithme qui décide $\mathcal{M} \models \varphi$ supprime d'abord successivement les variables $x_p, x_{p-1}, \ldots x_2$ avec $p-1$ itérations de la procédure du lemme 63 et associe à la formule $\varphi_1(x_1) \equiv \exists x_2 \ldots \exists x_p \psi(x_1, \ldots, x_p)$ et à toute σ-structure $\mathcal{M}$, une σ'-formule sans quantificateur $\varphi'_1(x_1)$ avec une seule variable libre et une σ'-structure $\mathcal{M}'$ telles que $\varphi_1(\mathcal{M}) = \varphi'_1(\mathcal{M}')$. Le temps de la dernière étape, le calcul de $\varphi'_1(\mathcal{M}')$, est négligeable en comparaison du temps de l'élimination de quantificateurs. Le fait suivant énonce comment les cardinalités de la signature et des domaines sont augmentées durant la transformation $(\sigma, \varphi, \mathcal{M}) \mapsto (\sigma', \varphi'_1, \mathcal{M}')$.

Fait 65 *Soit $v \in \bar{x}$ une variable de φ. Les bornes supérieures suivantes sont toujours vérifiées durant l'élimination des quantificateurs :*

- $card(\sigma'_v) \leq card(\sigma_v) + k$;
- $card(D'_v) = O(card(D_v) \cdot k^{2k})$.

Preuve. Soit $u_1, \ldots, u_r \in \bar{x}$ la liste des enfants de v dans T, $r \leq 2$ (branchements au plus binaires). Par le lemme 63, quand la variable u_i est éliminée, la cardinalité $\text{card}(\sigma_v)$ de la signature associée à son parent v est augmentée de k_i où k_i est le poids de u_i. De même, la cardinalité du domaine de v est multipliée par un entier inférieur ou égal à la valeur $h(k_i, k) = O(k_i^k)$. On a $k_i \leq w_T(T_{u_i})$, par le lemme 61, et $\sum_{i\in[r]} w_T(T_{u_i}) \leq w_T(T_v) \leq k$ où T_{u_i} (resp. T_v) est le sous-arbre de racine u_i (resp. v) dans T. Cela implique

$$\text{card}(\sigma'_v) \leq \text{card}(\sigma_v) + \sum_{i\in[r]} w_T(T_{u_i}) \leq \text{card}(\sigma_v) + k$$

et

$$\begin{aligned} \text{card}(D'_v) &= O(\text{card}(D_v) \cdot \prod_{i\in[r]} w_T(T_{u_i})^k) \\ &= O(\text{card}(D_v) \cdot k^{2k}) \end{aligned}$$

puisque $r \leq 2$, comme demandé. □

Soit Time(u) le temps de l'élimination de la variable u de parent v dans T. Par le lemme 63, on a

$$\text{Time}(u) = O(m_u \cdot \text{card}(D'_u) + k^k(\text{card}(D'_u) + \text{card}(D'_v)) + k^k \cdot \text{card}(\sigma'_v) \cdot \text{card}(D'_v))$$

où m_u est le nombre d'égalités impliquant u et v dans φ. Par le fait précédent, on a :

$$\text{Time}(u) = O(k^{2k}(m_u + k^k + k^k \text{card}(\sigma)) \cdot \text{card}(D))$$

Au final, le temps total de l'élimination des quantificateurs est

$$\begin{aligned}\sum_{i\in[2,p]} \text{Time}(x_i) &= O(k^{2k}(m\cdot \text{card}(D) + |\varphi|(k^k \text{card}(D) + k^k|\mathcal{M}|)))\\ &= 2^{O(k\log k)}|\varphi|\cdot|\mathcal{M}|\end{aligned}$$

□

La borne de complexité de la proposition précédente s'étend aux requêtes $\mathbf{F\text{-}ACQ}^{all\neq}$ à une variable libre. Ce qui donne la proposition suivante, analogue à la proposition 50.

Proposition 66 *Soit $\varphi(x)$ une σ-formule de $\boldsymbol{F\text{-}ACQ}^{all\neq}$ (resp. $\boldsymbol{ACQ}^{all\neq}$) possédant une seule variable libre x et une clique d'inégalités à k éléments. Pour chaque σ-structure $\mathcal{M}$, la requête $\varphi(\mathcal{M})$ peut être calculée en temps $2^{O(k\log k)}\cdot|\varphi|\cdot|\mathcal{M}|$.*

Preuve. Similaire à celle de la proposition 50 pour $\mathbf{F\text{-}ACQ}^{\neq}$ en utilisant ici les lemmes 63 et 62. □

Nous appliquons maintenant nos résultats aux deux exemples présentés.

Corollaire 67
- *Le problème k-*Path *(version de décision) est décidable en temps $2^{O(k\log k)}|G|$.*
- *Le problème* MultidimensionalMatching *(version de décision) est décidable en temps $2^{O(kl\log(kl))}card(E)$.*

Preuve. On applique la proposition 64 avec les conditions suivantes sur la formule φ, sa taille de clique d'inégalités et la structure $\mathcal{M}$.
- Pour k-Path : $|\varphi| = O(k)$, clique de taille $k+1$, structure $\mathcal{M} = G$;
- Pour MultidimensionalMatching : $|\varphi| = O(kl)$, clique de taille kl et structure $\mathcal{M}$ de taille $k\cdot\text{card}(E)$.

□

Remarque 22 *La même borne de complexité pour le problème de décision k-*Path *a été obtenue par Monien [Mon85]. De meilleures bornes ont été trouvées par la suite : par exemple, Alon et al. [AYZ95] donnent, pour le problème de décision k-*Path*, un algorithme en temps $2^{O(k)}|G|\log|G|$ en utilisant la méthode du "color-coding".*

2.7 Résultat de dichotomie pour les requêtes acycliques

Le but de cette section est d'établir le théorème de dichotomie annoncé pour la complexité du problème de l'énumération pour les requêtes **ACQ** ou $\mathbf{ACQ}^{\neq}$.

2.7.1 Formules connexe-acycliques relationnelles et leur évaluation

Dans le cas relationnel, la classe des formules conjonctives connexe-acycliques (correspondant à la classe **F-CCQ** ou **F-CCQ**$^{\neq}$ du cas fonctionnel) est encore définie par une certaine propriété de connexité portant sur l'hypergraphe (acyclique) H associé à la formule. Cette caractérisation fait intervenir un ensemble $S \subseteq V(H)$ qui, dans le cas d'une formule, est l'ensemble des variables libres de la formule. On utilisera aussi une notion d'extension de l'hypergraphe H par hyperarêtes incluses. En termes de requête relationnelle, cela revient à dire qu'on ajoute à la requête conjonctive des atomes "redondants" qui sont des "projections" (au sens des relations) d'atomes (relations) déjà présents dans la conjonction.

Définition 96 ($\subseteq$-extension) *Soit H un hypergraphe. On dit que l'hypergraphe H' est une* ***$\subseteq$-extension*** *de H si H' est obtenu à partir de H en ajoutant des hyperarêtes incluses. Plus précisément, H' vérifie les deux conditions suivantes :*

- $E(H) \subseteq E(H')$;
- $\forall e' \in E(H')\ \exists e \in E(H)\ e' \subseteq e$.

Définition 97 (S-connexe-acyclique, S-connexe-acyclique par $\subseteq$-extension) *Soit $H = (V, E)$ un hypergraphe et soit un ensemble de sommets $S \subseteq V$. On dit que le couple (T, A) est une structure arborescente S-connexe de H si T est une structure arborescente de H, et A est une partie connexe de l'ensemble des sommets de T, $A \subseteq V(T)$, telle que $\bigcup_{e \in A} e = S$.*

On dit que l'hypergraphe H est S-connexe-acyclique si H admet une structure arborescente S-connexe (T, A).

De façon plus générale encore, un hypergraphe H est S-connexe-acyclique par $\subseteq$-extension s'il existe une $\subseteq$-extension H' de H telle que H' est S-connexe-acyclique.

On voit facilement qu'étant donnés H et $S \subseteq V(H)$, si l'hypergraphe H est S-connexe-acyclique alors il est S-connexe-acyclique par $\subseteq$-extension et que si H est S-connexe-acyclique par $\subseteq$-extension alors il est acyclique.

Définition 98 *Une formule conjonctive* ***CQ*** *(resp.* ***CQ***$^{\neq}$*) φ est connexe-acyclique si l'hypergraphe associé à φ est S-connexe-acyclique par $\subseteq$-extension où S est l'ensemble free(φ) des variables libres de φ.*
On note ***CCQ*** *(resp.* ***CCQ***$^{\neq}$*) la classe des formules φ de* ***CQ*** *(resp.* ***CQ***$^{\neq}$*) qui sont connexe-acycliques.*

Exemple 68 *Soit φ la σ-formule* ***F-ACQ***$^{\neq}$ *suivante telle que $\sigma = \{R\}$ et free(φ) $= \{x, y\}$:*

$$\varphi(x, y) \equiv \exists z \exists t \exists u (R(x, y, z) \wedge R(t, x, y) \wedge R(u, x, y))$$

Trivialement, l'hypergraphe acyclique de φ n'est pas $\{x,y\}$-connexe-acyclique mais il est $\{x,y\}$-connexe-acyclique par $\subseteq$-extension : prendre l'extension H' de H telle que $E(H') = E(H) \cup \{\{x,y\}\}$. Dans le cadre relationnel, cela revient à ajouter dans la matrice de φ un atome de la forme $R'(x,y)$ (resp. étendre toute σ-structure $\mathcal{M}$ par une relation binaire R', projection de la relation R sur ses deux premiers arguments), ce qui donne une σ'-formule

$$\varphi(x,y) = \exists z \exists t \exists u (R'(x,y) \wedge R(x,y,z) \wedge R(t,x,y) \wedge R(u,x,y))$$

(avec $\sigma' = \sigma \cup \{R'\}$) et une σ'-expansion $\mathcal{M}'$ de $\mathcal{M}$ telles que $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$.

Cet exemple suggère le résultat général suivant qui établit la correspondance entre le cadre relationnel et le cadre fonctionnel.

Lemme 69 *Soit φ une formule **CCQ**$^{\neq}$. Alors il existe une formule φ'' de **F-CCQ**$^{\neq}$ et une réduction exacte du problème* Enum(φ) *au problème* Enum(φ'').

Preuve. Soient φ une σ-formule de $\mathbf{CCQ}^{\neq}$ de la forme

$$\varphi(\bar{x}) \equiv \exists \bar{y} \psi(\bar{x}, \bar{y})$$

où ψ est sans quantificateur, H est l'hypergraphe associé à φ et H' est une $\subseteq$-extension de H qui est free(φ)-connexe-acyclique. Soit $\mathcal{M}$ une σ-structure.

Considérons la signature $\sigma' = \sigma \cup \{R_e : e \in E(H')\}$ où R_e est un nouveau symbole de relation d'arité card(e) associé à l'hyperarête e. On construit la σ'-formule suivante :

$$\varphi'(\bar{x}) \equiv \exists \bar{y} (\psi(\bar{x}, \bar{y}) \wedge \bigwedge_{e' \in E(H') - E(H)} R_{e'}(\bar{z_{e'}}))$$

où $\bar{z_{e'}}$ est le tuple de variables de e' : si $e' = \{v_1, \ldots, v_k\}$, on prend $\bar{z_{e'}} = (v_1, \ldots, v_k)$.

On construit la σ'-structure $\mathcal{M}'$ qui est l'expansion de $\mathcal{M}$ telle que, pour chaque hyperarête $e' \in E(H') - E(H)$ telle que $e' \subseteq e \in E(H)$, $R_{e'}$ est la projection de R_e sur les variables de e'. Les atomes ajoutés $R_{e'}(\bar{z_{e'}})$ étant redondants dans φ', on voit que $\varphi'(\mathcal{M}') = \varphi(\mathcal{M})$. De plus, $\mathcal{M}'$ est calculable en temps $O(|\mathcal{M}|)$.

Soit (T, A) une structure arborescente free(φ)-connexe de H'. On désigne par ψ' la matrice de φ'. On introduit, pour chaque σ-atome A_i, d'hyperarête associée $e \in A$, une nouvelle variable α_i et pour chaque σ-atome B_i, d'hyperarête associée $e \notin A$, une nouvelle variable β_i. Considérons la formule

$$\varphi''(\alpha_1, \ldots, \alpha_l) \equiv \exists \beta_1 \ldots \exists \beta_m \psi''(\alpha_1, \ldots, \alpha_l, \beta_1, \ldots, \beta_m)$$

où ψ'' est la formule obtenue à partir de ψ' par la traduction (1) des formules $\mathbf{ACQ}^{\neq}$ en formules $\mathbf{F\text{-}ACQ}^{\neq}$.

On constate les deux points suivants.

- Par construction, la formule φ'' est dans $\mathbf{F\text{-}CCQ}^{\neq}$;
- Il existe une réduction exacte du problème $\textsc{Enum}(\varphi)$ au problème $\textsc{Enum}(\varphi'')$: c'est la composition de la réduction exacte de $\textsc{Enum}(\varphi)$ à $\textsc{Enum}(\varphi')$ considérée ci-dessus (rappel : $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$) et de la réduction exacte de $\textsc{Enum}(\varphi')$ à $\textsc{Enum}(\varphi'')$ correspondant à la traduction (1).

On a donc prouvé le lemme.

□

Le théorème suivant est une conséquence immédiate des Lemmes 69 et 57

Théorème 70 *Soit φ une formule $\boldsymbol{CCQ}^{\neq}$. Alors $\textsc{Enum}(\varphi) \in \textsc{Constant-Delay}_{lin}$.*

Nous venons de voir que les formules connexe-acycliques sont optimales en termes de complexité d'énumération. Dans la section suivante, nous allons montrer que la reconnaissance de ces formules s'effectue en temps linéaire.

2.7.2 Caractérisation des hypergraphes S-connexe-acycliques par $\subseteq$-extension

Le but de cette section est de donner une caractérisation des hypergraphes S-connexe-acycliques par $\subseteq$-extension et un algorithme pour tester si un hypergraphe H possède cette propriété par rapport à un ensemble $S \subseteq V(H)$. Pour cela, nous introduisons la notion de S-chemin.

Définition 99 (S-chemin) *Soit $H = (V, E)$ un hypergraphe et soit $S \subseteq V$. Un S-chemin est un chemin $(x, a_1, \ldots, a_k, y)$, $k \geq 1$, de l'hypergraphe H tel que*

- *x et y appartiennent à S,*
- *les sommets intermédiaires $a_i, i \in [k]$, appartiennent à $V - S$,*
- *il n'existe pas d'hyperarête qui contient à la fois x et y.*

Le lemme principal de cette sous-section est le suivant :

Lemme 71 *Un hypergraphe H est S-connexe-acyclique par $\subseteq$-extension si et seulement si H est acyclique et n'admet pas de S-chemin.*

Pour prouver le lemme 71, nous avons besoin des notions et faits suivants.

Lemme 72 *Soit $H = (V, E)$ un hypergraphe acyclique et A une partie de V. S'il n'existe pas d'hyperarête $e \in E$ telle que $A \subseteq e$ alors il existe deux sommets $x, y \in A$ non voisins dans H.*

Preuve. Ce lemme ne fait qu'exprimer qu'un hypergraphe acyclique est conforme [BFMY83].
□

Définition 100 (structure arborescente induite) *Soit T une structure arborescente d'un hypergraphe $H = (V, E)$ et soit une partie $A \subseteq V$. On note $T[A]$ l'arbre T dans lequel chaque noeud e est remplacé par $e \cap A$.*

On a, pour chaque hypergraphe H et chaque partie $A \subseteq V(H)$, les propriétés suivantes.

Lemme 73 *Soient $H = (V, E)$ un hypergraphe, A une partie de V et $H[A]$ l'hypergraphe induit de H par A.*

1. *Si H est acyclique alors $H[A]$ est acyclique.*
2. *Pour tout ensemble $S \subseteq V$, si H est S-connexe-acyclique alors $H[A]$ est $(S \cap A)$-connexe-acyclique.*
3. *Pour tout ensemble $S \subseteq V$, si H est S-connexe-acyclique par $\subseteq$-extension alors $H[A]$ est $(S \cap A)$-connexe-acyclique par $\subseteq$-extension.*

Preuve. (1) Soit T une structure arborescente de H. Clairement, l'hypergraphe $H[A]$ est acyclique, de structure arborescente $T[A]$.

(2) Si (T, B) est une structure aborescente S-connexe de H, alors le couple $(T[A], B_A)$ où $B_A = \{e \cap A : e \in B\}$ est une structure arborescente $(S \cap A)$-connexe de l'hypergraphe induit $H[A]$ car l'égalité $S = \bigcup_{e \in B} e$ implique $S \cap A = \bigcup_{e \in B}(e \cap A) = \bigcup_{e \in B_A} e$.

(3) Soit H' une $\subseteq$-extension de l'hypergraphe H qui est S-connexe-acyclique. Alors l'hypergraphe induit $H'[A]$ est une $\subseteq$-extension de l'hypergraphe induit $H[A]$, et, par le point (2), $H'[A]$ est un hypergraphe $(S \cap A)$-connexe-acyclique. □

Dans le but de construire une structure arborescente S-connexe d'une $\subseteq$-extension de H, nous allons décomposer les hyperarêtes de H en "composantes connexes".

Définition 101 (S-composante) *Pour un hypergraphe $H = (V, E)$ et une partie $S \subseteq V$, on note $E_{\not\subseteq S}$ l'ensemble des hyperarêtes $\{e \in E : e \not\subseteq S\}$.*

On appelle S-composante d'une hyperarête $e \in E_{\not\subseteq S}$ l'ensemble des hyperarêtes $e' \in E_{\not\subseteq S}$ telles que dans l'hypergraphe induit $H[V - S]$, les hyperarêtes induites $e - S$ et $e' - S$ sont reliées par un "chemin" $(e_0, \ldots, e_k)$, $e_0 = e - S$, $e_k = e' - S$, $e_i \in E(H[V - S])$ et $e_i \cap e_{i+1} \neq \emptyset$, pour tout $i < k$.

Il est clair que l'ensemble des hyperarêtes $E_{\not\subseteq S}$ se partitionne en S-composantes disjointes deux à deux.

Pour prouver le lemme 71 et justifier l'algorithme qui le mettra en oeuvre nous prouvons les trois lemmes suivants.

Lemme 74 *Soit $H = (V, E)$ un hypergraphe acyclique de structure arborescente T. Soit une partie $S \subseteq V$ et soit $E_1, \ldots, E_l$ la liste des S-composantes de $E_{\not\subseteq S}$. Pour chaque $i \in [l]$, on note $H_i = (V_i, E_i)$ l'hypergraphe d'ensemble d'hyperarêtes E_i avec $V_i = \bigcup_{e \in E_i} e$ et on note $e_i = V_i \cap S$.*

1. *E_i forme une partie connexe de l'arbre T, le sous-arbre noté T_i, et donc H_i est un hypergraphe acyclique dont une structure arborescente est le sous-arbre T_i de T.*
2. *Soient x, y deux sommets de e_i qui sont voisins dans H. Alors x et y sont aussi voisins dans l'hypergraphe H_i.*
3. *On a l'équivalence suivante : il existe une hyperarête $e'_i \in E_i$ qui contient e_i si et seulement si il existe une hyperarête $e''_i \in E$ qui contient e_i.*

Preuve.

(1) : Pour établir que E_i forme une partie connexe de T, il suffit de prouver que, quelles que soient deux hyperarêtes $e, e' \in E_i$ telles que $(e \cap e') - S \neq \emptyset$ et une hyperarête e'' du chemin reliant e et e' dans l'arbre T, on a encore $e'' \in E_i$. La propriété de connexité de T impose $e \cap e' \subseteq e''$, et donc $(e \cap e' \cap e'') - S \neq \emptyset$, ce qui implique $e'' \in E_i$.

(2) : Soit T_x et T_y les deux sous-arbres de T formés respectivement des noeuds de T contenant x et contenant y. Puisque $x, y \in V_i \cap S$, il existe des hyperarêtes e_x et e_y telles que $x \in e_x \in E_i$ et $y \in e_y \in E_i$. D'où $e_x \in V(T_i) \cap V(T_x) \neq \emptyset$ et $e_y \in V(T_i) \cap V(T_x) \neq \emptyset$, et donc la réunion des sous-arbres $T_i \cup T_x \cup T_y$ forme un sous-arbre (partie connexe) noté T' de T. Supposons que x et y sont voisins dans H par l'arête $e_{x,y}$. Autrement dit, $x, y \in e_{x,y} \in E$. Prouvons par l'absurde que $e_{x,y} \in E_i$, c'est-à-dire supposons $e_{x,y} \notin E_i$. On a donc $e_{x,y} \in V(T_x) \cap V(T_y) - V(T_i)$. Cela implique que dans l'arbre T', on a deux chemins distincts reliant e_x et e_y : d'une part, le chemin formé de la concaténation du chemin reliant e_x à $e_{x,y}$ dans T_x et du chemin reliant e_y à $e_{x,y}$ dans T_y et d'autre part le chemin reliant e_x à e_y dans T_i. Ceci contredit l'acyclicité de l'arbre T'. On a donc prouvé $e_{x,y} \in E_i$.

(3) : Cela découle immédiatement des équivalences suivantes, puisque H et H_i sont des hypergraphes acycliques et donc conformes :

$\exists e'_i \in E_i \; e_i \subseteq e'_i \Leftrightarrow e_i$ est une clique de $H_i \Leftrightarrow e_i$ est une clique de $H \Leftrightarrow \exists e''_i \in E \; e_i \subseteq e''_i$. La seconde équivalence est une conséquence de l'implication (2). Le lemme est donc prouvé.

□

Lemme 75 *Sous l'hypothèse que H est acyclique (ou même seulement que H_i est acyclique) et avec les notations du lemme précédent, les assertions suivantes sont équivalentes :*

1. *H'_i est acyclique ;*
2. *H'_i n'a pas de cycle ;*
3. *e_i est une clique de H_i ;*

4. *e_i est inclus dans une hyperarête e'_i de H_i ;*
5. *H'_i est une $\subseteq$-extension de H_i.*

Preuve. Il suffit de prouver le circuit d'implications (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (4) $\Rightarrow$ (5) $\Rightarrow$ (1). Toutes sont évidentes ou ont déjà été vues, sauf l'implication (3) $\Rightarrow$ (4) qui vient du fait que H_i est conforme et l'implication (2) $\Rightarrow$ (3). Prouvons la contraposée $non(3) \Rightarrow non(2)$. Supposons donc $non(3)$: il existe donc deux sommets $x, y \in e_i$ qui ne sont pas voisins dans H_i. Construisons un cycle de H'_i : il s'agit du chemin reliant x à y dans H_i auquel on ajoute l'hyperarête e_i pour "fermer" le cycle. On a donc prouvé $non(2)$. Le lemme est prouvé. □

Lemme 76 *Soient $H, T, S, E_i, H_i = (V_i, E_i)$, e_i, pour $i \in [l]$, les objets vérifiant les hypothèses du lemme 74 ; en particulier T_i est le sous-arbre de T, structure arborescente de H, qui forme une structure arborescente de H_i. On définit l'hypergraphe H' d'ensemble de sommets V et dont l'ensemble des hyperarêtes est la réunion disjointe*

$$E(H') = E(H[S]) \uplus \biguplus_{i \in [l]} E_i$$

Alors les six propriétés suivantes sont équivalentes :

1. *H est S-connexe-acyclique par $\subseteq$-extension ;*
2. *H ne contient pas de S-chemin ;*
3. *Pour tout $i \in [l]$, il existe une hyperarête $e''_i \in E$ telle que $e_i \subseteq e''_i$;*
4. *Pour tout $i \in [l]$, il existe une hyperarête $e'_i \in E_i$ telle que $e_i \subseteq e'_i$;*
5. *Pour tout $i \in [l]$, H'_i est acyclique ;*
6. *(a) Pour tout $i \in [l]$, $e_i \in E(H[S])$ et il existe une hyperarête $e'_i \in E_i$ telle que $e_i \subseteq e'_i$;*
   (b) H' est S-connexe-acyclique de structure arborescente (T', A) où $A = E(H[S])$ et l'arbre T' est la réunion des structures arborescentes, disjointes deux à deux, T_S de $H[S]$ et T_i de H_i, $i \in [l]$, avec ajout, pour chaque $i \in [l]$, de l'arête $\{e_i, e'_i\}$ reliant T_S et T_i.

Preuve. L'équivalence (3) $\Leftrightarrow$ (4) n'est autre que l'équivalence du Lemme 74(3), pour tout $i \in [l]$. L'implication (6) $\Rightarrow$ (1) découle immédiatement du fait que, par construction, H' est une $\subseteq$-extension de H. L'équivalence (4) $\Leftrightarrow$ (5) n'est autre que l'équivalence (4) $\Rightarrow$ (1) du lemme 75. Il reste donc à prouver les implications (1) $\Rightarrow$ (2), (2) $\Rightarrow$ (3) et (4) $\Rightarrow$ (6).

(1) $\Rightarrow$ (2) : Nous allons prouver cette implication par l'absurde. On suppose donc qu'on a à la fois (1) et non (2) : H est S-connexe-acyclique par $\subseteq$-extension et H possède un S-chemin $P = (x, a_1, \ldots a_k, y)$, $k \geq 1$. Par définition, les sommets x et y appartiennent à S et tous les sommets a_i appartiennent à $V - S$. Sans perte de généralité, on peut supposer que le chemin P est minimal

(pour les sommets). Par le lemme 73, l'hypergraphe induit $H[P]$ est $(S \cap P)$-connexe-acyclique par $\subseteq$-extension. Puisque $S \cap P = \{x, y\}$, $H[P]$ est $\{x, y\}$-connexe-acyclique par $\subseteq$-extension. Or, par définition, $H[P]$ est constitué des hyperarêtes binaires $\{x, a_1\}, \{a_1, a_2\}, \dots \{a_{k-1}, a_k\}, \{a_k, y\}$ avec, éventuellement, en plus, des hyperarêtes singletons. Par conséquent, toute $\subseteq$-extension de $H[P]$ est encore de la même forme. En conséquence, toute structure arborescente de l'hypergraphe extension admet le "chemin" $\{x, a_1\}, \{a_1, a_2\}, \dots \{a_{k-1}, a_k\}, \{a_k, y\}$, avec $k \geq 1$, et n'est donc pas $\{x, y\}$-connexe. C'est absurde puisqu'on a dit que $H[P]$ est $\{x, y\}$-connexe-acyclique par $\subseteq$-extension. On a donc prouvé (1) $\Rightarrow$ (2).

(2) $\Rightarrow$ (3) : Prouvons la contraposée non(3) $\Rightarrow$ non(2). Supposons donc que, pour un certain $i \in [l]$, il n'existe aucune hyperarête $e \in E$ telle que $e_i \subseteq e$. Toutes les hypothèses du lemme 72 sont donc réalisées avec l'hypergraphe $H = (V, E)$ et la partie e_i de V. Par le lemme 72, il existe deux sommets $x, y \in e_i$ non voisins dans H. Par définition de e_i, il existe deux hyperarêtes de E_i, e_x et e_y, qui contiennent respectivement x et y. Puisque e_x et e_y sont dans $E_{\not\subseteq S}$, il existe deux sommets, non nécessairement distincts, $x' \in e_x - S$ et $y' \in e_y - S$. Par définition de la S-composante E_i, dans l'hypergraphe induit $H[V - S]$, les hyperarêtes $e_x - S$ et $e_y - S$ appartiennent à un "chemin" $(e_0, e_1, \dots, e_k)$, avec $e_0 = e_x - S$, $e_k = e_y - S$, et $e_i \cap e_{i+1} \neq \emptyset$, pour tout $i < k$. Autrement dit, on a dans $H[V - S]$ un chemin $(a_1, \dots, a_k)$ où chaque a_i est un sommet choisi de $e_{i-1} \cap e_i$, donc dans $V - S$. On en déduit que $(x, a_1, \dots, a_k, y)$ est un S-chemin de H.

(4) $\Rightarrow$ (6) : On suppose donc que, pour chaque $i \in [l]$, il existe une hyperarête $e'_i \in E_i$ telle que $e_i \subseteq e'_i$. On va en déduire les conditions (a) et (b) du (6).

(a) : Puisque $V_i = \bigcup_{e \in E_i} e$ on a $e'_i \subseteq V_i$. On en déduit la chaîne d'inclusions : $e_i \subseteq e'_i \cap S \subseteq V_i \cap S = e_i$, ce qui implique $e_i = e'_i \cap S$ et donc $e_i \in E(H[S])$ puisque $e'_i \in E(H)$. On a donc prouvé (a).

(b) : T' est par construction un arbre dont l'ensemble des noeuds est $E(H')$. On a $\bigcup_{e \in A} e = S$ pour $A = E(H[S])$ puisque tout sommet de V et donc tout sommet de S appartient à une hyperarête de H. Il reste à vérifier que T' vérifie la propriété de connexité. Puisque chacun de ses sous-arbres T_S et T_i, $i \in [l]$, la vérifie, il suffit de montrer que si un sommet x appartient à deux noeuds de deux sous-arbres différents parmi ces $l + 1$ sous-arbres, alors x appartient aussi aux noeuds e_i et e'_i reliant ces sous-arbres. Soit $x \in e \cap e'$ avec $e \in E(H[S])$ et $e' \in E_i$. On en déduit $x \in e \cap e' \subseteq S \cap V_i = e_i \subseteq e'_i$. Donc $x \in e_i$ et $x \in e'_i$, comme demandé. Soit maintenant $x \in e \cap e'$ avec $e \in E_i$ et $e' \in E_j$, pour $i \neq j$. Par définition des S-composantes, $e \cap e' \subseteq S$. Donc $x \in S$ et, puisque $x \in e \cap S \in E(H[S])$ et $x \in e' \cap S \in E(H[S])$, on obtient $x \in e_i$, $x \in e'_i, x \in e_j, x \in e'_j$ comme dans le cas précédent. En conséquence, T' vérifie la propriété de connexité et donc (T', A) est une structure arborescente S-connexe de H'.

□

Remarque 23 *Finalement, l'équivalence* (1) $\Leftrightarrow$ (2) *du lemme 76 prouve le lemme 71 puisqu'un hypergraphe S-connexe acyclique par $\subseteq$-extension est acyclique.*

Les équivalences précédentes permettent de reconnaître efficacement les hypergraphes S-connexes-acycliques par $\subseteq$-extension, c'est-à-dire de résoudre le problème suivant.

CONNEX-ACYCLIC	
Entrée:	Un hypergraphe $H = (V, E)$ et un ensemble $S \subseteq V$
Sortie:	H est-il S-connexe-acyclique par $\subseteq$-extension ? Si oui, donner un hypergraphe S-connexe-acyclique H' qui est une $\subseteq$-extension de H et donner sa structure arborescente S-connexe (T', A).

Théorème 77 *Il existe un algorithme linéaire qui calcule le problème* CONNEX-ACYCLIC.

Preuve. Le problème CONNEX-ACYCLIC est calculé par l'algorithme suivant :

- Si H n'est pas acyclique renvoyer Faux ;
- Construire l'hypergraphe induit $H[S]$;
- Construire les S-composantes E_i de H et donc les hypergraphes $H_i = (V_i, E_i)$ et aussi les ensembles $e_i = V_i \cap S$, $i \in [l]$;
- Tester si $H'_i = H_i + e_i$ est acyclique, pour tout $i \in [l]$;
- Si l'un n'est pas acyclique, renvoyer Faux ;
- Construire la structure arborescente S-connexe (T', A) de la condition (6)(b).

Complexité : On voit facilement que chaque étape se fait en temps linéaire $O(|H|)$. On applique à chacun des hypergraphes H et H'_i, $i \in [l]$, l'algorithme de Tarjan et Yannakakis [TY84, TY85] qui décide si un hypergraphe H est acyclique et, si oui, calcule une structure arborescente de H, tout ceci en temps linéaire.

Correction : justifiée par les équivalences précédentes.

□

Exemples d'exécution de l'algorithme pour CONNEX-ACYCLIC

1) Soit $H = (V, E)$ avec $V = \{x, y, z\}$, $E = \{\{x, y\}, \{y, z\}\}$ et $S = \{x, z\}$ (donc $V - S = \{y\}$). Alors $H[S] = (S, E_S)$ avec $E_S = \{\{x\}, \{z\}\}$. $E_{\not\subseteq S}$ a exactement une S-composante $E_1 = \{\{x, y\}, \{y, z\}\} = E$ et donc $E(H'_1) = E_1 \cup \{e_1\} = \{\{x, y\}, \{y, z\}, \{x, z\}\}$ et H'_1 n'est pas acyclique. Donc H n'est pas connexe-acyclique par $\subseteq$-extension.

2) Soit l'hypergraphe $H = (V, E)$ représenté par la figure 2.3 avec $V = \{a, b, c, d, e, f, g, h, i\}$, $E = \{\{a, b, c\}, \{c, d\}, \{b, c, d, e, f\}, \{c, d, f, g, h\}, \{g, i\}\}$ et $S = \{c, d, f, g\}$.

Alors $H[S] = (S, E_S)$ avec $E_S = \{\{c\}, \{c, d\}, \{c, d, f\}, \{c, d, f, g\}, \{g\}\}$ qui est acyclique. $E_{\not\subseteq S}$ possède les trois S-composantes

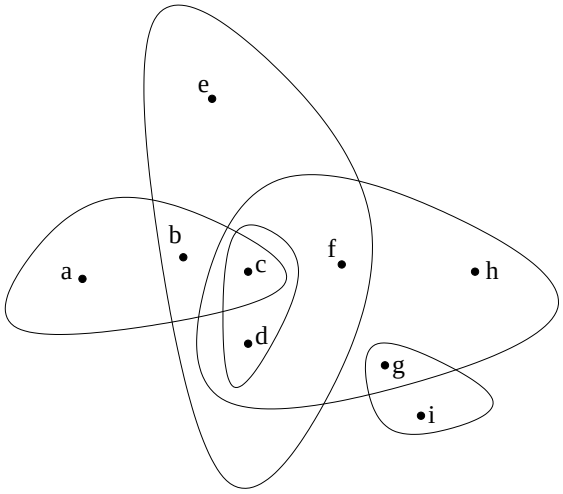


FIG. 2.3 – L'hypergraphe H avec $S = \{c, d, f, g\}$

- $E_1 = \{\{a, b, c\}, \{b, c, d, e, f\}\}$,
- $E_2 = \{\{c, d, f, g, h\}\}$,
- $E_3 = \{\{g, i\}\}$.

On a donc $e_1 = V_1 \cap S = \{c, d, f\}$, $e_2 = \{c, d, f, g\}$, $e_3 = \{g\}$.

On vérifie aisément que les trois hypergraphes $H'_i = H_i + e_i$ sont acycliques. L'hypergraphe H est donc S-connexe-acyclique par $\subseteq$-extension.

2.7.3 Résultat de dichotomie

Nous souhaitons montrer que le problème de l'évaluation d'une requête conjonctive acyclique qui n'est pas **CCQ** est "difficile" dans un sens précis. Cela établira une forme de maximalité des requêtes **CCQ** et $\mathbf{CCQ}^{\neq}$ pour le théorème 70. Dans ce but, nous allons exhiber une réduction exacte du problème du produit de deux matrices booléennes, de dimension $n \times n$, problème noté MATRIXPRODUCT, au problème ENUM(φ) et on s'appuie sur la conjecture suivante : le problème MATRIXPRODUCT n'est pas calculable en temps $O(n^2)$ (voir par exemple [CKSU05]).

MATRIXPRODUCT	
Entrée:	Deux matrices booléennes A et B de dimensions $n \times n$
Sortie:	Le produit $A \times B$

Nous allons coder le problème du produit de matrices booléennes par une requête **ACQ**. Pour cela, nous codons la donnée de ce problème, c'est-à-dire deux matrices booléennes $n \times n$, par une structure relationnelle.

Définition 102 *Une structure bimatricielle est une σ_{AB}-structure relationnelle $\mathcal{M} = (D, A, B)$*

où $D = [n], \sigma_{AB} = \{A, B\}$, et A, B sont des relations binaires sur D. Le problème de la multiplication de deux matrices booléennes $n \times n$ est exprimé par la σ_{AB}-requête conjonctive acyclique suivante :

$$\Pi(x, y) \equiv \exists z(A(x, z) \wedge B(z, y))$$

Plus précisément, Enum(Π) *est le problème d'énumérer, pour chaque structure bimatricielle $\mathcal{M} = ([n], A, B)$ donnée, les couples d'indices $(i, j) \in [n]^2$ où 1 apparaît dans la matrice produit $C = A \times B$, i.e. $C_{i,j} = 1$, pour $i, j \in [n]$.*

Pour des raisons techniques, nous imposons à nos formules d'être simples dans le sens suivant.

Définition 103 *Une formule conjonctive est dite simple si chaque symbole de relation apparaît dans au plus un atome.*

Remarque 24 *Etant données une σ-formule φ de $\boldsymbol{ACQ}$ (resp. $\boldsymbol{CCQ}$, $\boldsymbol{ACQ}^{\neq}$, $\boldsymbol{CCQ}^{\neq}$) et une σ-structure $\mathcal{M}$, on peut calculer en temps linéaire, une σ'-formule simple φ' de $\boldsymbol{ACQ}$ (resp. $\boldsymbol{CCQ}$, $\boldsymbol{ACQ}^{\neq}$, $\boldsymbol{CCQ}^{\neq}$) et une σ'-structure $\mathcal{M}'$ telles que $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$.*

Preuve. Il suffit de dupliquer chaque (symbole de) relation autant de fois que ce symbole apparaît dans φ. □

Lemme 78 *Soit φ une formule simple de $\boldsymbol{ACQ}$ (resp. $\boldsymbol{ACQ}^{\neq}$) mais qui n'est pas dans $\boldsymbol{CCQ}$ (resp. qui n'est pas dans $\boldsymbol{CCQ}^{\neq}$). Alors il existe une réduction exacte du problème* Enum(Π) *au problème* Enum(φ).

Preuve. Supposons $\varphi \in \mathbf{ACQ} - \mathbf{CCQ}$. L'hypergraphe $H_\varphi = (V, E)$ de φ est acyclique mais n'est pas S-connexe par $\subseteq$-extension pour $S = \text{free}(\varphi)$. Par le lemme 71, H_φ admet un S-chemin qu'on peut supposer minimal : c'est une séquence de sommets distincts $P = (x, z_1, \ldots, z_k, y)$, pour $k \geq 1$, telle que

- P est un chemin de H_φ : il existe $k+1$ hyperarêtes $e_0, e_1, \ldots, e_{k-1}, e_k \in E$ qui contiennent les paires respectives $e'_0 = \{x, z_1\}, e'_1 = \{z_1, z_2\}, \ldots, e'_{k-1} = \{z_{k-1}, z_k\}, e'_k = \{z_k, y\}$;
- P est un S-chemin de H_φ : $x, y \in S$ et $z_1, \ldots, z_k \notin S$;
- P est minimal : pour chaque $e \in E, \text{card}(e \cap P) \leq 1$ ou $e \cap P = e'_i$, pour un certain $i \in [0, k]$.

Sans perte de généralité, supposons que la σ-formule (simple) φ est de la forme

$$\varphi(x, y, \bar{t}) \equiv \exists z_1 \ldots \exists z_k \exists \bar{u} \psi(x, y, \bar{z}, \bar{t}, \bar{u})$$

où $k \geq 1$, $\bar{t} = (t_1, \ldots, t_m), m \geq 0, \bar{u} = (u_1, \ldots, u_q), q \geq 0$, ψ est de la forme $\psi \equiv \bigwedge_{e \in E} A_e$ où $A_e = R_e(L_e)$ et L_e est une suite de variables distinctes qui consiste en tous les éléments (sommets) de l'hyperarête e. On peut supposer (sans perte de généralité) que chaque atome A_e de φ est d'une des σ-formes suivantes $(1-5)$ où $\bar{v} \subseteq \{\bar{t}, \bar{u}\}$:

1. $R_e(x, z_1, \bar{v})$;
2. $R_e(z_k, y, \bar{v})$;
3. $R_e(z_i, z_{i+1}, \bar{v})$ où $1 \leq i < k$;
4. $R_e(w, \bar{v})$ où $w \in \{x, y, z_1, \ldots, z_k\}$;
5. $R_e(\bar{v})$.

Transformation de la structure : A chaque σ_{AB}-structure $\mathcal{M} = (D, A, B)$, on associe une structure $r(\mathcal{M}) = \mathcal{M}'$ de signature $\sigma = \{R_e : e \in E\}$, définie de la manière suivante.

- $\mathcal{M}' = (D', (R_e)_{e \in E})$;
- $D' = D \cup \{\bot\}$: ici $\bot$ est un spécial de "remplissage" ;
- A chaque hyperarête $e \in E$ de cardinalité p, on associe une relation R_e d'arité p, définie en fonction de la forme (1-5 : voir au dessus) de son unique occurrence dans φ :
  1. $R_e = A \times \{\bot\}^{p-2}$;
  2. $R_e = B \times \{\bot\}^{p-2}$;
  3. $R_e = I \times \{\bot\}^{p-2}$ où I est la relation d'identité (égalité) de D ($I = \{(a, a) : a \in D\}$) ;
  4. $R_e = D \times \{\bot\}^{p-1}$;
  5. $R_e = \{\bot\}^p$.

Les faits suivants sont faciles à prouver :

Fait 1 : La fonction $r : \mathcal{M} \mapsto \mathcal{M}'$ est calculable en temps $O_\varphi(|\mathcal{M}|)$.

Fait 2 : $\varphi(\mathcal{M}') = \Pi(\mathcal{M}) \times \{\bot\}^m$.

Indication : Les variables x, y jouent le même rôle dans $\varphi(x, y, \bar{t})$ que dans $\Pi(x, y)$.

Du fait 2, on déduit le fait suivant.

Fait 3 : La projection $(a, b, \bar{c}) \mapsto (a, b)$ est une fonction bijective de l'ensemble $\varphi(\mathcal{M}')$ dans l'ensemble $\Pi(\mathcal{M})$.

Grâce à ces faits, nous avons exhibé et justifié une réduction exacte du problème Enum(Π) au problème Enum(φ). Cela prouve le lemme pour $\varphi \in \mathbf{ACQ} - \mathbf{CCQ}$. Dans le cas où φ appartient à $\mathbf{ACQ}^{\neq} - \mathbf{CCQ}^{\neq}$, nous pouvons facilement adapter la réduction de la manière suivante. On oblige chaque paire de variables à avoir des domaines distincts : plus précisément, on duplique le domaine D' en $D' \times [h]$ où $h = \text{card}(\text{var}(\varphi))$, et on donne à la i-ème variable, $i \in [h]$, le domaine spécifique $D_i = D' \times \{i\}$. Enfin, on modifie la structure $\mathcal{M}'$ en conséquence. □

Finalement, on a prouvé le résultat suivant :

Théorème 79 *[Théorème de dichotomie] Soit φ une formule simple **ACQ** (resp. $\mathbf{ACQ}^{\neq}$). Supposons que le produit de deux matrices booléennes $n \times n$ ne peut pas être calculé en temps $O(n^2)$. Alors les trois propriétés suivantes sont équivalentes :*

1. *$\varphi \in$ **CCQ** (resp. $\mathbf{CCQ}^{\neq}$) ;*

2. ENUM(φ) $\in$ CONSTANT-DELAY$_{lin}$;
3. *Le problème* ENUM(φ) *est calculable en temps total* $O_\varphi(|\mathcal{M}| + |\varphi(\mathcal{M})|)$.

Preuve. On a, sous l'hypothèse du théorème, les implications suivantes : on a (1) $\Rightarrow$ (2) par le théorème 70 ; (2) $\Rightarrow$ (3) est évident ; (3) $\Rightarrow$ (1) s'obtient par contraposée. En effet, soit une formule $\varphi \in \mathbf{ACQ} - \mathbf{CCQ}$ (resp. $\varphi \in \mathbf{ACQ}^{\neq} - \mathbf{CCQ}^{\neq}$). Par le lemme 78, il existe une réduction exacte du problème ENUM(Π) au problème ENUM(φ). Or, par hypothèse, le problème ENUM(Π) qui n'est autre que le problème MATRIXPRODUCT n'est pas calculable en temps linéaire. Donc le problème ENUM(φ) n'est pas calculable en temps $O_\varphi(|\mathcal{M}| + |\varphi(\mathcal{M})|)$. En effet, s'il l'était, alors, par réduction exacte donc linéaire, le problème ENUM(Π) serait aussi calculable en temps linéaire. On a donc prouvé (3) $\Rightarrow$ (1). □

2.7.4 Résultat de trichotomie

Nous avons classé les formules acycliques (avec ou sans inégalités) en deux catégories : celles énumérables à délai constant et celles qui ne le sont pas. Il serait intéressant de voir ce qu'il advient dans le cas où une formule conjonctive n'est pas acyclique. Nous allons tout d'abord montrer que si φ est une formule simple de **CQ** dont l'hypergraphe possède un cycle alors φ "exprime" le problème TRIANGLE suivant.

TRIANGLE	
Entrée:	Un graphe (non orienté) G
Sortie:	G admet-il un triangle (une clique de taille 3) ?

On ne connaît pas actuellement d'algorithme linéaire pour tester si un graphe contient un triangle. On conjecture qu'il n'en existe pas.

Lemme 80 *Soit φ une σ-formule simple de **CQ** (resp. **CQ**$^{\neq}$) tel que l'hypergraphe $H_\varphi = (V_\varphi, E_\varphi)$ de φ contient un cycle. Alors il existe une réduction linéaire du problème* TRIANGLE *au problème* MC(φ).

Preuve. La preuve est très similaire à celle du lemme 78. Soit une σ-formule simple $\varphi \in \mathbf{CQ}$ dont l'hypergraphe $H_\varphi = (V_\varphi, E_\varphi)$ contient un cycle $C = (x_1, x_2 \ldots, x_k)$ où $x_1, \ldots, x_k$ sont des variables distinctes.

Sans perte de généralité, supposons que φ est de la forme

$$\varphi(x_1, \ldots, x_k, y_1, \ldots, y_m) \equiv \bigwedge_{e \in E_\varphi} A_e$$

où $A_e = R_e(L_e)$ et L_e est une suite de variables distinctes qui consiste en tous les éléments (sommets) de e. On peut supposer (sans perte de généralité) que chaque atome A_e de φ est

d'une des σ-formes suivantes $(1-6)$ où $\bar{v} \subseteq \{y_1, \ldots, y_m\}$:

1. $R_e(x_1, x_2, \bar{v})$;
2. $R_e(x_2, x_3, \bar{v})$;
3. $R_e(x_1, x_k, \bar{v})$;
4. $R_e(x_i, x_{i+1}, \bar{v})$ où $3 \leq i < k$;
5. $R_e(w, \bar{v})$ où $w \in \{x_1, \ldots, x_k\}$;
6. $R_e(\bar{v})$.

Transformation de la structure : A chaque graphe $G = (V, E)$, on associe une structure $r(G) = \mathcal{M}_G$ de signature $\sigma = \{R_e : e \in E_\varphi\}$ définie de la manière suivante.

- $\mathcal{M}_G = (D, (R_e)_{e \in E_\varphi})$;
- $D = V \cup \{\bot\}$: $\bot$ est un symbole spécial de "remplissage" ;
- A chaque hyperarête $e \in E_\varphi$ de cardinalité p, on associe une relation R_e d'arité p, définie selon la forme (1-6 : voir au dessus) de l'unique occurrence de R_e dans φ :
  1. $R_e = E \times \{\bot\}^{p-2}$;
  2. $R_e = E \times \{\bot\}^{p-2}$;
  3. $R_e = E \times \{\bot\}^{p-2}$;
  4. $R_e = I \times \{\bot\}^{p-2}$ où I est la relation d'identité (égalité) de V $(I = \{(a, a) : a \in V\})$;
  5. $R_e = V \times \{\bot\}^{p-1}$;
  6. $R_e = \{\bot\}^p$.

Les faits suivants sont faciles à prouver :

Fait 1 : La fonction $r : G \mapsto \mathcal{M}_G$ est calculable en temps $O_\varphi(|G|)$.

Fait 2 : $\varphi(\mathcal{M}_G) = \textsc{Triangle}(G) \times \{\bot\}^{k+m-3}$ où $\textsc{Triangle}(G)$ désigne l'ensemble des triplets $(a, b, c) \in V^3$ qui forment des triangles de G.

On a ainsi prouvé le lemme dans le cas où $\varphi \in \mathbf{CQ}$ et H_φ possède un cycle. Le cas $\varphi \in \mathbf{CQ}^{\neq}$ (et H_φ possède un cycle) se traite avec la même adaptation que dans la preuve du Lemme 78. Ce qui clôt la preuve du Lemme. □

Malheureusement, tout hypergraphe non acyclique ne possède pas nécessairement un cycle. Cependant cette propriété est vérifiée pour une large classe d'hypergraphes.

Définition 104 *Un hypergraphe H est dit k-conforme si toute clique de H de taille supérieure ou égale à k est contenue dans une hyperarête de H.*

Le lemme suivant découle du théorème 1.

Lemme 81 *Un hypergraphe 4-conforme est acyclique si et seulement si il ne contient pas de cycle.*

On en déduit le théorème suivant :

Théorème 82 (trichotomie) *Soit φ une formule simple **CQ** (resp. **CQ**$^{\neq}$) ayant un hypergraphe 4-conforme. Supposons que le problème* Triangle *ne peut pas être décidé en temps linéaire et que le produit de deux matrices booléennes $n \times n$ ne peut pas être calculé en temps $O(n^2)$. Alors, les deux propriétés suivantes sont vérifiées.*

1. *La formule φ est acyclique si et seulement si le problème* Enum(φ) *peut être calculé à délai $O_\varphi(|\mathcal{M}|)$.*
2. *Si φ est acyclique alors les trois propositions suivantes sont équivalentes :*
   - *$\varphi \in$ **CCQ** (resp. **CCQ**$^{\neq}$) ;*
   - *Enum$(\varphi) \in$ Constant-Delay$_{lin}$;*
   - *Le problème* Enum(φ) *est calculable en temps total $O_\varphi(|\mathcal{M}| + |\varphi(\mathcal{M})|)$.*

Preuve. Point 1 : La preuve de l'implication $\Rightarrow$ est donnée par la proposition 52. On prouve la réciproque par contraposée. Supposons que φ est une formule conjonctive simple non acyclique et dont l'hypergraphe est 4-conforme. D'après le lemme 81, cet hypergraphe possède un cycle. D'après le lemme 80, il existe une réduction linéaire du problème Triangle au problème mc(φ). Or, comme le problème Triangle n'est pas, par hypothèse, décidable en temps linéaire, le problème mc(φ) ne l'est pas non plus et donc le problème Enum(φ) n'est pas calculable à délai linéaire. Le point 2 est exactement le théorème 79. □

2.8 Requêtes de largeur arborescente connexe bornée

2.8.1 Définition et évaluation

Dans cette section, nous introduisons la notion de décomposition arborescente connexe d'une formule conjonctive : c'est une variante de la notion de décomposition arborescente d'un hypergraphe ou d'une formule conjonctive. Pour simplifier les notations, on considérera des décompositions arborescentes $(T, (B_t)_{t \in V(T)})$ telles que, pour chaque noeud $t \in V(T)$, on a $t = B_t$, autrement dit, telles que les sacs sont identifiés aux noeuds de l'arbre T. La décomposition pourra alors être représentée uniquement par T.

Définition 105 (décomposition arborescente S-connexe) *Soient $H = (V, E)$ un hypergraphe et S un sous-ensemble de V.*

Une décomposition arborescente S-connexe de H est un couple (T, A) où T est une décomposition arborescente de H et A est une partie connexe de $V(T)$ tel que $\bigcup_{B \in A} B = S$.

Définition 106 *Soient un hypergraphe $H = (V, E)$ et un ensemble de sommets $S \subseteq V$. On appelle largeur arborescente S-connexe de H, notée $ctw(H, S)$, la largeur minimale d'une décomposition arborescente S-connexe de H.*
*Soit φ une formule **CQ** (resp. $\mathbf{CQ}^{\neq}$). La largeur arborescente connexe de φ est la largeur arborescente S-connexe de l'hypergraphe associé à φ pour $S = free(\varphi)$.*

Exemple 83 *Soit $H = (V, E)$ l'hypergraphe donné par $V = \{x, y, z\}$, $E = \{\{x, y\}, \{y, z\}\}$ et, soit $S = \{x, z\}$. H possède la décomposition arborescente S-connexe (T, A) à deux noeuds reliés $B_1 = \{x, z\}$ et $B_2 = \{x, y, z\}$ (formant une décomposition arborescente de H) et $A = \{B_1\}$. On constate que $ctw(H, S) = 2$.*

Lemme 84 *Soient φ une σ-formule **CQ** (resp. $\mathbf{CQ}^{\neq}$), de largeur arborescente connexe k et une σ-structure $\mathcal{M}$ de domaine D. Alors on peut calculer en temps $O_\varphi(card(D)^{k+1})$ une σ'-formule **CCQ** (resp. $\mathbf{CCQ}^{\neq}$) φ' (de taille ne dépendant que de $|\varphi|$), $\sigma \subseteq \sigma'$, et une σ'-expansion $\mathcal{M}'$ telles que $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$.*

Preuve. Soit φ une formule **CQ** (resp. $\mathbf{CQ}^{\neq}$) de signature σ. Considérons (T, A) une décomposition arborescente S-connexe de φ de largeur k où S est l'ensemble des variables libres de φ. Soit la signature $\sigma' = \sigma \cup \{R_B : B \in V(T)\}$ où chaque R_B est un symbole de relation d'arité $\text{card}(B)$ et soit la formule φ' de signature σ' :

$$\varphi' = \varphi \wedge \bigwedge_{B \in V(T)} R_B(\bar{z_B})$$

où $\bar{z_B}$ désigne le tuple des variables du sac B. Considérons $\mathcal{M}'$ l'expansion de $\mathcal{M}$ de signature σ' où $R_B = D^{\text{card}(B)}$, pour tout $B \in V(T)$. Par définition d'une décomposition arborescente S-connexe, la formule φ' est connexe-acyclique. De plus, comme tout tuple de $D^{\text{card}(B)}$ est contenu dans la relation R_B, on a $\varphi(\mathcal{M}) = \varphi'(\mathcal{M}')$. Enfin le temps de calcul de $\mathcal{M}'$ est

$$O_\varphi(|\mathcal{M}| + \sum_{B \in V(T)} \text{card}(D)^{\text{card}(B)}) = O_\varphi(\text{card}(D)^{k+1})$$

□

Par le lemme 84 et le théorème 70, on obtient le résultat suivant :

Théorème 85 *Etant donnée une σ-formule φ de $\mathbf{CQ}^{\neq}$, de largeur arborescente connexe k, le problème* Enum(φ) *peut être calculé avec un précalcul en temps $O_\varphi(card(D)^{k+1})$ et à délai constant, pour une σ-structure $\mathcal{M}$ de domaine D.*

De même, en utilisant le lemme 84 et le théorème 58, on obtient le résultat suivant :

Théorème 86 *Etant donnée une σ-formule φ de **CQ**, de largeur arborescente connexe k, le problème* Count(φ) *peut être calculé en temps $O_\varphi(card(D)^{k+1})$, pour une σ-structure $\mathcal{M}$ de domaine D.*

2.8.2 Calcul de la largeur arborescente connexe

Nous donnons maintenant un algorithme polynomial pour calculer la largeur arborescente connexe d'une formule conjonctive. Nous devons résoudre le problème suivant.

CONNEX-TREE-DECOMPOSITION	
Entrée:	Un hypergraphe $H = (V, E)$ et un ensemble $S \subseteq V$
Paramètre:	un entier k
Sortie:	Une décomposition arborescente S-connexe de H de largeur k, si elle existe, et Faux sinon

Pour cela, nous introduisons la notion d'hypergraphe S-complété.

Définition 107 (hypergraphe S-complété) *Soit H un graphe. L'hypergraphe S-complété H_S de H est construit de la manière suivante :*

- $V(H_S) = V(H)$;
- $E(H_S) = E(H) \cup \{\{x, y\} : x, y \in S$ *et il existe un S-chemin de H entre x et $y\}$.*

Nous avons aussi besoin de quelques nouveaux résultats et définitions portant sur les décompositions arborescentes des hypergraphes.

Lemme 87 *Soient K une clique d'un hypergraphe $H = (V, E)$, T une décomposition arborescente de H et $\{B_1, B_2\}$ une arête de T. Soient T_1 et T_2 les deux sous-arbres de T séparés par l'arête $\{B_1, B_2\}$. On note $U_1 = \bigcup_{B \in V(T_1)} B$ et $U_2 = \bigcup_{B \in V(T_2)} B$. On a l'une des inclusions $K \subseteq U_1$ ou $K \subseteq U_2$.*

Preuve. On a évidemment $U_1 \cap U_2 = B_1 \cap B_2$ et $V = U_1 \cup U_2$. Supposons que l'on ait $K \nsubseteq U_1$ et $K \nsubseteq U_2$. Alors l'ensemble $E(H[K - (U_1 \cap U_2)])$ est formé de deux sortes d'hyperarêtes :

- des hyperarêtes incluses dans $K_1 = K \cap (U_1 - U_2) \neq \emptyset$,
- des hyperarêtes incluses dans $K_2 = K \cap (U_2 - U_1) \neq \emptyset$.

Donc aucune hyperarête ne relie les deux parties non vides et disjointes de K, K_1 et K_2, ce qui contredit l'hypothèse que K est une clique de H. Donc on a prouvé que $K \subseteq U_1$ ou $K \subseteq U_2$. □

Lemme 88 *Pour tout hypergraphe H, toute clique K de H et toute décomposition arborescente T de H, il existe un sac $B \in V(T)$ avec $K \subseteq B$.*

Preuve. Par induction sur le nombre de noeuds de T : c'est évident si $\text{card}(V(T)) = 1$. Supposons que $\text{card}(V(T)) \geq 2$: soit B_0 une feuille de T et soit T' l'arbre T privé de sa feuille B_0. Par le lemme précédent, on a $K \subseteq B_0$ ou $K \subseteq \bigcup_{B \in V(T')} B$. Dans le second cas, K est une clique de l'hypergraphe $H' = H[V']$ où $V' = \bigcup_{B \in V(T')} B$, de décomposition arborescente T'. Par hypothèse d'induction, il existe un sac $B \in V(T')$ avec $K \subseteq B$. Ce qui prouve le lemme par induction. □

Définition 108 (décomposition arborescente induite) *Soit T une décomposition arborescente d'un hypergraphe $H = (V, E)$ et soit S une partie de V. On note $T[S]$ l'arbre T dans lequel chaque noeud B est remplacé par $B \cap S$.*

La propriété de connexité étant réalisée sur T est aussi réalisée sur son arbre induit $T[S]$. Ce qui donne le lemme

Lemme 89 *Soit un hypergraphe $H = (V, E)$ et une partie S de V. Si T est une décomposition arborescente de H alors $T[S]$ est une décomposition arborescente de l'hypergraphe induit $H[S]$.*

On obtient un lemme similaire pour les sous-hypergraphes.

Lemme 90 *Soit T une décomposition arborescente d'un hypergraphe H et H' un sous-hypergraphe de H. L'arbre T' obtenu en remplaçant chaque noeud B de T par $B \cap V(H')$ est une décomposition arborescente de H'.*

Comme, dans les deux cas, la largeur de la décomposition arborescente ne peut que diminuer, on obtient le corollaire suivant :

Corollaire 91 *Pour tout hypergraphe induit $H[S]$ et tout sous-hypergraphe H' de H, on a $tw(H[S]) \leq tw(H)$ et $tw(H') \leq tw(H)$.*

Lemme 92 *Soit (T, A) une décomposition arborescente S-connexe d'un hypergraphe H. Si x et y sont deux sommets de H reliés par un S-chemin de H, $P = (x, a_1, \ldots, a_k, y)$, alors T admet un sac contenant x et y ; en conséquence, T est aussi une décomposition arborescente de l'hypergraphe S-complété H_S.*

Preuve. Soit T_x (resp. T_y) le sous-arbre de T formé des sacs contenant x (resp. y). Par hypothèse, A est l'ensemble des noeuds $V(T_S)$ d'un sous-arbre T_S de T avec $\bigcup_{B \in A} B = S$. Puisque $x, y \in S$, il existe deux noeuds de T_S, notés $B_{S,x}$ et $B_{S,y}$, tels que $x \in B_{S,x}$ et $y \in B_{S,y}$. A cause de P, la partie $C = \{a_1, \ldots, a_k\}$ de V est une partie connexe de H. Par conséquent, l'ensemble de noeuds $E_{\cap C} = \{B \in V(T) : B \cap C \neq \emptyset\}$ forme un sous-arbre $T_{\cap C}$ de T. L'hyperarête du chemin P, contenant x et a_1 (resp. a_k et y) est incluse dans un sac B_{x,a_1} (resp. $B_{a_k,y}$) de l'arbre T. Par définition, on a $B_{x,a_1} \in V(T_x) \cap V(T_{\cap C})$ et $B_{a_k,y} \in V(T_y) \cap V(T_{\cap C})$. Puisque $B_{S,x} \in V(T_S) \cap V(T_x)$ et $B_{x,a_1} \in V(T_x) \cap V(T_{\cap C})$, la réunion des arbres T_S, T_x et $T_{\cap C}$ est un sous-arbre de T. Puisque les arbres T_S et $T_{\cap C}$ sont disjoints, par construction, et contiennent respectivement les sacs $B_{S,y}$ et $B_{a_k,y}$, l'unique chemin reliant les noeuds $B_{S,y}$ et $B_{a_k,y}$ dans l'arbre T_y doit contenir au moins une arête de l'arbre T_x, donc au moins un noeud B de l'arbre T_x, et, en conséquence, $B \in V(T_x) \cap V(T_y)$. Autrement dit, le sac B de T contient à la fois x et y. □

Lemme 93 *Soient un hypergraphe $H = (V, E)$ et une partie $S \subseteq V$ tels que H n'admet pas de S-chemin. Soit $E_1, \ldots, E_l$ la liste des S-composantes de $E_{\not\subseteq S}$. Soient les ensembles $V_i = \bigcup_{e \in E_i} e$ et $e_i = V_i \cap S$, pour $i \in [l]$. On a les deux propriétés suivantes :*

1. *L'ensemble e_i est une clique de H ;*
2. *Si H admet une décomposition arborescente T de largeur $width(T) \leq k$, alors H admet aussi une décomposition arborescente S-connexe (T', A) de largeur $width(T') \leq k$.*

Preuve. 1) Par l'absurde : Supposons qu'il existe deux sommets $x, y \in e_i$ qui ne sont pas voisins dans H. Par définition de e_i, il existe deux hyperarêtes de E_i, e_x et e_y, qui contiennent respectivement x et y. Par définition de la S-composante E_i, dans l'hypergraphe induit $H[V-S]$, les hyperarêtes $e_x - S$ et $e_y - S$ sont les "extrémités" d'un "chemin" $(e_0, e_1, \ldots, e_k)$ avec $e_0 = e_x - S$, $e_k = e_y - S$ et $e_j \cap e_{j+1} \neq \emptyset$, pour tout $j < k$. Autrement dit, on a dans $H[V - S]$ un chemin $(a_1, a_2, \ldots, a_k)$ où chaque a_j est un élément choisi de $e_{j-1} \cap e_j$, donc dans $V - S$. Comme on a supposé que x et y ne sont pas voisins dans H, on constate donc que le chemin $(x, a_1, \ldots, a_k, y)$ est un S-chemin de H, ce qui est en contradiction avec l'hypothèse du lemme.

2) Par le corollaire 91 appliqué à l'hypergraphe induit $H[S]$, on a $tw(H[S]) \leq tw(H) = k$. Soit T_S une décomposition arborescente de $H[S]$ de largeur k. Puisque $e_i \subseteq S$ est une clique de H, c'est aussi une clique de $H[S]$. Donc, par le lemme 88, il existe un noeud $B_i \in V(T_S)$ avec $e_i \subseteq B_i$. On considère maintenant, pour chaque $i \in [l]$, l'hypergraphe H'_i défini par $H'_i = (V_i, E'_i)$ où E'_i est la réunion de E_i et de l'ensemble des hyperarêtes formant la clique e_i dans H et restreintes à V_i. Par le corollaire 91, on a $tw(H'_i) \leq tw(H) \leq k$. Puisque e_i est une clique de H'_i, toujours par le lemme 88, il existe un noeud $B'_i \in V(T_i)$ avec $e_i \subseteq B'_i$. Soit T' l'arbre obtenu par la réunion disjointe des arbres T_S et T_i, $i \in [l]$, en ajoutant l'arête $\{B_i, B'_i\}$ pour relier l'arbre T_S à l'arbre T_i. On observe les quatre points suivants.

a) T' vérifie la propriété de connexité car T_S et T_i la vérifient et l'on a $S \cap V_i = e_i \subseteq B_i \cap B'_i$.

b) T' est une décomposition arborescente de H, car, pour chaque $e \in E$, on a soit $e \subseteq S$, et donc il existe $B \in V(T_S)$ tel que $e \subseteq B$, soit $e \not\subseteq S$ et, en conséquence, il existe $i \in [l]$ tel que $e \in E_i$, et donc il existe $B \in V(T_i)$ tel que $e \subseteq B$: dans tous les cas, chaque hyperarête $e \in E$ est incluse dans un sac de T'.

c) Comme pour les arbres T_S et T_i, on a $\text{width}(T') \leq k$.

d) $\bigcup_{B \in V(T_S)} = S$.

On a donc établi que, pour $A = V(T_S)$, (T', A) est une décomposition arborescente S-connexe de H, de largeur k. □

Le théorème suivant fait la synthèse des résultats précédents.

Théorème 94 *1. Soit H un hypergraphe, soit une partie $S \subseteq V(H)$ et soit H_S l'hypergraphe S-complété de H. On a $ctw(H, S) = tw(H_S)$;*

2. *Le problème* Connex-Tree-Decomposition *est* **FPT** *et, plus précisément, est calculable en temps* $O_k(|E(H)| + |V(H)|^2)$ *et donc en temps* $O_k(|H|^2)$*, pour tout hypergraphe* H*.*

Preuve. Le point 1 découle des lemmes 93 et 92. Pour prouver le point 2, on remarquera que l'hypergraphe S-complété H_S de H est de taille $O_k(|E(H)| + |V(H)|^2)$ et se calcule en un temps du même ordre ; on utilise aussi le fait que calculer une décomposition arborescente de largeur k d'un hypergraphe si elle existe est un problème **FPT** et, plus précisément, est, pour H_S, de complexité $O_k(|E(H)| + |V(H)|^2)$ [Bod96]. □

Corollaire 95 *On peut reconnaître en temps* $O_k(|\varphi|^2)$ *si une formule conjonctive* φ *est de largeur arborescente connexe* k*.*

2.9 Conclusion

On peut énoncer plusieurs questions et conjectures en lien avec les résultats de ce chapitre.

2.9.1 Amélioration des constantes de complexité

Peut-on améliorer la borne $k!$ de la cardinalité d'un représentant, pour toute table $(E, \bar{f})$ avec $\bar{f} = (f_1, \ldots, f_k)$, et le temps pour calculer un tel représentant ? On conjecture qu'on peut toujours trouver un représentant de cardinalité $2^{O(k)}$ avec un temps de calcul en $2^{O(k)} \cdot \mathrm{card}(E)$. Un tel résultat améliorerait nos bornes en temps pour les requêtes $\mathbf{ACQ}^{\neq}$. Typiquement, le problème $\textsc{mc}(\mathbf{ACQ}^{\neq})$ se résoudrait alors en temps $2^{O(k)}|\varphi| \cdot |\mathcal{M}|$ au lieu de $2^{O(k \log k)}|\varphi| \cdot |\mathcal{M}|$. Le même genre de question se pose pour les requêtes $\mathbf{F\text{-}ACQ}^{all\neq}$ et la notion de représentant qui leur est associée. On conjecture que, pour toute table (E, f), avec $f : E \to \mathcal{P}_k(F)$, et tout entier l, il existe un l-représentant de (E, f) de cardinalité au plus $2^{O(k+l)}$ qu'on peut calculer en temps $2^{O(k+l)} \cdot \mathrm{card}(E)$. Un tel résultat permettrait d'obtenir un algorithme pour le problème de décision associé k-Path (existence d'un chemin de longueur k dans un graphe) de complexité en temps $2^{O(k)} \cdot |G|$, à comparer avec la borne $2^{O(k)}|G| \log |G|$ obtenue par Alon et al [AYZ95] pour le même problème par la méthode du "color coding".

2.9.2 Amélioration des théorèmes de classification

La question générale qui se pose est la suivante : quelles hypothèses de complexité simples et plausibles doit-on poser afin de rendre encore plus naturels (généraux) nos théorèmes de classification (dichotomie et trichotomie), ceci des deux manières suivantes :

1. en supprimant l'hypothèse que la formule est simple ;
2. dans le cas de la trichotomie, en supprimant l'hypothèse que l'hypergraphe est 4-conforme.

Chapitre 3

Evaluation de requêtes MSO sur des structures arborescentes

Sommaire

3.1 Introduction

Ce chapitre traite principalement de l'énumération des solutions des requêtes **MSO** dans les structures aborescentes, c'est-à-dire les structures de largeur arborescente bornée ou de largeur de clique bornée. On connaît le résultat classique de Courcelle [Cou90b, Cou92] complété par Flum, Frick et Grohe [FFG02] qui établit que ces requêtes se calculent en temps total linéaire. Notre contribution principale est d'étendre ce résultat au problème de l'énumération : nous prouvons que, pour toute formule **MSO** φ et toute classe de structures $\mathcal{S}$ de largeur arborescente bornée, on a $\textsc{Enum}(\varphi, \mathcal{S}) \in \textsc{Lin-Delay}_{lin}$. Autrement dit, pour toute structure $\mathcal{M} \in \mathcal{S}$, on peut calculer $\varphi(\mathcal{M})$ à délai $O(|S|)$, pour chaque solution $S \in \varphi(\mathcal{M})$, après un prétraitement en temps $O(|\mathcal{M}|)$.

Pour démontrer ce résultat, nous procédons en cinq étapes.

1) Une première étape logique : partant d'une requête **MSO** φ sur une structure $\mathcal{M}$ de largeur arborescente bornée, on construit, par une **MSO**-interprétation linéaire, une formule **MSO** φ' et un arbre étiqueté T vérifiant $\varphi(\mathcal{M}) = \varphi'(T)$.

2) Une simplification de la requête : on se ramène, par une réduction exacte, à une nouvelle formule **MSO** sur un arbre T ; cette formule est plus simple car elle n'a plus qu'une seule variable libre, variable ensembliste dont l'interprétation est un ensemble de feuilles de l'arbre.

3) Un passage à un automate d'arbre : il s'agit essentiellement du théorème de Thatcher et Wright [TW68] qui établit l'équivalence entre la définissabilité **MSO** d'une classe d'arbres étiquetés et sa reconnaissance par automate fini.

4) Une caractérisation efficace des solutions : on construit un système récursif d'équations qui possède les propriétés nécessaires à une énumération efficace de ses solutions.

5) Mise en oeuvre algorithmique : nous obtenons le schéma efficace d'énumération voulu à partir du système récursif d'équations construit précédemment et à l'aide de structures de données adaptées.

Notre contribution la plus originale se situe essentiellement dans les étapes 4 et 5 ci-dessus.

Enfin, nous étudions le problème du calcul de la j-ème solution pour les mêmes requêtes. Du système d'équations ensemblistes précédent, nous déduisons un système d'équations numériques qui portent sur les cardinalités des ensembles de solutions. Ce système d'équations nous permet de calculer, par recherche dichotomique, la j-ème solution cherchée en temps $O(|S| \log |\mathcal{M}|)$.

Il faut noter que les résultats de complexité obtenus dans ce chapitre ont aussi été obtenus, de manière indépendante par Bruno Courcelle [Cou06], à la différence suivante près : les

prétraitements de ses algorithmes ne sont pas linéaires mais utilisent un temps $O(|\mathcal{M}| \log |\mathcal{M}|)$.

3.2 Préliminaires

3.2.1 Σ-arbres

On s'intéresse dans ce chapitre à l'évaluation de formules **MSO** sur des arbres binaires localement complets. On donne d'abord une définition de ce qu'on entend ici par arbre.

Définition 109 (Σ-arbre) *Soit Σ un alphabet fini. Un Σ-arbre est un quadruplet $T = (Tree(T), D = Dom(T), root(T), label_T)$ vérifiant les conditions suivantes :*

- *$Tree(T)$ est un arbre binaire enraciné non vide (arbre sous-jacent à T), $D = Dom(T)$ est son ensemble de noeuds (encore appelé domaine de T) et $root(T)$ est sa racine ;*
- *chaque noeud qui n'est pas une feuille possède exactement deux enfants, son enfant gauche et son enfant droit ;*
- *$label_T$ est une fonction de D dans Σ appelée étiquetage de T.*

On notera $Leaves(T)$ l'ensemble des feuilles de T.

On peut aussi associer à un Σ-arbre T une structure $\mathcal{M}_T = (D, f_1, f_2, (\text{label}_\alpha)_{\alpha\in\Sigma})$ de domaine D et de signature $\tau_\Sigma = \{f_1, f_2, (\text{label}_\alpha)_{\alpha\in\Sigma}\}$ qu'on définit comme suit :

- Pour $i = 1$ (resp. $i = 2$) et pour tous $x, y \in D$, on a $\mathcal{M}_T \models f_i(x, y)$ si et seulement si y est l'enfant gauche (resp. droit) de x ;
- Pour tous $\alpha \in \Sigma$ et $x \in D$, on a $\mathcal{M}_T \models \text{label}_\alpha(x)$ si et seulement si $\text{label}_T(x) = \alpha$.

Par abus de langage, on confondra souvent le Σ-arbre T, encore appelé arbre T, avec sa τ_Σ-structure associée $\mathcal{M}_T$.

3.2.2 Automates d'arbre

Les automates d'arbre sont une généralisation des automates de mot. On donne ici les définitions classiques pour les automates d'arbre déterministes dits "bottom-up". Pour plus de détails voir [CDG+97, Lib04].

Définition 110 (automate d'arbre) *Un automate de Σ-arbre est un tuple $\Gamma = (\Sigma, Q, \delta, \delta_0, F)$ où*

- *Q est un ensemble fini d'états ;*
- *δ est une fonction de $\Sigma \times Q^2$ dans Q, appelée fonction de transition ;*
- *δ_0 est une fonction de Σ dans Q, appelée fonction initiale de transition ;*
- *F est une partie de Q, appelée ensemble des états acceptants.*

Définition 111 (calcul d'un automate, calcul acceptant) *Soit un Σ-arbre T et un automate $\Gamma = (\Sigma, Q, \delta, \delta_0, F)$. Le calcul de Γ sur T est la fonction, notée Γ_T, de $Dom(T)$ dans Q telle que*

- *pour toute feuille x de T, $\Gamma_T(x) = \delta_0(label_T(x))$;*
- *pour tout noeud interne x, ayant pour enfants respectifs gauche et droit y et z, $\Gamma_T(x) = \delta(label_T(x), \Gamma_T(y), \Gamma_T(z))$.*

Le calcul Γ_T est acceptant si $\Gamma_T(root(T)) \in F$ (la racine de l'arbre est "acceptée").

Définition 112 (arbre accepté, langage reconnu par un automate) *Un automate Γ accepte l'arbre T si le calcul Γ_T est acceptant.*
Le langage (d'arbres) reconnu par l'automate Γ est l'ensemble des arbres T tels que Γ accepte T.

Remarque 25 *Dans cette définition, on appelle arbre un Σ-arbre. On fera souvent, par la suite, cet abus de langage quand aucune ambiguité n'est possible.*

Le résultat de caractérisation suivant dû à Thatcher et Wright est fondamental.

Théorème 96 *[TW68] Soit $\mathcal{T}$ un ensemble de Σ-arbres. Alors les deux assertions suivantes sont équivalentes :*

- *$\mathcal{T}$ est défini par un τ_Σ-énoncé* **MSO** *φ ;*
- *$\mathcal{T}$ est reconnu par un automate de Σ-arbre Γ.*

De plus, le passage d'un énoncé **MSO** à un automate d'arbre est effectif : il existe une fonction f et un algorithme qui, étant donné un énoncé **MSO** φ, renvoie un automate d'arbre qui reconnaît le même langage d'arbres que φ, ceci en temps $f(|\varphi|)$. La complexité de cette fonction f est non élémentaire : plus précisément, Reinhardt [Rei01] montre qu'il existe une famille infinie $\mathcal{C}$ d'énoncés **MSO** et une fonction f non élémentaire telles que, pour tout énoncé $\varphi \in \mathcal{C}$, tout automate d'arbre équivalent à φ (au sens du Théorème 96) est de taille supérieure à $f(|\varphi|)$.

3.3 Caractérisation de l'ensemble des solutions d'une requête

On voudrait ici évaluer et énumérer le plus efficacement possible une requête $\varphi(T)$ pour une formule **MSO** φ (avec des variables libres) sur un Σ-arbre T.

3.3.1 Première simplification : se ramener à des formules sans variable du premier ordre

Sans perte de généralité, on peut restreindre le problème aux requêtes sans variable du premier ordre. En effet, il suffit de considérer une variable du premier ordre comme une variable du second

ordre dont on impose qu'elle soit un singleton. Plus précisément, pour une formule $\varphi(\bar{X}, y_1, \dots y_l)$, on associe la formule suivante :

$$\begin{array}{rcl} \varphi'(\bar{X}, Y_1, \dots Y_l) & \equiv & \bigwedge_{i\in[l]} \exists y \forall z (Y_i(z) \leftrightarrow z = y) \\ & & \wedge \forall y_1 \dots \forall y_l (\bigwedge_{i\in[l]} Y_i(y_i) \rightarrow \varphi(\bar{X}, y_1, \dots, y_l)) \end{array}$$

Clairement, $(\mathcal{M}, A_1, \dots A_k, B_1, \dots, B_l) \models \varphi'(\bar{X}, \bar{Y})$ si et seulement si chaque B_i est un singleton tel que $B_i = \{b_i\}$ et $(\mathcal{M}, A_1, \dots A_k, b_1, \dots, b_l) \models \varphi(\bar{X}, \bar{y})$.

3.3.2 Currification de l'arbre

Par souci de simplicité, on se restreint aux Σ-arbres dont les noeuds internes sont étiquetés par une couleur factice @ et aux formules où l'interprétation des variables du second ordre est restreinte aux ensembles de feuilles. Le prochain lemme explique comment réduire le cas général à ces conditions.

Lemme 97 *Soit Σ un alphabet fini. Soit $\Sigma_@ = \Sigma \cup \{@\}$ son extension avec le nouveau symbole @. Il existe une transformation $\varphi \mapsto \varphi'$ associant à une τ_Σ-formule* **MSO** *une $\tau_{\Sigma_@}$-formule* **MSO** *et une transformation $T \mapsto T'$ d'un Σ-arbre T en un $\Sigma_@$-arbre T' vérifiant les conditions suivantes :*

1. *la transformation $\varphi \mapsto \varphi'$ (resp. $T \mapsto T'$) est calculable en temps $O(|\varphi|)$ (resp. $O(|T|)$) ;*
2. *l'ensemble des feuilles de T' est l'ensemble des noeuds de T avec les mêmes étiquettes ;*
3. *les noeuds internes de T' sont étiquetés par @ ;*
4. *$\varphi(T) = \varphi'(T')$.*

Preuve. Transformation de l'arbre : Pour chaque Σ-arbre T, on construit le $\Sigma_@$-arbre T', encore appelé arbre currifié de T, de la manière suivante :

- $\mathrm{Dom}(T') = \mathrm{Dom}(T) \cup \{(a, i) : a$ est un noeud interne de T et $1 \leq i \leq 2\}$;
- pour chaque noeud interne a de T, on fait $T' \models f_1((a,1), a) \wedge f_2((a,1), (a,2))$;
- pour chaque enfant b de a dans T, $T \models f_i(a, b)$, $i = 1, 2$, on fait $T' \models f_i((a,2), b)$ si b est une feuille de T et $T' \models f_i((a,2), (b,1))$ si b est un noeud interne de T ;
- enfin, on fait $\mathrm{label}_{T'}(a) = \mathrm{label}_T(a)$, pour chaque noeud a de T (feuille de T'), et $\mathrm{label}_{T'}(a) =$ @, pour chaque autre noeud (interne) a de T'.

Il est clair que la transformation $T \mapsto T'$ est calculable en temps $O(|T|)$ et que l'arbre T' satisfait les conditions (2) et (3).

Transformation de la formule : Comme d'habitude, on procède par interprétation. Pour cela, on introduit les $\tau_{\Sigma_@}$-formules suivantes :

- $\Delta(x) \equiv \neg \mathrm{label}_@(x)$;

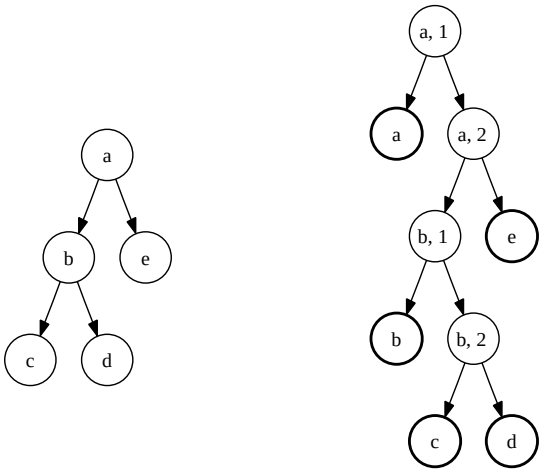


FIG. 3.1 – Un arbre T et son arbre currifié T'

– Pour $i = 1, 2$, $\Theta_{f_i}(x, y) \equiv \exists x_1 \exists x_2 \exists y_1 (f_1(x_1, x) \wedge f_2(x_1, x_2) \wedge f_i(x_2, y_1) \wedge ((\Delta(y_1) \wedge y = y_1) \vee (\neg\Delta(y_1) \wedge f_1(y_1, y))))$.

Il est facile de vérifier les équivalences suivantes, pour tout Σ-arbre T et son $\Sigma_@$-arbre currifié T' :

– Pour tout $a \in \mathrm{Dom}(T')$, on a $a \in \mathrm{Dom}(T)$ si et seulement si $(T', a) \models \Delta(x)$;

– pour tous $a, b \in \mathrm{Dom}(T)$ et tout $i = 1, 2$, on a $(T, a, b) \models f_i(x, y)$ si et seulement si $(T', a, b) \models \Theta_{f_i}(x, y)$.

□

Par abus de langage, on appelle encore $\Sigma_@$-arbre un $(\Sigma \cup \{@\})$-arbre dont l'étiquette de chaque noeud interne est @ et l'étiquette de chaque feuille appartient à Σ. Comme les noeuds internes sont étiquetés par @, on utilisera la notation $q_1, q_2 \to q$ qui signifie $\delta(@, q_1, q_2) = q$ pour la fonction de transition δ de l'automate Γ.

3.3.3 Une dernière simplification : se ramener à des formules à une seule variable libre

Pour simplifier les traitements qui vont suivre et, tout particulièrement, les notations, on a justifié trois sortes de restrictions dans les requêtes **MSO** sur les arbres :

– toutes les variables libres de la formule sont des variables du second ordre ;

– les noeuds internes des arbres ne portent pas d'information : ils ont une étiquette factice @ ;

– les variables libres (toutes du second ordre) sont contraintes à prendre leurs valeurs dans les ensembles de feuilles ; une telle requête est donc de la forme :

$$\{(S_1, \ldots, S_k) \in (\mathcal{P}(\mathrm{Leaves}(T)))^k : (T, \bar{S}) \models \varphi(\bar{X})\}$$

On va maintenant ajouter et justifier une dernière restriction : se ramener à une formule à une seule variable libre (du second ordre). Pour simplifier, on explique comment opérer cette transformation pour une $\tau_{\Sigma_@}$-formule **MSO** $\varphi(X_1, X_2)$ à deux variables libres (la généralisation à k variables libres étant facile mais fastidieuse à présenter).

Transformation de l'arbre

On transforme chaque $\Sigma_@$-arbre T en un nouveau $\Sigma_@$-arbre T' en dupliquant chaque feuille de T. Plus précisément, on remplace chaque feuille t de T par le sous-arbre de profondeur 1, de racine t, avec pour enfants gauche et droit, les deux nouvelles feuilles $(t,1)$ et $(t,2)$ et avec les nouvelles étiquettes $\text{label}_{T'}(t) = @$ et $\text{label}_{T'}((t,i)) = \text{label}_T(t)$ pour $i = 1, 2$.

Cette duplication des feuilles de T réalisée dans T' va nous permettre de représenter tout couple (S_1, S_2) d'ensembles de feuilles de T par un ensemble S de feuilles de T' que l'on définit par l'équation (*) suivante :

$$S = \{(t,1) : t \in S_1\} \cup \{(t,2) : t \in S_2\}$$

Notons que sa taille est $|S| = \Theta(|S_1| + |S_2|)$ et que la transformation réciproque $S \mapsto (S_1, S_2)$ est aussi calculable en temps $O(|S|)$.

Transformation de la formule

De même que l'on a transformé tout $\Sigma_@$-arbre T en un $\Sigma_@$-arbre T' de taille $|T'| = O(|T|)$, de même il est facile de transformer (traduire) une $\tau_{\Sigma_@}$-formule $\varphi(X_1, X_2)$ en une $\tau_{\Sigma_@}$-formule $\varphi'(X)$ à une seule variable libre de façon à avoir l'équivalence suivante, pour tous ensembles $S_1, S_2 \subseteq \text{Leaves}(T)$ et pour l'ensemble $S \subseteq \text{Leaves}(T')$ défini par l'équation (*) ci-dessus : $(T, S_1, S_2) \models \varphi(X_1, X_2)$ si et seulement si $(T', S) \models \varphi'(X)$.

3.3.4 Automate associé à une requête MSO à une variable libre

Le théorème 96 ne s'applique qu'aux énoncés (formules sans variable libre) de **MSO**. Pour l'adapter aux formules **MSO** à une variable libre, auxquelles on vient de se ramener, on considère l'alphabet enrichi $\Sigma'_@ = (\Sigma \times \{0,1\}) \cup \{@\}$. A tout $\Sigma_@$-arbre T et à tout ensemble $S \subseteq \text{Leaves}(T)$, on associe le $\Sigma'_@$-arbre T^S identique à T excepté en ce qui concerne sa fonction étiquetage qu'on définit comme suit : pour tout noeud interne t, $\text{label}_{T^S}(t) = \text{label}_T(t) = @$, et, pour toute feuille $t \in \text{Leaves}(T)$, on prend $\text{label}_{T^S}(t) = (\text{label}_T(t), \text{car}_S(t))$, où car_S désigne la fonction caractéristique de l'ensemble S : $\text{car}_S(t) = 1$ si $t \in S$ et $\text{car}_S(t) = 0$ si $t \notin S$.

Lemme 98 *Soit Σ un alphabet fini. A toute $\tau_{\Sigma_@}$-formule* **MSO** *à 1 variable $\varphi(X)$, on peut associer un automate Γ de $\Sigma'_@$-arbre avec $\Sigma'_@ = (\Sigma \times \{0,1\}) \cup \{@\}$ tel que, pour tout $\Sigma_@$-arbre*

T et tout ensemble $S \subseteq Leaves(T)$, on ait l'équivalence : $(T,S) \models \varphi(X)$ si et seulement si Γ accepte l'arbre T^S.

Preuve. Il est facile (et on laisse le soin de le faire au lecteur) de construire un $\tau_{\Sigma'_@}$-énoncé **MSO** φ' tel que, pour tout $\Sigma_@$-arbre T et tout ensemble $S \subseteq \text{Leaves}(T)$, on ait l'équivalence : $(T,S) \models \varphi(X)$ si et seulement si $T^S \models \varphi'$. Le résultat découle alors du théorème de caractérisation 96. □

3.3.5 Caractérisation des ensembles acceptés par un automate

L'objectif est maintenant de caractériser les ensembles $S \subseteq \text{Leaves}(T)$ tels que l'automate Γ accepte l'arbre T^S. Pour cela, on aura besoin des définitions, notations et remarques suivantes.

Définition 113 (réunion disjointe, produit d'ensembles de parties) *Si S et S' sont deux ensembles disjoints, alors $S \uplus S'$ désigne leur réunion (disjointe).*

Si on a deux ensembles de parties $P_1 \subseteq \mathcal{P}(E_1)$ et $P_2 \subseteq \mathcal{P}(E_2)$ de deux ensembles disjoints E_1 et E_2, on définit le produit $P_1 \otimes P_2 = \{S_1 \uplus S_2 : S_1 \in P_1 \text{ et } S_2 \in P_2\}$.

Définition 114 (sous-arbre, restriction d'un ensemble à ce sous-arbre) *Pour un arbre T et un noeud $x \in Dom(T)$, on note T_x le sous-arbre de T de racine x.*

Pour un ensemble $S \subseteq Leaves(T)$, on note S_x sa restriction au sous-arbre T_x : $S_x = S \cap Leaves(T_x)$. Si x est un noeud interne d'enfants y_1 et y_2 dans T, on a clairement $S_x = S_{y_1} \uplus S_{y_2}$.

Soit T un $\Sigma_@$-arbre ($\Sigma_@ = \Sigma \cup \{@\}$) et soit $S \subseteq \text{Leaves}(T)$. Rappelons qu'on note T^S le $\Sigma'_@$-arbre (où $\Sigma'_@ = \Sigma \times \{0,1\} \cup \{@\}$) obtenu en complétant les étiquettes de T avec la fonction caractéristique de S : pour chaque $x \in \text{Leaves}(T)$, $\text{label}_{T^S}(x) = (\text{label}_T(x), \text{car}_S(x))$.

Définition 115 (configuration) *Soit $\Gamma = (\Sigma, Q, \delta, \delta_0, F)$ un automate de $\Sigma'_@$-arbre et soit un $\Sigma_@$-arbre T. On appelle configuration de Γ sur T, ou simplement configuration, un couple $(x,q) \in Dom(T) \times Q$. On note $Conf = Dom(T) \times Q$.*

Pour une partie $S \subseteq \text{Leaves}(T)$ et un noeud $x \in \text{Dom}(T)$, on note $q = \Gamma_{T^S}(x)$ l'état $q \in Q$ de x dans le calcul de Γ sur l'arbre T^S. Il est clair que cet état q (et donc la configuration (x,q)) ne dépend que de la restriction S_x de S à l'arbre T_x. Pour cette raison, on utilisera la notation suggestive $S_x \overset{\Gamma,T_x}{\longmapsto} q$ et on définit l'ensemble

$$S(x,q) = \{S \subseteq \text{Leaves}(T_x) : S \overset{\Gamma,T_x}{\longmapsto} q\}.$$

On cherche à calculer l'ensemble, noté S_{accept}, des parties $S \subseteq \text{Leaves}(T)$ telles que $S \overset{\Gamma,T}{\longmapsto} q$, pour un état $q \in F$. Cela peut être fait de manière récursive, ce qu'exprime les deux propriétés suivantes, faciles à vérifier.

– Propriété (a) : Pour tout noeud interne x de T ayant pour enfants y_1 et y_2, on a

$$S(x,q) = \biguplus_{\substack{(q_1,q_2)\in Q^2 \\ q_1,q_2\to q}} S(y_1,q_1)\otimes S(y_2,q_2)$$

– Propriété (b) :

$$S_{accept} = \biguplus_{q\in F} S(\mathrm{root}(T),q)$$

Remarque essentielle : Dans les deux cas, les unions sont disjointes du fait du déterminisme de l'automate Γ. Plus précisément, pour deux états q et q' distincts, on a toujours $S(x,q)\cap S(x,q') = \emptyset$.

A première vue, les propriétés (a) et (b) vues ci-dessus semblent donner le principe d'un algorithme efficace pour énumérer les ensembles S tels que Γ accepte T^S. Malheureusement, cela ne suffit pas pour obtenir un temps $O(|S|)$ pour calculer chaque solution S. En effet, un tel algorithme peut perdre du temps à cause des ensembles vides rencontrés. Plus précisément, dans l'énumération des éléments S d'un produit $P_1 \otimes P_2$ (où les opérandes sont de la forme $P_i = S(y_i,q_i), i = 1,2$), on perd du temps dans le cas où $\emptyset \in P_1$ ou $\emptyset \in P_2$ ou dans le cas où $P_1 = \emptyset$ ou $P_2 = \emptyset$. Pour surmonter cette difficulté, on exclut le cas de la solution éventuelle $S = \emptyset$ qu'on traitera à part et on fait en sorte que chaque ensemble considéré ne soit pas vide et n'ait pas comme élément l'ensemble vide. On s'intéressera donc exclusivement aux configurations (x,q) pour lesquelles l'ensemble suivant :

$$S^*(x,q) = \{S \subseteq \mathrm{Leaves}(T_x) : S \neq \emptyset \text{ et } S \overset{\Gamma,T_x}{\longmapsto} q\}$$

n'est pas vide.

Définition 116 (état vide, configuration vide) *On appelle état vide d'un noeud $x \in Dom(T)$ l'unique état $q_\emptyset \in Q$ tel que $\emptyset \overset{\Gamma,T_x}{\longmapsto} q_\emptyset$. La configuration $(x,q_\emptyset)$ est appelée configuration vide de x et on note $Conf^{vide} = \{(x,q_\emptyset) \in Conf : \emptyset \overset{\Gamma,T_x}{\longmapsto} q_\emptyset\}$.*

Définition 117 (transition vide) *Soient (x,q), (y_1,q_1) et (y_2,q_2) trois configurations telles que y_1 et y_2 soient les enfants gauche et droit de x dans T et $q_1,q_2 \to q$. On écrit $(x_1,q_1) \overset{\emptyset}{\longmapsto} (x,q)$ (resp. $(x_2,q_2) \overset{\emptyset}{\longmapsto} (x,q)$) si (x_2,q_2) (resp. (x_1,q_1)) est la configuration vide de x_2 (resp. x_1).*

On désigne par $\overset{*}{\longmapsto}$ la fermeture réflexive et transitive de la relation $\overset{\emptyset}{\longmapsto}$.

Remarque 26 *Il est évident que si on a $(x',q') \overset{\emptyset}{\longmapsto} (x,q)$, alors tout ensemble $S \in S^*(x',q')$ appartient aussi à $S^*(x,q)$, c'est-à-dire $S^*(x',q') \subseteq S^*(x,q)$. Plus généralement, on obtient la même inclusion par induction si on a $(x',q') \overset{*}{\longmapsto} (x,q)$.*

Dans notre calcul récursif des ensembles $S^*(x,q)$, on aura besoin d'utiliser les ensembles intermédiaires $S^{useful}(x,q)$ ainsi définis :

– Si x est une feuille de T alors

$$S^{useful}(x,q) = \begin{cases} \{\{x\}\} & \text{si } \delta_0((\text{label}_T(x),1)) = q \\ \emptyset & \text{sinon} \end{cases}$$

– Si x est un noeud interne dont les enfants sont y_1 et y_2 dans T alors

$$S^{useful}(x,q) = \{S \subseteq \text{Leaves}(T_x) : S_{y_1} \neq \emptyset, S_{y_2} \neq \emptyset \text{ et } S \overset{\Gamma,T_x}{\longmapsto} q\}$$

Le lemme suivant permet de définir les ensembles $S^{useful}(x,q)$ et $S^*(x,q)$ par récursivité indirecte.

Lemme 99 *Soit (x,q) une configuration de Γ sur T.*

1. *Si x est une feuille alors*

$$S^{useful}(x,q) = \begin{cases} \{\{x\}\} & \textit{si } \delta_0((label_T(x),1)) = q \\ \emptyset & \textit{sinon} \end{cases}$$

2. *Si x est un noeud interne ayant pour enfants y_1 et y_2 alors*

$$S^{useful}(x,q) = \biguplus_{\substack{(q_1,q_2)\in Q^2 \\ q_1,q_2 \to q}} S^*(y_1,q_1) \otimes S^*(y_2,q_2)$$

3. *On a toujours :*

$$S^*(x,q) = \biguplus_{\substack{(x',q')\in Conf \\ (x',q') \overset{*}{\longmapsto} (x,q)}} S^{useful}(x',q')$$

Preuve.

1) C'est la définition.

2) On a les égalités successives :

$$\begin{aligned} S^{useful}(x,q) &= \{S \subseteq \text{Leaves}(T_x) : S_{y_1} \neq \emptyset, S_{y_2} \neq \emptyset, S \overset{\Gamma,T_x}{\longmapsto} q\} \\ &= \biguplus_{\substack{(q_1,q_2)\in Q^2 \\ q_1,q_2 \to q}} \{S \subseteq \text{Leaves}(T_x) : S_{y_1} \neq \emptyset, S_{y_2} \neq \emptyset, S_{y_1} \overset{\Gamma,T_{y_1}}{\longmapsto} q_1, S_{y_2} \overset{\Gamma,T_{y_2}}{\longmapsto} q_2\} \end{aligned}$$

qu'on peut reformuler en

$$S^{useful}(x,q) = \biguplus_{\substack{(q_1,q_2)\in Q^2 \\ q_1,q_2\to q}} \{S_1 \uplus S_2 : S_1 \subseteq \text{Leaves}(T_{y_1}), S_2 \subseteq \text{Leaves}(T_{y_2}), S_1 \neq \emptyset, S_2 \neq \emptyset, S_1 \overset{\Gamma,T_{y_1}}{\longmapsto} q_1, S_2 \overset{\Gamma,T_{y_2}}{\longmapsto} q_2\}$$

$$= \biguplus_{\substack{(q_1,q_2)\in Q^2 \\ q_1,q_2\to q}} S^*(y_1,q_1) \otimes S^*(y_2,q_2)$$

3) On établit l'égalité des deux ensembles en prouvant leurs inclusions réciproques.
$\supseteq$: Supposons qu'on a $(x',q') \overset{*}{\longmapsto} (x,q)$. On a les inclusions $S^{useful}(x',q') \subseteq S^*(x',q') \subseteq S^*(x,q)$.
$\subseteq$: Soit un ensemble $S \in S^*(x,q)$. Soit x' le plus petit ancêtre commun de tous les sommets de S dans l'arbre T et soit (x',q') la configuration de x' dans le calcul de Γ sur T^S. Par définition, on a $(x',q') \overset{*}{\longmapsto} (x,q)$ et $S \in S^*(x',q')$. Deux cas se présentent :

- soit x' est une feuille de T et $S = \{x'\} \in S^{useful}(x',q')$;
- soit x' est un noeud interne dont les enfants sont y_1 et y_2 dans T : par définition de x', aucune des deux configurations (y_1,q_1) et (y_2,q_2) qui sont enfants de (x',q') n'est une configuration vide. On a donc $S \cap \text{Leaves}(T_{y_1}) \neq \emptyset$ et $S \cap \text{Leaves}(T_{y_2}) \neq \emptyset$. Autrement dit, on a $S = S_1 \uplus S_2$ pour $S_1 \subseteq \text{Leaves}(T_{y_1})$ et $S_2 \subseteq \text{Leaves}(T_{y_2})$ avec $S_1 \neq \emptyset$ et $S_2 \neq \emptyset$. Donc $S \in S^*(y_1,q_1) \otimes S^*(y_2,q_2)$ avec $(q_1,q_2) \to q$. Par conséquent, on obtient $S \in S^{useful}(x',q')$ par la clause (2) (prouvée) du lemme.

On a ainsi prouvé la double inclusion annoncée et donc l'équation (3).

Il reste à vérifier que les ensembles impliqués dans la réunion donnée dans cette équation (3) sont deux à deux disjoints. Soient deux configurations distinctes (y_1,q_1) et (y_2,q_2) telles que $(y_1,q_1) \overset{*}{\longmapsto} (x,q)$ et $(y_2,q_2) \overset{*}{\longmapsto} (x,q)$. Si $y_1 = y_2$ et $q_1 \neq q_2$, les ensembles $S^{useful}(y_1,q_1)$ et $S^{useful}(y_2,q_2)$ sont disjoints à cause du déterminisme de Γ. Si y_1 et y_2 sont indépendants dans T, c'est-à-dire ne sont ni descendants ni ancêtres l'un de l'autre, alors les ensembles $S^{useful}(y_1,q_1) \subseteq \mathcal{P}(\text{Leaves}(T_{y_1}))$ et $S^{useful}(y_2,q_2) \subseteq \mathcal{P}(\text{Leaves}(T_{y_2}))$ sont trivialement disjoints. Supposons enfin que y_2 est descendant strict de y_1 (cas symétrique : y_2 est ancêtre strict de y_1). Plus précisément, supposons que y_1 a deux enfants z_1 et z_2 et que y_2 est descendant (au sens large) de z_1 (le cas où y_2 est descendant de z_2 est symétrique). Soit $S \in S^{useful}(y_1,q_1)$. Par définition, on a $S \cap \text{Leaves}(T_{z_i}) \neq \emptyset$ pour tout $i = 1,2$. Puisque, par définition, les ensembles $\text{Leaves}(T_{z_2})$ et $\text{Leaves}(T_{y_2})$ sont disjoints (car z_2 et y_2 sont indépendants dans T) et que l'ensemble $S^{useful}(y_2,q_2)$ est inclus dans $\text{Leaves}(T_{y_2})$, on en conclut que $S \notin S^{useful}(y_2,q_2)$. On a donc établi que, dans l'équation (3), la réunion est bien une réunion disjointe, ce qui achève la preuve du lemme. □

3.3.6 Encore un raffinement de la caractérisation

Les équations (2) et (3) du lemme précédent ne sont pas encore suffisantes pour obtenir l'efficacité attendue dans l'énumération. Il se peut que certains ensembles qui interviennent comme opérandes dans les unions disjointes des équations (2) et (3) soient vides. Pour éviter d'examiner ces ensembles vides, on introduit deux nouvelles notions : les *paires utiles* et les *configurations utiles*.

Définition 118 (paire utile) *Soit x un noeud interne dont les enfants sont y_1 et y_2 dans T et soit (x, q) une configuration de x. On appelle* paire utile pour la configuration (x, q) *tout couple d'états $(q_1, q_2) \in Q^2$ tel que l'on a $q_1, q_2 \to q$ et tel que les deux ensembles $S^*(y_1, q_1)$ et $S^*(y_2, q_2)$ sont non vides ; on définit l'ensemble des paires utiles :*

$$Pair^{useful}(x, q) = \{(q_1, q_2) \in Q^2 : q_1, q_2 \to q \text{ et } S^*(y_1, q_1) \neq \emptyset \text{ et } S^*(y_2, q_2) \neq \emptyset\}$$

Définition 119 (configuration utile) *Une configuration (x, q) est dite utile si l'ensemble $S^{useful}(x, q)$ n'est pas vide ; on note l'ensemble des configurations utiles :*

$$Conf^{useful} = \{(x, q) \in Conf : S^{useful}(x, q) \neq \emptyset\}$$

En se restreignant aux objets (paires et configurations) qui sont "utiles", on obtient les équations suivantes, variantes des équations (2) et (3) ci-dessus, et dont chaque opérande de réunion disjointe n'est pas vide.

– Si x est un noeud interne ayant pour enfants y_1 et y_2, alors

$$S^{useful}(x, q) = \biguplus_{(q_1,q_2)\in \mathrm{Pair}^{useful}(x,q)} S^*(y_1, q_1) \otimes S^*(y_2, q_2) \qquad (2')$$

– Pour tout noeud x, on a

$$S^*(x, q) = \biguplus_{\substack{(x',q')\in \mathrm{Conf}^{useful} \\ (x',q') \xrightarrow{*} (x,q)}} S^{useful}(x', q') \qquad (3')$$

Toujours pour des raisons d'efficacité, on a besoin de définir les configurations (x, q) pour lesquelles $S^*(x, q)$ n'est pas vide.

Définition 120 (configuration possible) *Une configuration (x, q) est possible si $S^*(x, q)$ n'est pas vide et on note*

$$Conf^{possible} = \{(x, q) \in Conf : S^*(x, q) \neq \emptyset\}$$

Ainsi, pour une configuration (x, q) donnée quelconque, l'ensemble des configurations (x', q') considérées dans l'équation (3') (et donc la famille d'ensembles dont on fait l'union disjointe) n'est pas vide si et seulement si (x, q) est une configuration possible.

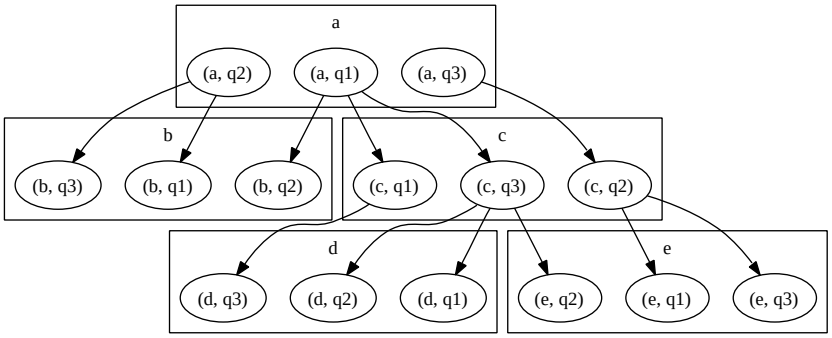


FIG. 3.2 – Un exemple de forêt des transitions vides

3.3.7 Forêt des transitions vides

Pour chaque configuration possible (x, q), on veut calculer efficacement l'ensemble (ensemble des indices de la réunion $\biguplus$ de l'équation (3')) :

$$\mathrm{Conf}^{useful}(x,q) = \{(x',q') \in \mathrm{Conf}^{useful} : (x',q') \overset{*}{\longmapsto} (x,q)\}$$

Pour cela, on définit le graphe orienté, noté $F^{\emptyset}$, qui représente le graphe de transition des configurations (de Γ sur T) dont l'une des opérandes est une configuration vide, graphe encore appelé graphe des *transitions vides*. Plus précisément, l'ensemble des sommets de $F^{\emptyset}$ est l'ensemble Conf des configurations $C = (x, q)$ de Γ sur T et son ensemble d'arcs est $\{(C_1, C_2) \in \mathrm{Conf}^2 : C_2 \overset{\emptyset}{\longmapsto} C_1\}$.

Par construction, le graphe $F^{\emptyset}$ est sans cycle et, puisque Γ est déterministe, c'est une forêt de même profondeur que T et dont l'ensemble des racines est l'ensemble $\{\mathrm{root}(T)\} \times Q$. On l'appellera la *forêt des transitions vides* de Γ sur T.

Exemple 100 *La figure 3.2 représente la forêt des transitions vides d'un automate Γ sur l'arbre T d'ensemble de noeuds $Dom(T) = \{a, b, c, d, e\}$, arbre qui est donné de façon implicite par les rectangles et les flèches.*

On constate que, par définition de la forêt $F^{\emptyset}$ et de la relation $\overset{*}{\longmapsto}$, l'ensemble des configurations (x', q') qui vérifient $(x', q') \overset{*}{\longmapsto} (x, q)$ est l'ensemble des noeuds du sous-arbre, noté $F^{\emptyset}_{(x,q)}$, de racine (x, q) de la forêt $F^{\emptyset}$, d'où on tire

$$\mathrm{Conf}^{useful}(x,q) = \mathrm{Conf}^{useful} \cap V(F^{\emptyset}_{(x,q)})$$

Notons $\leq_{prefix}$ un ordre préfixe des noeuds de la forêt $F^{\emptyset}$. Il est facile de numéroter les configurations utiles dans cet ordre préfixe ; on les notera

$$\text{Conf}^{useful}[0] <_{prefix} \text{Conf}^{useful}[1] <_{prefix} \ldots <_{prefix} \text{Conf}^{useful}[u-1]$$

où $u = \text{card}(\text{Conf}^{useful})$.

Comme dans tout ordre préfixe d'une forêt, l'ensemble des noeuds de chaque sous-arbre de la forêt $F^{\emptyset}$ forme un intervalle de l'ordre préfixe. On en déduit que, pour chaque configuration possible (x, q), donc telle que l'ensemble $\text{Conf}^{useful}(x, q)$ n'est pas vide, cet ensemble correspond à un intervalle non vide de la numérotation préfixe des configurations utiles : on note cet intervalle $[\text{first}(x, q), \text{last}(x, q)]$. Autrement dit, on a l'égalité :

$$\text{Conf}^{useful}(x, q) = \{\text{Conf}^{useful}[i] : \text{first}(x, q) \leq i \leq \text{last}(x, q)\}$$

Il est essentiel de noter que tous les calculs de ces objets, calcul préalable des configurations vides, calcul et parcours préfixe de la forêt $F^{\emptyset}$, construction des tableaux $\text{Conf}^{useful}[i]$, pour $i \in [0, u-1]$, $\text{first}(x, q)$ et $\text{last}(x, q)$, pour $(x, q) \in \text{Conf}$, se font en temps $O(|\Gamma| \cdot |T|)$.

Tout ceci nous permet d'implémenter efficacement l'équation (3′) qui devient :

pour toute configuration possible (x, q),

$$S^*(x, q) = \biguplus_{i \in [\text{first}(x,q), \text{last}(x,q)]} S^{useful}(\text{Conf}^{useful}[i]) \qquad (3")$$

3.3.8 Précalculs nécessaires

Pour mettre en oeuvre efficacement, c'est-à-dire à délai linéaire (en la taille de la solution S calculée), les équations précédentes (2') et (3"), il est nécessaire de calculer préalablement les ensembles $\text{Conf}^{possible}$, Conf^{useful} et $\text{Pair}^{useful}(x, q)$, pour toute configuration (x, q) utile. On représentera les ensembles $\text{Conf}^{possible}$ et Conf^{useful} par des tableaux booléens à deux indices $x \in \text{Dom}(T)$ et $q \in Q$ et on exécutera les précalculs ci-dessous.

Il est clair que chacun de ces précalculs se fait en temps $O(|T| \cdot |Q|^2) = O(|\Gamma| \cdot |T|)$.

3.3.9 Equations finales

Enfin, on définit l'ensemble des parties non vides $S \subseteq \text{Leaves}(T)$ telles que $S \xrightarrow{\Gamma,T} q$, pour un état final q de Γ :

$$S^*_{accept} = \biguplus_{q \in F} S^*(\text{root}(T), q)$$

Là encore, on se restreindra aux états finals possibles.

Algorithm 11 Précalcul de $\text{Conf}^{possible}$

$\text{Conf}^{possible} \leftarrow \emptyset$
pour toute feuille x et tout état $q \in Q$ tels que $\delta_0((\text{label}_T(x), 1)) = q$ **faire**
  ajouter (x, q) dans $\text{Conf}^{possible}$
fin pour
pour tout noeud interne x d'enfants y_1 et y_2 (les x sont parcourus dans l'ordre postfixé) **faire**
  pour tout $q_1 \in Q$ et tout $q_2 \in Q$ **faire**
    si $(y_1, q_1) \in \text{Conf}^{possible}$ ou $(y_2, q_2) \in \text{Conf}^{possible}$ **alors**
      soit l'état q tel que $q_1, q_2 \rightarrow q$
      ajouter (x, q) dans $\text{Conf}^{possible}$
    fin si
  fin pour
fin pour

Algorithm 12 Précalcul des ensembles Conf^{useful} et $\text{Pair}^{useful}(x, q)$

$\text{Conf}^{useful} \leftarrow \emptyset$
pour toute feuille x et tout état $q \in Q$ tels que $\delta_0((\text{label}_T(x), 1)) = q$ **faire**
  ajouter (x, q) dans Conf^{useful}
fin pour
pour tout noeud interne x d'enfants y_1 et y_2 **faire**
  pour tout $q \in Q$ **faire**
    $\text{Pair}^{useful}(x, q) \leftarrow \emptyset$
  fin pour
  pour tout $q_1 \in Q$ et tout $q_2 \in Q$ **faire**
    si $(y_1, q_1) \in \text{Conf}^{possible}$ et $(y_2, q_2) \in \text{Conf}^{possible}$ **alors**
      soit l'état q tel que $q_1, q_2 \rightarrow q$
      ajouter (x, q) dans Conf^{useful}
      ajouter (q_1, q_2) dans $\text{Pair}^{useful}(x, q)$
    fin si
  fin pour
fin pour

Définition 121 (états finals possibles) *Un état final q est possible si la configuration $(root(T), q)$ est possible. L'ensemble de ces états est noté*

$$F^{possible} = \{q \in F : (root(T), q) \in Conf^{possible}\}$$

On a évidemment la partition de l'ensemble S^*_{accept} en la réunion disjointe d'ensembles non vides donnée par l'équation suivante :

$$S^*_{accept} = \biguplus_{q \in F^{possible}} S^*(\text{root}(F), q) \tag{4}$$

En considérant, tout à la fin, le cas de l'ensemble vide $S = \emptyset$, on calculera facilement

$$S_{accept} = \begin{cases} S^*_{accept} \cup \{\emptyset\} & \text{si l'état vide de root}(T) \text{ appartient à } F \\ S^*_{accept} & \text{sinon.} \end{cases} \tag{5}$$

3.3.10 Mise en oeuvre algorithmique

A côté des objets précalculés, un point essentiel de notre algorithme est le fait que chaque ensemble non vide $S \subseteq \text{Leaves}(T)$ que nous manipulons sera représenté par l'ensemble des feuilles d'un arbre binaire dans sa représentation chaînée habituelle. Une telle représentation (non canonique) a les deux propriétés suivantes :

- L'affectation $S \leftarrow S_1 \uplus S_2$ (réunion disjointe) s'effectue en temps constant par la création d'un noeud racine t pour S dont les enfants gauche et droit sont choisis être les racines t_1 et t_2 des arbres qui représentent respectivement S_1 et S_2.
- Tout arbre représentant un ensemble S est de taille $O(|S|)$ et on effectue l'affichage de S par un simple parcours des feuilles de cet arbre, donc en temps $O(|S|)$.

Les équations (1), (2'), (3") et (4) précédentes justifient l'algorithme suivant, noté Affiche-les-S-acceptés. Cet algorithme utilise les objets précalculés et les procédures suivantes, notées Enum$S^{useful}(x, q)$ et Enum$S^*(x, q)$ calculées par une récursivité indirecte en fonction de la hauteur du sous-arbre T_x de T de racine x.

Complexité de l'algorithme d'énumération

Puisque l'algorithme se ramène essentiellement à l'appel de procédure Enum$S^*(\text{root}(T), q)$ qu'on itère pour tous les états $q \in F^{possible}$, il suffit de montrer que, dans l'appel de procédure Enum$S^*(x, q)$ où (x, q) est une configuration possible, chaque ensemble S généré est calculé en temps (= délai) au plus $c_1|S|$, pour une constante c_1 indépendante de T mais aussi de Γ. C'est la conséquence du lemme suivant.

Lemme 101 *Il existe une constante c, indépendante de T et Γ, qui vérifie les deux conditions suivantes :*

Algorithm 13 Enum$S^{useful}(x, q)$

Précondition: (x, q) est une configuration utile

```
si x est une feuille alors
  mettre en sortie l'ensemble {x}
sinon
  soient y1 et y2 les enfants gauche et droit de x
  pour tout (q1, q2) ∈ Pair^useful(x, q) faire
    pour tout S1 ∈ EnumS*(y1, q1) faire
      pour tout S2 ∈ EnumS*(y2, q2) faire
        mettre en sortie S1 ⊎ S2
      fin pour
    fin pour
  fin pour
fin si
```

Algorithm 14 Enum$S^*(x, q)$

Précondition: (x, q) est une configuration possible

```
pour i variant de first(x, q) à last(x, q) faire
  (x', q') ← Conf^useful[i]
  appeler EnumS^useful(x', q')
fin pour
```

Algorithm 15 Affiche-les-S-acceptés

```
pour tout q ∈ F^possible faire
  pour tout S ∈ EnumS*(root(T), q) faire
    Affiche-Ensemble(S) (c'est-à-dire, afficher les feuilles de l'arbre binaire représentant S puis
    détruire la racine de cet arbre)
  fin pour
fin pour
```

1. *Pour chaque configuration utile (x, q), chaque ensemble S généré dans l'appel de procédure Enum$S^{useful}(x, q)$ l'est après un délai au plus $c(4|S| - 3)$.*
2. *Pour chaque configuration possible (x, q), chaque ensemble S généré dans l'appel de procédure Enum$S^{*}(x, q)$ l'est après un délai au plus $c(4|S| - 2)$.*

Preuve. Les points (1) et (2) sont prouvés simultanément par induction sur la hauteur du sous-arbre T_x de T de racine x.

Cas de base : Si x est une feuille de T, l'ensemble $S = \{x\}$ (en fait l'arbre-feuille qui le représente) est généré par Enum$S^{useful}(x, q)$ en temps constant et est donc de même généré par Enum$S^{*}(x, q)$ en temps constant. On prend c supérieur à cette constante.

Induction : Point (1) : Soit (x, q) une configuration utile d'un noeud interne x et soient y_1 et y_2 les enfants gauche et droit de x dans T. Soit S un ensemble généré par l'appel de procédure Enum$S^{useful}(x, q)$, qui appelle à son tour Enum$S^{*}(y_1, q_1)$ et Enum$S^{*}(y_2, q_2)$ qui construisent les ensembles respectifs S_1 et S_2 dont la réunion disjointe est $S = S_1 \uplus S_2$. Par hypothèse d'induction, S_1 (resp. S_2) est calculé avec délai au plus $c(4|S_1| - 2)$ (resp. $c(4|S_2| - 2)$. En conséquence, $S = S_1 \uplus S_2$ est calculé avec un délai, noté delay(S), qui est au plus

$$c(4|S_1| - 2) + c(4|S_1| - 2) + c'$$

(c' correspond essentiellement au temps pour réaliser la réunion disjointe de S_1 et S_2.) En supposant $c \geq c'$, on obtient

$$\text{delay}(S) \leq c(4(|S_1| + |S_2|) - 4) + c = c(4|S| - 3)$$

comme demandé pour le point (1).

Point (2) : Soit un ensemble S généré par l'appel de procédure Enum$S^{*}(x, q)$ qui produit ce S par l'appel emboîté Enum$S^{useful}(x', q')$ pour un noeud x' descendant (au sens large) de x dans T. Par hypothèse d'induction (ou à cause du point (1) précédent si $x' = x$), le délai de Enum$S^{useful}(x', q')$ pour produire S est au plus $c(4|S| - 3)$. Le délai, noté delay(S), de Enum$S^{*}(x, q)$ pour produire S est donc

$$\text{delay}(S) \leq c(4|S| - 3) + c''$$

pour une constante c''. Si on prend encore $c \geq c''$, on obtient

$$\text{delay}(S) \leq c(4|S| - 2)$$

comme demandé. Le lemme est donc prouvé. □

Lemme 102 *Il existe une constante c_1 indépendante de T et Γ telle que l'algorithme Affiche-les-S-acceptés énumère les ensembles S tels que l'automate Γ accepte le $\Sigma'_{@}$-arbre T^S, à délai, pour produire S, au plus $c_1|S|$, et en espace, sur ce délai, au plus $c_1|S|$.*

Preuve. Le délai a déjà été justifié : prendre $c_1 \geq 4c$. La mémoire utilisée pour produire S est, d'une part, celle utilisée par l'arbre binaire qui représente S et possède $\text{card}(S) = |S|$ feuilles et donc $2|S| - 1$ noeuds et, d'autre part, celle utilisée par la pile de récursivité, de hauteur au plus $|S|$. Notons qu'il est essentiel qu'à l'issue de l'affichage de S, la procédure Affiche-Ensemble(S) détruise la racine de l'arbre binaire qui représente cet ensemble. La mémoire utilisée par cette racine peut alors être réutilisée pour l'arbre d'un nouvel ensemble à générer. □

3.3.11 Résultats finals pour l'énumération

On donne ces résultats sous deux formes selon que l'on se donne un automate d'arbre ou une formule **MSO**.

Proposition 103 *Il existe une constante c et un schéma d'énumération A, tels que, étant donnés un alphabet fini Σ, un automate Γ de $\Sigma'_@$-arbre et un $\Sigma_@$-arbre T (où $\Sigma_@ = \Sigma \cup \{@\}$ et $\Sigma'_@ = (\Sigma \times \{0,1\}) \cup \{@\}$), l'algorithme A énumère l'ensemble*

$$\{S \subseteq \mathit{Leaves}(T) : \Gamma \text{ accepte l'arbre étiqueté } T^S\}$$

avec un délai et et une mémoire [5] au plus $c|S|$, après précalcul de temps au plus $c \cdot |\Gamma| \cdot |T|$.

Preuve. On a déjà justifié le délai et on a vu que chaque objet précalculé l'est en temps $O(|\Gamma| \cdot |T|)$. □

Théorème 104 *Pour toute τ_Σ-formule* **MSO** *φ, le problème d'énumération* Enum$(\varphi, \Sigma\text{-}Tree)$ *pour la classe Σ-Tree des Σ-arbres, est dans* Lin-Delay$_{lin}$.

Plus précisément, il existe une constante c, une fonction calculable f (non élémentaire) et un schéma d'énumération A tels que, étant donnés un alphabet fini Σ, une τ_Σ-formule $\varphi(X_1, \ldots, X_k)$ dans **MSO** *et un Σ-arbre T, le schéma A :*

- *réalise un précalcul de temps $f(|\varphi|) \cdot |T|$;*
- *puis énumère chaque solution $\bar{S} = (S_1, \ldots, S_k) \in \varphi(T)$ avec un délai et une mémoire au plus $c|\bar{S}|$.*

En conséquence, A fonctionne en temps total

$$f(|\varphi|) \cdot |T| + c|\varphi(T)|$$

Preuve. c'est une conséquence immédiate des résultats précédents. Notons que le passage (décodage) d'un ensemble S au k-uplet d'ensembles $(S_1, \ldots, S_k)$, codé par S, se fait en temps $O(|S|)$. □

[5] la mémoire est celle utilisée dans le délai pour produire S ; en particulier, elle n'inclut pas la mémoire utilisée par les objets précalculés.

3.4 Calcul de la j-ème solution

3.4.1 Problématique

Comme on a traduit les équations (1), (2'), (3"), (4), (5) par des procédures d'énumération, on peut de façon similaire leur associer des formules calculant la j-ème solution du problème. Pour cela il faut préciser dans quel ordre total on prend les solutions. Nous nous contenterons ici d'un ordre ad hoc, induit par les équations elles-mêmes. On aura besoin des définitions et notations suivantes.

Définition 122 (i-ème élément) *Soit P un ensemble muni d'un ordre total $<$. On note $P[i]$ le i-ème élément de P pour l'ordre $<$. Pour simplifier, on prend encore $i \in [0, card(P) - 1]$ et en particulier $P[0]$ désigne en fait le premier élément de $(P, <)$.*

On définit l'ordre induit d'une union disjointe ou d'un produit d'ensembles ordonnés.

Définition 123 (ordre induit d'une union disjointe) *Soient $(P_1, <_1)$ et $(P_2, <_2)$ deux ensembles ordonnés disjoints. L'ordre induit de $P_1 \uplus P_2$ est obtenu par réunion disjointe de l'ordre $<_1$ de P_1 placé avant l'ordre $<_2$ de P_2, c'est-à-dire, pour tout entier $i < card(P_1) + card(P_2)$,*

$$(P_1 \uplus P_2)[i] = \begin{cases} P_1[i] & si\ i < card(P_1) \\ P_2[i - card(P_1)] & sinon \end{cases}$$

Définition 124 (ordre induit d'un produit) *Soient $(P_1, <_1)$ et $(P_2, <_2)$ deux ensembles ordonnés tels que $P_1 \subseteq \mathcal{P}(E_1)$ et $P_2 \subseteq \mathcal{P}(E_2)$, pour deux ensembles disjoints E_1 et E_2. L'ordre induit $<$ de $P_1 \otimes P_2$ est défini de la façon suivante : pour tout entier $i < card(P_1) \cdot card(P_2)$, on prend*

$$(P_1 \otimes P_2)[i] = P_1[i\ div\ card(P_2)] \uplus P_2[i\ mod\ card(P_2)]$$

Autrement dit, $<$ est l'ordre lexicographique produit de $<_1$ et $<_2$.

Le calcul du i-ème élément d'un ensemble S s'appuie sur les deux principes suivants :

- Cas $S = S_1 \otimes S_2$: Soit $n = \mathrm{card}(S_2)$; alors $S[i] = S_1[i \operatorname{div} n] \uplus S_2[i \bmod n]$.
- Cas $S = S_0 \uplus S_1 \uplus \ldots \uplus S_m$: Soit $\mathrm{sum}S(j) = \sum_{i \in [0,j-1]} \mathrm{card}(S_i)$ pour $j \in [0, m]$; alors, $S[i] = S_j[i']$ où j est le plus grand j tel que $\mathrm{sum}S(j) \leq i$ et $i' = i - \mathrm{sum}S(j)$.

Dans le but d'appliquer ces principes aux équations (1), (2'), (3"), (4), (5), nous avons besoin de précalculer les valeurs suivantes :

- Les cardinalités $\mathrm{card}(S^*(x, q))$, pour toute configuration possible (x, q), et $\mathrm{card}(S^{useful}(x, q))$, pour toute configuration utile (x, q).
- Les sommes itérées :

$$\mathrm{sum}Pair^{useful}(x,q,i) = \sum_{\substack{j\in[0,i-1]\\(q_1,q_2)=\mathrm{Pair}^{useful}(x,q)[j]}} \mathrm{card}(S^*(y_1,q_1)\otimes S^*(y_2,q_2))$$

et

$$\mathrm{sum}S^{useful}(i) = \sum_{j\in[0,i-1]} \mathrm{card}(S^{useful}[j])$$

3.4.2 Calcul des cardinalités des $S^*(x,q)$ et $S^{useful}(x,q)$

Nous donnons ici deux algorithmes pour calculer respectivement $\mathrm{card}(S^*(x,q))$ et $\mathrm{card}(S^{useful}(x,q))$, pour tout sommet $x \in \mathrm{Dom}(T)$ et tout état $q \in Q$. Les deux algorithmes (donnés ci-dessous) procèdent par un parcours "bottom-up" de l'arbre.

Algorithm 16 Précalcul des $\mathrm{card}(S^*(x,q))$

pour toute feuille x et tout état $q \in Q$ tels que $\delta_0((\mathrm{label}_T(x),1)) = q$ **faire**
  $\mathrm{card}(S^*(x,q)) \leftarrow 1$
fin pour
pour tout noeud interne x d'enfants y_1 et y_2 (les x sont parcourus dans l'ordre postfixé) **faire**
  pour tout $q \in Q$ **faire**
    $\mathrm{card}(S^*(x,q)) \leftarrow 0$
  fin pour
  pour tout $q_1 \in Q$ et tout $q_2 \in Q$ **faire**
    si $(y_1,q_1) \in \mathrm{Conf}^{possible}$ et $(y_2,q_2) \in \mathrm{Conf}^{vide}$ **alors**
      soit l'état q tel que $q_1, q_2 \to q$
      $\mathrm{card}(S^*(x,q)) \leftarrow \mathrm{card}(S^*(x,q)) + \mathrm{card}(S^*(y_1,q_1))$
    fin si
    si $(y_1,q_1) \in \mathrm{Conf}^{vide}$ et $(y_2,q_2) \in \mathrm{Conf}^{possible}$ **alors**
      soit l'état q tel que $q_1, q_2 \to q$
      $\mathrm{card}(S^*(x,q)) \leftarrow \mathrm{card}(S^*(x,q)) + \mathrm{card}(S^*(y_2,q_2))$
    fin si
    si $(y_1,q_1) \in \mathrm{Conf}^{possible}$ et $(y_2,q_2) \in \mathrm{Conf}^{possible}$ **alors**
      soit l'état q tel que $q_1, q_2 \to q$

      $\mathrm{card}(S^*(x,q)) \leftarrow \mathrm{card}(S^*(x,q)) + \mathrm{card}(S^*(y_1,q_1)) \times \mathrm{card}(S^*(y_2,q_2))$
    fin si
  fin pour
fin pour

Algorithm 17 Précalcul des $\text{card}(S^{useful}(x,q))$

pour toute feuille x et tout état $q \in Q$ tels que $\delta_0((\text{label}_T(x),1)) = q$ **faire**
  $\text{card}(S^{useful}(x,q)) \leftarrow 1$
fin pour
pour tout noeud interne x d'enfants y_1 et y_2 **faire**
  pour tout $q \in Q$ **faire**
    $\text{card}(S^{useful}(x,q)) \leftarrow 0$
  fin pour
  pour tout $q_1 \in Q$ et tout $q_2 \in Q$ **faire**
    si $(y_1,q_1) \in \text{Conf}^{possible}$ et $(y_2,q_2) \in \text{Conf}^{possible}$ **alors**
      soit l'état q tel que $q_1, q_2 \rightarrow q$
      $\text{card}(S^{useful}(x,q)) \leftarrow \text{card}(S^{useful}(x,q)) + \text{card}(S^*(y_1,q_1)) \times \text{card}(S^*(y_2,q_2))$
    fin si
  fin pour
fin pour

3.4.3 Phase dynamique de la recherche de la j-ème solution

Comme pour l'énumération, nous donnons deux algorithmes $\text{Jth}S^{useful}(x,q,i)$ et $\text{Jth}S^*(x,q,i)$ qui calculent la i-ème solution des ensembles respectifs $S^{useful}(x,q)$ et $S^*(x,q)$. Ces algorithmes (donnés ci-dessous) sont définis par récursivité croisée fondée sur la taille de l'arbre T_x de racine x.

3.4.4 Analyse de l'algorithme et conséquences

Nous nous restreignons au cas où les additions et multiplications peuvent se faire en temps constant. Pour cela, il est nécessaire de borner le nombre total de solutions d'une requête dans le souci de ne manipuler que des entiers contenus chacun dans un nombre constant de registres. Sous ces conditions, la division et le modulo peuvent s'implémenter, par des procédés classiques, en temps logarithmique en la taille de la structure pour le coût uniforme.

Définition 125 (formule polynomialement bornée) *Une τ-formule φ est dite polynomialement bornée sur une classe $\mathcal{S}$ de structures s'il existe un polynôme p tel que, pour toute τ-structure $\mathcal{M}$ de $\mathcal{S}$, on a $card(\varphi(\mathcal{M})) \leq p(|\mathcal{M}|)$.*

Remarque 27 *Soit φ une τ_Σ-formule polynomialement bornée sur les Σ-arbres, c'est-à-dire pour laquelle $card(\varphi(T)) \leq p(|T|)$, pour tous les Σ-arbres T. Alors il est facile de voir que les cardinalités de tous les ensembles $S^*(x,q)$ et $S^{useful}(x,q)$ construits lors des appels récursifs de*

Algorithm 18 $\mathrm{Jth}S^{useful}(x, q, i)$

Précondition: (x, q) est une configuration utile et $i < \mathrm{card}(S^{useful}(x, q))$

si x est une feuille **alors**

return $\{x\}$

sinon

chercher le plus grand j tel que $\mathrm{sumPair}^{useful}(x, q, j) \leq i$

$i' \leftarrow i - \mathrm{sumPair}^{useful}(x, q, j)$

$(q_1, q_2) \leftarrow \mathrm{Pair}^{useful}(x, q)[j]$

$n \leftarrow \mathrm{card}(S^*(y_2, q_2))$

$S_1 \leftarrow \mathrm{Jth}S^*(x_1, q_1, i' \operatorname{div} n)$

$S_2 \leftarrow \mathrm{Jth}S^*(x_2, q_2, i' \bmod n)$

return $S_1 \uplus S_2$

fin si

Algorithm 19 $\mathrm{Jth}S^*(x, q, i)$

Précondition: (x, q) est une configuration possible et $i < \mathrm{card}(S^*(x, q))$

chercher le plus grand j tel que $\mathrm{sum}S^{useful}(j) - \mathrm{sum}S^{useful}(\mathrm{first}(x, q)) \leq i$

$i' \leftarrow i - \mathrm{sumS}^{useful}(j) + \mathrm{sumS}^{useful}(\mathrm{first}(x, q))$

$(y, q') \leftarrow S^{useful}[j]$

return $\mathrm{Jth}S^{useful}(y, q', i')$

l'algorithme d'énumération sont bornés par card($\varphi(T)$). Il en découle que chacun de ces nombres peut être contenu dans un nombre constant de registres.

On voit aisément que la phase de précalcul s'effectue en temps $O(|T| \cdot |\Gamma|)$.

Complexité de la phase dynamique

Analogue à la phase d'énumération du schéma d'énumération, cette phase dynamique se ramène à l'appel de la procédure Jth$S^*(\text{root}(T), q, i)$, pour un état final $q \in F$ facile à déterminer. Comme on l'a signalé précédemment, on se restreint au cas où les entiers manipulés utilisent un nombre constant de registres.

Il suffit de montrer que l'appel de procédure Jth$S^*(x, q, i)$ où (x, q) est une configuration possible, s'exécute en temps au plus $c_1|S| \log |T|$, pour une constante c_1 indépendante de T, mais aussi de Γ. C'est la conséquence du lemme suivant.

Lemme 105 *Il existe une constante c, indépendante de T et Γ, qui vérifie les deux conditions suivantes :*

1. *Pour chaque configuration utile (x, q), la sortie S de l'appel Jth$S^{useful}(x, q, i)$ est calculée en temps au plus $c(4|S| - 3)(\log |T| + \log |Q|)$.*
2. *Pour chaque configuration possible (x, q), la sortie S de l'appel Jth$S^*(x, q, i)$ est calculée en temps au plus $c(4|S| - 2)(\log |T| + \log |Q|)$.*

Preuve. Les points (1) et (2) sont prouvés simultanément par induction sur la hauteur du sous-arbre T_x de T de racine x.

Cas de base : Si x est une feuille de T, l'ensemble $S = \{x\}$ (en fait l'arbre-feuille qui le représente) est calculé par Jth$S^{useful}(x, q, i)$ en temps constant et est donc de même calculé par la procédure Jth$S^*(x, q, i)$ en temps constant. On prend c supérieur à cette constante.

Induction : Point (1) : Soit (x, q) une configuration utile d'un noeud interne x et soient y_1 et y_2 les enfants gauche et droit de x dans T. Soit S la sortie de l'appel de procédure Jth$S^{useful}(x, q, i)$ qui appelle à son tour Jth$S^*(y_1, q_1, i_1)$ et Jth$S^*(y_2, q_2, i_2)$ qui construisent les ensembles respectifs S_1 et S_2 dont la réunion est $S = S_1 \uplus S_2$. Par hypothèse d'induction, S_1 (resp. S_2) est calculé en temps au plus $c(4|S_1| - 2)(\log |T| + \log |Q|)$ (resp. $c(4|S_2| - 2)(\log |T| + \log |Q|)$). En conséquence, l'ensemble $S = S_1 \uplus S_2$ est calculé en temps, noté time(S), qui est au plus

$$c(4|S_1| - 2)(\log |T| + \log |Q|) + c(4|S_2| - 2)(\log |T| + \log |Q|) + c'(\log |T| + \log |Q|)$$

Le terme $c'(\log |T| + \log |Q|)$ correspond essentiellement au temps de la recherche dichotomique dans un tableau de taille bornée par $\text{card}(Q)^2$, et au temps de la division et du modulo. En

supposant $c \geq c'$, on obtient

$$\text{time}(S) \leq c(4(|S_1|+|S_2|)-4)(\log|T|+\log|Q|)+c(\log|T|+\log|Q|) = c(4|S|-3)(\log|T|+\log|Q|)$$

comme demandé pour le point (1).

Point (2) : Soit S la sortie de l'appel de procédure $\text{Jth}S^*(x,q,i)$ qui produit S par l'appel emboîté $\text{Jth}S^{useful}(x',q',i')$, pour un noeud x' descendant (au sens large) de x. Par hypothèse d'induction (ou à cause du point (1) précédent si $x' = x$), le temps que met $\text{Jth}S^{useful}(x',q',i')$ pour produire S est au plus $c(4|S|-3)(\log|T|+\log|Q|)$. Le temps, noté $\text{time}(S)$, de $\text{Jth}S^*(x,q,i)$ pour produire S est donc

$$\text{time}(S) \leq c(4|S|-3)(\log|T|+\log|Q|) + c''(\log|T|+\log|Q|)$$

pour une constante c'', où l'expression $c''(\log|T|+\log|Q|)$ mesure le temps de la recherche dichotomique dans un tableau de taille bornée par $\text{card}(T)\cdot\text{card}(Q)$. Si on prend encore $c \geq c''$, on obtient

$$\text{time}(S) \leq c(4|S|-2)(\log|T|+\log|Q|)$$

comme demandé. Le lemme est donc prouvé. □

On en déduit le théorème suivant :

Théorème 106 *Soit $\varphi(\bar{X})$ une τ_Σ-formule* **MSO** *polynomialement bornée sur la classe des Σ-arbres. Alors le problème* $\text{JTH}(\varphi, \Sigma\text{-}Tree)$ *est dans* $dyn(n, m\log(n))$.

En particulier, on obtient :

Théorème 107 *Soit φ une τ_Σ-formule* **MSO** *dont les variables libres sont du premier ordre. Alors* $\text{JTH}(\varphi, \Sigma\text{-}Tree) \in dyn(n, \log(n))$.

Preuve. Comme les variables de φ sont du premier ordre, $\text{card}(\varphi(\mathcal{M}))$ est borné par $\text{card}(\text{Dom}(\mathcal{M}))^k$ où k est le nombre de variables libres de φ. □

3.5 Généralisation aux structures arborescentes

3.5.1 Classes de structures de largeur arborescente bornée

Le but de cette section est de démontrer qu'il existe une **MSO**-interprétation linéaire de toute classe de structures de largeur arborescente bornée dans la classe des Σ-arbres et que, par conséquent, nos résultats se généralisent à ces classes de structures. Ce résultat est déjà connu (voir par exemple [ALS91, FFG02]). Mais, par souci de complétude, nous en donnons ici une nouvelle preuve. Pour simplifier les notations, nous nous intéressons uniquement au cas des graphes, mais la preuve se généralise facilement aux structures quelconques.

Décomposition arborescente sympathique

Pour présenter notre réduction, nous introduisons d'abord la notion de décomposition arborescente sympathique. Cette notion est inspirée de Bodlaender et Kloks [BK96].

Définition 126 (Décomposition arborescente sympathique) *Soit G un graphe. Une décomposition arborescente $(T, (B_t)_{t \in V(T)})$ de G est dite sympathique si T est un arbre binaire enraciné vérifiant les conditions suivantes :*

- *si t a deux enfants t' et t'' alors $B_t = B_{t'} = B_{t''}$;*
- *si t a un seul enfant t' alors $card(B_t \triangle B_{t'}) = 1$, autrement dit, B_t et $B_{t'}$ ne diffèrent que sur exactement un élément (de plus ou de moins) ;*
- *si t est une feuille ou la racine alors $B_t = \emptyset$.*

Pour simplifier les notations, on appellera encore T le couple $(T, (B_t)_{t \in V(T)})$.

Fait 108 *Il existe un algorithme qui, étant donné un graphe de largeur arborescente k, calcule une décomposition arborescente sympathique de largeur k en temps $O_k(|G|)$.*

Preuve. Soit une décomposition T de largeur k de G obtenue en temps linéaire par l'algorithme de Bodlaender [Bod96]. On modifie T en intercalant autant de noeuds intermédiaires que nécessaire de façon à respecter toutes les conditions d'une décomposition sympathique. C'est une construction facile mais fastidieuse à décrire et nous laissons le lecteur l'imaginer et s'en convaincre. □

Un noeud t qui a pour parent le sommet t' et tel que $B_t = B_{t'} \uplus \{x\}$ est appelé le noeud *introduisant* x. Un noeud t qui a pour parent le noeud t' et tel que $B_{t'} = B_t \uplus \{x\}$ est appelé noeud *oubliant* x. Remarquons qu'un noeud t de l'arbre ne peut pas être à la fois oubliant et introduisant et qu'il peut introduire ou oublier au plus un sommet de G. Pour chaque sommet x du graphe G, il existe exactement un noeud t qui introduit x. On appelle I_x ce noeud de T.

Représentation des sacs par des tuples

Nous allons représenter les (au plus) $k+1$ éléments d'un sac B de T par un $(k+1)$-tuple de sorte que chaque sommet de G apparaisse toujours à la même position dans tous les sacs qui le contiennent. Plus formellement, étant donnée une décomposition arborescente sympathique $T = (T, (B_t)_{t \in V(T)})$ de largeur k d'un graphe $G = (V, E)$, il est facile de construire en temps linéaire une fonction $L : V(T) \to (V \cup \{\perp\})^{k+1}$, $t \mapsto (L_i(t))_{i \in [k+1]}$ vérifiant les conditions suivantes :

- pour tout sommet x de G et pour tout noeud $t \in V(T)$, on a l'équivalence : $x = B_t$ si et seulement si il existe un (unique) i tel que $x \in L_i(t)$;

– pour tous noeuds $t, t' \in V(T)$, on a l'implication : si $x \in B_t \cap B_{t'}$ alors $x = L_i(t) = L_i(t')$ pour un même i.

Notons que le symbole $\perp$ désigne les positions vides dans L ; en particulier, pour chaque $t \in V(T)$, le nombre d'indices i tels que $L_i(t) = \perp$ est $k + 1 - \text{card}(B_t)$.

L est appelée *représentation par tuples* des sacs de $(T, (B_t)_{t \in V(T)})$. On appelle *position* de x dans L, l'indice (unique) i où x apparait dans chaque tuple de L où il est présent.

Exprimer une structure de largeur arborescente bornée dans un Σ-arbre

Lemme 109 *Soit σ une signature relationnelle et soit $\mathcal{S}$ la classe des σ-structures de largeur arborescente au plus k. Il existe une* **MSO***-interprétation linéaire de $\mathcal{S}$ dans la classe des Σ_k-arbres, pour un certain alphabet Σ_k dépendant seulement de k et σ.*

Preuve. Comme annoncé plus haut, nous traitons uniquement le cas des graphes : $\sigma = \{E\}$. Soit $G = (V, E)$ un graphe de largeur arborescente k. A ce graphe on associe, en temps linéaire, une décomposition arborescente sympathique $T = (T, (B_t)_{t \in V(T)})$ de largeur k, et une représentation L par $(k+1)$-tuples des sacs de T. A ce graphe G muni de T et de L, on associe la structure $\mathcal{M}_T = (D, f_1, f_2, (L_i)_{i \in [k+1]}, (I_i)_{i \in [k+1]}, (E_{i,j})_{i,j \in [k+1]})$ suivante :

– Le domaine D est l'ensemble des noeuds de l'arbre binaire T : $D = V(T)$;
– f_1 et f_2 sont les relations enfants gauche et droit de cet arbre T ;
– Pour chaque sommet x du graphe G, on identifie x au noeud $t = I_x$ de l'arbre T introduisant x, d'où $V \subseteq D$;
– Les L_i, I_i et $E_{i,j}$ sont des relations unaires sur D et, pour chaque noeud $t \in D$, le tuple des valeurs $L_i(t)$, $I_i(t)$ et $E_{i,j}(t)$ forme l'étiquette du noeud t ; on note Σ_k l'ensemble des étiquettes possibles ; ces relations unaires sont définies comme suit :
  – $L_i = \{t \in D : L_i(t) \in V\}$, autrement dit, on a $L_i(t)$ si et seulement si la position i dans $L(t)$ n'est pas vide ;
  – $I_i = \{x \in V : x \text{ est en position } i \text{ dans } L\}$, autrement dit, on a $I_i(t)$ si et seulement si t est le noeud introduisant x en position i ;
  – $E_{i,j} = \{t \in D : (L_i(t), L_j(t)) \in E\}$, autrement dit, on a $E_{i,j}(t)$ si et seulement si le sac B_t contient, en positions i et j dans L, deux sommets adjacents de G.

Il est clair que la construction de $\mathcal{M}_T$ à partir de G et de sa décomposition arborescente sympathique $(T, (B_t)_{t \in V(T)})$ s'effectue en temps linéaire. Donc la transformation $G \mapsto \mathcal{M}_T$ est aussi linéaire.

Il est clair aussi que la formule $\Delta(x) \equiv \bigvee_{i \in [k+1]} I_i(x)$ interprète le domaine V de G dans $\mathcal{M}_T$: pour tout noeud $a \in D$, on a $a \in V$ si et seulement si $(\mathcal{M}_T, a) \models \Delta(x)$.

On va construire la formule $\Theta_E(x, y)$ qui interprète la relation d'arête E en utilisant plusieurs formules intermédiaires.

La formule "chemin" suivante utilise la formule

$$\text{child}(x,y) \equiv f_1(x,y) \vee f_2(x,y)$$

et exprime que l'ensemble de noeuds A forme un chemin orienté (de l'arbre T) reliant x à y (qui est donc descendant de x dans T) :

$$\text{chemin}(A,x,y) \equiv A(x) \wedge \forall z((A(z) \wedge z \neq y) \rightarrow \exists t(\text{child}(z,t) \wedge A(t))) \wedge \forall z(A(z) \rightarrow \neg\text{child}(z,x))$$

On utilise maintenant le fait que $(T,(B_t)_{t\in V(T)})$ est une décomposition arborescente sympathique et a pour représentation par tuples L. Pour chaque arête $\{x,y\}$ de G, il existe un sac B_t qui contient x et y. Soient i et j les positions respectives de x et y dans L. On a donc $E_{i,j}(t)$. Soient I_x et I_y les noeuds de l'arbre qui introduisent respectivement x et y. On rappelle l'identification $x = I_x$ et $y = I_y$. Par construction, le chemin de T entre les noeuds x (resp. y) et t est inclus dans L_i (resp. L_j). A partir des arguments précédents, on démontre sans difficulté l'équivalence demandée : pour tous sommets $a,b \in V$, on a $\{a,b\} \in E$ si et seulement si $(\mathcal{M}_T,a,b) \models \Theta_E(x,y)$ avec

$$\Theta_E(x,y) \equiv \exists A\exists B\exists t(\text{chemin}(A,x,t) \wedge \text{chemin}(B,y,t) \wedge \bigvee_{i,j\in[k+1]} (I_i(x) \wedge I_j(y) \wedge E_{i,j}(t) \wedge \forall t'(A(t') \rightarrow L_i(t')) \wedge \forall t'(B(t') \rightarrow L_j(t'))))$$

Justification : La clause $A \subseteq L_i$ de Θ_E impose que le sommet x introduit à la position i du noeud I_x (début du chemin A) soit toujours présent (à la position i) dans le noeud B_t (t fin du chemin A). Le raisonnement est le même pour le sommet y à la position j avec le chemin B. En conclusion, puisque $L_i(t) = x$ et $L_j(t) = y$, la clause $E_{i,j}(t)$ exprime bien $\{x,y\} \in E$.

Fin de la preuve : on a donc établi que le triplet $I = (t_I, \Delta, \Theta_E)$ où t_I est la transformation $G \mapsto \mathcal{M}_T$ est une **MSO**-interprétation linéaire de la classe des graphes de largeur arborescente k dans la classe des Σ_k-arbres. □

Théorème 110 *Soit $\mathcal{S}$ une classe de σ-structures de largeur arborescente bornée. Alors, pour toute σ-formule φ de* **MSO**, *le problème* Enum$(\varphi,\mathcal{S})$ *est dans* Lin-Delay$_{lin}$. *Autrement dit, pour toute σ-structure $\mathcal{M} \in \mathcal{S}$, on peut énumérer les solutions S de $\varphi(\mathcal{M})$ à délai $O(|S|)$ après un précalcul en temps $O(|\mathcal{M}|)$.*

De plus, si la formule φ est polynomialement bornée sur la classe $\mathcal{S}$, alors le problème Jth$(\varphi,\mathcal{S})$ *appartient à la classe dyn$(n, m\log n)$.*

3.5.2 Classes de graphes de largeur de clique bornée

Le résultat précédent se généralise à toute classe de graphes de largeur de clique bornée. En effet, Courcelle et al. [CMR00] ont prouvé qu'il existe une **MSO**-interprétation linéaire des graphes représentés par une clique-décomposition de largeur k, dans les Σ-arbres. Par ailleurs, on rappelle que Hlineny et Oum [HO07] ont exhibé un algorithme qui prend en entrée un graphe G et un entier $k > 0$ et qui vérifie les conditions suivantes.

- Cet algorithme fonctionne en temps $O_k(n^3)$ où n est le nombre de sommets de G ;
- Soit l'algorithme calcule une clique-décomposition de largeur $f(k)$, soit il certifie que l'on a $\mathrm{cw}(G) > k$.

Ceci implique le théorème suivant.

Théorème 111 *Soit $\mathcal{S}$ une classe de graphes de largeur de clique bornée. Alors, pour toute $\{E\}$-formule φ de* **MSO**, *le problème* $\textsc{Enum}(\varphi, \mathcal{S})$ *est dans* $enum(n^3, m)$.

De plus, si la formule φ est polynomialement bornée sur la classe $\mathcal{S}$, alors le problème $\textsc{Jth}(\varphi, \mathcal{S})$ *appartient à la classe* $dyn(n^3, m \log n)$.

3.6 Conclusion

Plusieurs questions restent ouvertes :

Notre algorithme énumère en complexité $\textsc{Lin-Delay}_{lin}$ les solutions d'une requête **MSO** $\varphi(T)$ sur un arbre T, ceci dans un ordre ad hoc. Peut-on faire mieux dans des cas particuliers ? Par exemple, soit $\varphi(\bar{x})$ une formule **MSO** où $\bar{x}$ est une liste de variables du premier ordre, et soit un arbre T. Peut-on énumérer les éléments de $\varphi(T)$ dans un certain ordre lexicographique avec toujours la même complexité $\textsc{Lin-Delay}_{lin}$? Segoufin [Seg08] donne une nouvelle preuve de notre résultat d'énumération restreint aux formules **MSO** dont les variables libres sont toutes du premier ordre, ceci en se basant sur une forme normale de ces formules établie par Colcombet [Col07]. Pourrait-on obtenir par ces techniques l'énumération souhaitée dans l'ordre lexicographique ? Dans un cas plus restreint, en nous basant sur les résultats de normalisation logique de Durand et Olive [DO06], obtenue par élimination de quantificateurs, nous pensons pouvoir montrer que, pour un ordre lexicographique quelconque, on peut énumérer dans $\textsc{Constant-Delay}_{lin}$ les solutions d'une requête du premier ordre $\varphi(T)$ sur un arbre T.

Dans le cas général, on conjecture que le calcul de la j-ème solution d'une requête **MSO** sur un arbre ne peut pas s'effectuer en temps linéaire (en la solution produite) après un précalcul linéaire (ou même polynomial) en la taille de l'arbre. Il serait intéressant de fournir un argument du type suivant : si la réponse était positive, alors on pourrait effectuer une recherche dans un tableau de taille n en temps $o(\log n)$, donc meilleur que par la recherche dichotomique.

Chapitre 4

Requêtes du premier ordre sur les structures de degré borné

Sommaire

4.1 Introduction

Ce chapitre traite de l'énumération des solutions et du calcul de la j-ème solution d'une requête **FO** dans une structure de degré borné. Ce travail est principalement une extension de l'article [DG07]. Comme dans cet article, nous nous ramenons à l'étude de ces deux problèmes pour les *structures bijectives*, c'est-à-dire les structures composées uniquement de relations unaires et de fonctions bijectives. Dans ce chapitre, qui est essentiellement une version de l'article [BDGO08], nous présentons d'abord une forme d'élimination des quantificateurs sur la classe des structures bijectives et donnons une forme normale récursive des formules du premier ordre sur ces structures. En utilisant cette forme normale, nous démontrons d'abord que l'on peut énumérer l'ensemble des solutions d'une requête **FO** *dans l'ordre lexicographique* avec un *prétraitement linéaire* et un *délai constant*. Ce résultat améliore celui de [DG07] qui présente, pour le même problème, un algorithme d'énumération qui est aussi de complexité Constant-Delay$_{lin}$ mais énumère les solutions dans un ordre ad hoc.

Notre seconde contribution de ce chapitre est la suivante : nous montrons que l'on peut trouver la j-ème solution d'une requête **FO** sur une structure de degré borné (pour un ordre ad hoc) en *temps constant* après un *prétraitement linéaire* en la taille de la structure. Il faut souligner que nous obtenons ainsi la première classe naturelle de requêtes pour laquelle est établie une borne de complexité minimale (complexité appelée CONSTANT-TIME$_{lin}$) pour la recherche de la j-ème solution.

4.2 Normalization of first-order formulas on bijective structures

4.2.1 Logical definitions

A central notion studied in this article as in [DG07] is that of *bijective* structure. It is defined as follows.

Définition 127 *The* arity of a signature *is the maximal arity of its symbols. A* unary signature *has arity* 1. *Thus, a unary signature is made of constant, monadic predicate and unary function symbols. By a* unary structure *we mean a structure over a unary signature. A* bijective structure *is a unary structure in which all the unary function symbols are interpreted as permutations of the domain. We will assume that any bijective structure contains* at least one *unary function. The class of all bijective structures is denoted by* BIJ.

As in [DG07], we will focus on first-order logic over bijective structures. For convenience, we will handle functions and their reciprocals directly in the syntax. This suggests the following definition.

Définition 128 *A* bijective term $\tau(x)$ *is of the form* $f_1^{\epsilon_1} \dots f_l^{\epsilon_l}(x)$ *where* $l \geq 0$, $\epsilon_i = \pm 1$, x *is a variable and each* $f_i^{\epsilon_i}$ *is either the function symbol* f_i *or its inverse* f_i^{-1}.

The notions of *bijective atomic formulas*, *bijective literals* and *bijective first-order formulas* are defined as usual from bijective terms.

We will often denote by τ or $\tau(x)$ a bijective term, according to our intention to ignore or to specify its constitutive variable. We will also make an extensive use of the intermediate notation $\tau(\overline{x})$, which indicates that the constitutive variable of τ *belongs* to the tuple $\overline{x}$. Alternatively, we will point up this fact through the expression "τ is a bijective term *on* $\overline{x}$".

4.2.2 First-order formulas vs. inductive descriptions

Définition 129 *Let* $\psi_1(\overline{x}), \dots, \psi_s(\overline{x})$ *be first-order formulas with the same variables. The disjunction* $\bigvee_{i=1}^{s} \psi_i(\overline{x})$ *is said* exclusive *if its disjuncts are mutually exclusive, that is, if* $\psi_i(\overline{x})$ *implies* $\neg\psi_j(\overline{x})$, *for all* $1 \leq i < j \leq s$.

We will denote by $\Upsilon_{i\in[s]}\psi_i$ a disjunction to emphasize that it is exclusive. We won't argue for the use of such a writing when its justification is obvious from the context. Clearly, the exclusive disjunction inherits the associativity and commutativity properties from the usual disjunction. In particular, a formula of the form $\Upsilon_{i\in I}\,\Upsilon_{j\in J}\,\psi_i^j$ can be written $\Upsilon_{i\in I,j\in J}\psi_i^j$.

We will need a normalization of quantifier-free formulas (Proposition 112) in order to prove that first-order formulas fulfil a kind of quantifier elimination result over bijective structures (Corollary 113). This normalization involves the notion of *inductive description*. Roughly speaking, an inductive description is a conjunction of bijective literals in which a free variable y can be singled out in such a way that the formula can be written $\psi(\overline{x}) \wedge \theta(\overline{x}, y)$ where ψ is an inductive description and where θ describes in a very simple way the connection between y and the other variables. Let us detail this notion :

Définition 130 *The set of* inductive descriptions*, denoted by* ID*, is the smallest set of quantifier-free conjunctive bijective formulas that contains :*

1. *quantifier-free conjunctive bijective formulas $\delta(x)$ of arity 1 ;*
2. *formulas of arity $d+1$ (with $d>0$) of the form*

$$\varphi(\overline{x}, y) \equiv \psi(\overline{x}) \wedge y = \tau(\overline{x}),$$

   where $\psi(\overline{x})$ is an inductive description of arity d and $\tau(\overline{x})$ is a bijective term built on one variable of $\overline{x}$;

3. *formulas of arity $d+1$ (with $d>0$) of the form*

$$\varphi(\overline{x}, y) \equiv \psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq \tau_i(\overline{x}),$$

   where δ and ψ are inductive descriptions of respective arity 1 and d, the $\tau_i(\overline{x})$ are bijective terms, and $\psi(\overline{x})$ logically implies both $\mathrm{PWD}_{i\in[m]}(\tau_i(\overline{x}))$ and $\bigwedge_{i\in[m]}\delta(\tau_i(\overline{x}))$.

(We abbreviate by $\mathrm{PWD}_{i\in[k]}(t_i)$ the formula $\bigwedge_{i\neq j\in[k]} t_i \neq t_j$ stating that the t_i are pairwise distinct.)

Given two sets $\mathcal{L}$, $\mathcal{L}'$ of formulas and a set $\mathcal{C}$ of structures, we write "$\mathcal{L} \subseteq \mathcal{L}'$ on $\mathcal{C}$" if for all $\varphi \in \mathcal{L}$ there exists $\varphi' \in \mathcal{L}'$ such that $\varphi(S) = \varphi'(S)$ for all $S \in \mathcal{C}$. We write "$\mathcal{L} = \mathcal{L}'$ on $\mathcal{C}$" when both $\mathcal{L} \subseteq \mathcal{L}'$ and $\mathcal{L}' \subseteq \mathcal{L}$ hold on $\mathcal{C}$. Recall that $\mathrm{FO}(\mathrm{qf})$ denotes the set of quantifier-free first-order formulas. Besides, we denote by $\Upsilon\mathrm{ID}$ the set of formulas that are exclusive disjunctions of inductive descriptions.

Proposition 112 *On bijective structures,* $\mathrm{FO}(qf) = \Upsilon\mathrm{ID}$.

Preuve. It is easily seen, by standard methods in logic, that each quantifier-free formula is equivalent to an exclusive disjunction $\Upsilon_{i\in[s]}\varphi_i$ where each φ_i is a conjunctive formula (that is, a conjunction of literals). Furthermore, since the logic ΥID is closed under exclusive disjunction, we can consider without loss of generality that the quantifier-free formula to be written in ΥID is conjunctive. So let φ be a quantifier-free conjunctive formula. We prove that φ is equivalent to an exclusive disjunction of inductive descriptions by induction on its arity. If $\text{arity}(\varphi) = 1$, the result is clear. Otherwise, distinguish one (arbitrary) free variable y in φ and write $\varphi \equiv \varphi(\overline{x}, y)$ with $y \notin \overline{x}$. Now φ has the shape :

$$\varphi \equiv \bigwedge \pm\alpha_i(\overline{x}, y) = \beta_i(\overline{x}, y) \wedge \bigwedge \pm U_i(\gamma_i(\overline{x}, y)) \tag{4.1}$$

where all the α_i, β_i, γ_i's are terms built on one variable of $\overline{x}, y$ and where the U_i's are unary relation symbols. By authorizing *bijective* terms, we can rewrite φ in such a way that each equality (resp. unequality) involving y and $\overline{x}$ has the shape $y = \tau(\overline{x})$ (resp. $y \neq \sigma(\overline{x})$), where τ and σ are now bijective terms on $\overline{x}$. Moreover, by sorting the atomic formulas of φ according to the variables involved in them, we can finally write φ under the form :

$$\varphi \equiv \theta(\overline{x}) \wedge \delta(y) \wedge \bigwedge y = \tau_i(\overline{x}) \wedge \bigwedge y \neq \sigma_i(\overline{x}) \tag{4.2}$$

where both θ and δ are quantifier-free conjunctive bijective formulas. (The conjuncts $U_i(\gamma_i)$ occurring in (4.1) have been incorporated in θ or δ, according to the fact that the term γ_i is built on $\overline{x}$ or on y.)

Two cases occur, according to the emptyness of the conjunction of equalities. **If it is not empty, it contains a conjunct** $y = \tau_j(\overline{x})$. Thus, by substituting the term $\tau_j(\overline{x})$ for each occurrence of y in φ, but that of the conjunct $y = \tau_j(\overline{x})$, we get the equivalent form :

$$\varphi \equiv \psi(\overline{x}) \wedge y = \tau_j(\overline{x})$$

where ψ is a quantifier-free conjunctive bijective formula and $\tau_j(\overline{x})$ is a bijective term. By induction hypothesis, ψ is equivalent to an exclusive disjunction of inductive descriptions, say $\psi \equiv \Upsilon_i \psi_i(\overline{x})$. Thus

$$\varphi \equiv \Upsilon_i \left(\psi_i(\overline{x}) \wedge y = \tau_j(\overline{x})\right)$$

and we are done, since each disjunct above is an inductive description, by item (2) of Definition 130.

If φ does not contain any conjunct of the form $y = \tau(\overline{x})$, Equation (4.2) has the form :

$$\varphi(\overline{x}, y) \equiv \theta(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq \sigma_i(\overline{x}). \tag{4.3}$$

Here, θ and δ are quantifier-free conjunctive bijective formulas and the $\sigma_i(\overline{x})$ are bijective terms. Let's keep in mind this last formula, that has to be written in ΥID. To carry on with the

normalization of φ, we are going to concentrate for a while on its subformula $\delta(y) \wedge \bigwedge_{i\in[m]} y \neq \sigma_i(\overline{x})$. More precisely, we are going to evaluate this subformula with respect to the number of distinct σ_i that satisfy δ. Actually, in order to isolate the heart of our reasonning from minor features, we will focus on formulas of the form $\delta(y)\wedge\bigwedge_{i\in[m]} y \neq v_i$, where the v_i are new first-order variables. Let us first introduce some notations.

For $m > 0$, we denote by $2^{[m]}$ the set of subsets of $[m]$, by $\min P$ the minimal element of a non empty $P \in 2^{[m]}$, and by $\bigcup\mathcal{P}$ the union set $\bigcup_{P\in\mathcal{P}} P$ of any $\mathcal{P} \subseteq 2^{[m]}$. If the elements of some $\mathcal{P} \subseteq 2^{[m]}$ are pairwise disjoint, we say that $\mathcal{P}$ is a *set of disjoint m-subsets* and we note $\mathcal{P} \in \textsc{sds}[m]$. Equivalently, $\mathcal{P} \in \textsc{sds}[m]$ if $\{(P)_{P\in\mathcal{P}}, [m] \setminus \bigcup\mathcal{P}\}$ is a partition of $[m]$.

Let δ be a first-order formula of arity 1 and $\overline{v} = v_1, \ldots, v_m$ be a tuple of first-order variables. A *δ-type for $\overline{v}$* is a formula $\delta^m_{\mathcal{P}}(\overline{v})$, associated with a given $\mathcal{P} \subseteq 2^{[m]}$, that comprehensively describes a particuliar behaviour of the variables with respect to δ. Such a formula asserts that, for any $i, j \in [m]$:

1. If i, j belong to a same $P \in \mathcal{P}$, then v_i, v_j must be interpreted by a same element ;
2. If i, j belong to different components of $\mathcal{P}$, then v_i, v_j must be interpreted by distinct elements ;
3. If $i \in \bigcup\mathcal{P}$, then v_i must be interpreted by some elements that fulfil δ ;
4. If $i \in [m] \setminus \bigcup\mathcal{P}$, then v_i must be interpreted by some element that does not fulfil δ.

For convenience, we formalize these assertions as follows :

$$(i) \quad \bigwedge_{P\in\mathcal{P}} \bigwedge_{i\in P} v_i = v_{\min P} \qquad (ii) \quad \bigwedge_{P\neq Q\in\mathcal{P}} v_{\min P} \neq v_{\min Q}$$

$$(iii) \quad \bigwedge_{P\in\mathcal{P}} \delta(v_{\min P}) \qquad (iv) \quad \bigwedge_{i\in[m]\setminus\bigcup\mathcal{P}} \neg\delta(v_i)$$

and we set : $\delta^m_{\mathcal{P}}(v_1, \ldots, v_m) \equiv (i) \wedge (ii) \wedge (iii) \wedge (iv)$. Now, the following remarks follow from the definition of $\delta^m_{\mathcal{P}}$: First, the formula $\bigvee_{\mathcal{P}\in\textsc{sds}[m]} \delta^m_{\mathcal{P}}(\overline{v})$ is a tautology and its disjuncts are mutually exclusive. Therefore, $\delta(y) \wedge \bigwedge_{i\in[m]} y \neq v_i$ is equivalent to

$$\left(\curlyvee_{\mathcal{P}\in\textsc{sds}[m]}\delta^m_{\mathcal{P}}(\overline{v})\right) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq v_i,$$

and hence to

$$\curlyvee_{\mathcal{P}\in\textsc{sds}[m]} \left(\delta^m_{\mathcal{P}}(\overline{v}) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq v_i\right).$$

Now, each disjunct of the above formula can be written

$$\delta^m_{\mathcal{P}}(\overline{v}) \wedge \delta(y) \wedge \bigwedge_{P\in\mathcal{P}} y \neq v_{\min P}.$$

This is because, under the hypothesis $\delta^m_{\mathcal{P}}(\overline{v}) \wedge \delta(y)$, each v_i such that $i \notin \bigcup \mathcal{P}$ clearly differs from y, since it doesn't satisfy δ while y does, and because for each $P \in \mathcal{P}$, the sets $\{v_i, i \in P\}$ and $\{v_{\min P}\}$ coincide. Therefore we get :

$$\delta(y) \wedge \bigwedge_{i\in[m]} y \neq v_i \;\sim\; \Upsilon_{\mathcal{P}\in \text{SDS}[m]} \left(\delta^m_{\mathcal{P}}(\overline{v}) \wedge \delta(y) \wedge \bigwedge_{P\in\mathcal{P}} y \neq v_{\min P} \right) \tag{4.4}$$

Notice that $\delta^m_{\mathcal{P}}(\overline{v})$ implies $\text{PWD}_{P\in\mathcal{P}}(v_{\min P}) \wedge \bigwedge_{P\in\mathcal{P}} \delta(v_{\min P})$ (immediately follows from the definition of $\delta^m_{\mathcal{P}}$).

Let us now come back to the initial formula $\varphi(\overline{x}, y)$ described in (4.3), that we aim to normalize. By (4.4), it can be written :

$$\theta(\overline{x}) \wedge \Upsilon_{\mathcal{P}\in \text{SDS}[m]} \left(\delta^m_{\mathcal{P}}(\sigma_1(\overline{x}), \ldots, \sigma_m(\overline{x})) \wedge \delta(y) \wedge \bigwedge_{P\in\mathcal{P}} y \neq \sigma_{\min P}(\overline{x}) \right),$$

or equivalently :

$$\Upsilon_{\mathcal{P}\in \text{SDS}[m]} \left(\theta(\overline{x}) \wedge \delta^m_{\mathcal{P}}(\sigma_1(\overline{x}), \ldots, \sigma_m(\overline{x})) \wedge \delta(y) \wedge \bigwedge_{P\in\mathcal{P}} y \neq \sigma_{\min P}(\overline{x}) \right).$$

Let us denote by $\psi_{\mathcal{P}}(\overline{x})$ the formula $\theta(\overline{x}) \wedge \delta^m_{\mathcal{P}}(\sigma_1(\overline{x}), \ldots, \sigma_m(\overline{x}))$. By induction hypothesis, $\psi_{\mathcal{P}}$ can be written as an exclusive disjunction of inductive descriptions, say $\Upsilon_{i\in[k]} \psi^i_{\mathcal{P}}$. Of course, each $\psi^i_{\mathcal{P}}$ implies $\psi_{\mathcal{P}}$, and hence $\delta^m_{\mathcal{P}}(\sigma_1(\overline{x}), \ldots, \sigma_m(\overline{x}))$. Therefore, according to the remark that follows Equation (4.4), each $\psi^i_{\mathcal{P}}$ implies $\text{PWD}_{P\in\mathcal{P}}(\sigma_{\min P}(\overline{x})) \wedge \bigwedge_{P\in\mathcal{P}} \delta(\sigma_{\min P}(\overline{x}))$. Finally, the initial formula (4.3) can be written

$$\Upsilon_{\mathcal{P}\in \text{SDS}[m]}\, \Upsilon_{i\in[k]} \left(\psi^i_{\mathcal{P}}(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{P\in\mathcal{P}} y \neq \sigma_{\min P}(\overline{x}) \right)$$

where each $\psi^i_{\mathcal{P}}$ is an inductive description that implies both $\text{PWD}_{P\in\mathcal{P}}(\sigma_{\min P}(\overline{x}))$ and $\bigwedge_{P\in\mathcal{P}} \delta(\sigma_{\min P}(\overline{x}))$. The conclusion follows from item (3) of Definition 130. □

Corollaire 113 *For each $(\varphi, S) \in \text{FO} \times \text{BIJ}$, there exists $\varphi' \in \Upsilon\text{ID}$ such that $\varphi'(S) = \varphi(S)$. Furthermore, there is some function f such that $|\varphi'| \leq f(|\varphi|)$ (upper bound independent of S) and φ' is computable from (φ, S) in time $f(|\varphi|)|S|$.*

Preuve.

We can assume, without loss of generality, that the formula φ of the statement has the form $\exists z \gamma(\overline{x}, z)$, where γ is a quantifier-free formula. By Proposition 112, such a γ can be written as a disjunction of inductive descriptions. Now, recall the three forms that a formula of ID can fit, according to Definition 130 :

(1) $\delta(y)$ with $\text{arity}(\delta) = 1$;

(2) $\psi(\overline{x}) \wedge y = \tau(\overline{x})$, where $\psi(\overline{x}) \in \text{ID}$;

(3) $\psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq \tau_i(\overline{x})$, where $\delta, \psi \in \text{ID}$ and $\psi(\overline{x})$ logically implies $\text{PWD}_{i\in[m]}(\tau_i(\overline{x})) \wedge \bigwedge_{i\in[m]} \delta(\tau_i(\overline{x}))$.

Moreover, although it is not emphasized in the statement of Proposition 112, this rewriting of γ in Υ_{ID} can be done in such a way that in each disjunct that fits the form (2) or (3) above, the variable y singularized in both these forms can be arbitrarily chosen[6]. In particular, when we write $\exists z \gamma(\overline{x}, z)$ as $\exists z \bigvee_i \gamma_i$, with the γ_i's in ID, we can ensure that in each γ_i of the form (2) or (3), the singularized variable y coincides with z. Consequently, the formula $\exists z \gamma(\overline{x}, z)$ can be written $\bigvee_i \exists y \gamma_i(\overline{x}, y)$, where each γ_i fits one of the forms (1), (2) or (3) of Definition 130, with y as the singularized variable.

Given a bijective structure S, it is now easy to translate each formula $\exists y \gamma_i(\overline{x}, y)$ into a quantifier-free formula $\tilde{\gamma}_i(\overline{x})$, equivalent on S. This is done according to the form of γ_i :

(1) $\exists y \delta(y)$ becomes false if $\delta(S) = \emptyset$, true otherwise ;

(2) $\exists y \left(\psi(\overline{x}) \wedge y = \tau(\overline{x})\right)$ becomes $\psi(\overline{x})$;

(3) $\exists y \left(\psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq \tau_i(\overline{x})\right)$ becomes false if $\delta(S)$ contains less than m elements, $\psi(\overline{x})$ otherwise.

The time complexity of this procedure amounts to that of computing $card(\delta(S))$ for some first-order formula $\delta(y)$ of arity 1, which is $O(|\delta|.|S|)$. Hence, the time of the transformation is $O(|S|)$, for a fixed formula. So, φ can be transformed into a quantifier-free formula φ' so that $\varphi(S) = \varphi'(S)$, and, clearly, one gets $|\varphi'| \leq g(|\varphi|)$, for some function g. We conclude by applying again Proposition 112. □

4.3 Complexity results on bijective structures

The enumeration, counting and j^{th} solution problems are very easy for queries defined by inductive descriptions as the following proposition asserts.

Proposition 114 *For each function $\varphi \in ID$,*

1. $\text{COUNT}(\varphi, \text{BIJ})$ *is computable in linear time ;*
2. $\text{J}^{th}(\varphi, \text{BIJ}, <_{lex}) \in \text{CONSTANT-TIME}_{lin}$;
3. $\text{ENUM}(\varphi, \text{BIJ}, <_{lex}) \in \text{CONSTANT-DELAY}_{lin}$.

Preuve. Thanks to Lemma 15, we just have to prove *1* and *2*. The algorithms described in this proof are recursive. Their behaviour on an input (φ, S) (resp. (φ, S, j)) is directed by the shape of

[6] This possibility is explicitly mentioned in the first paragraph of the proof of Proposition 112 : when $\text{arity}(\varphi) > 1$, the normalization procedure "distinguishes one (arbitrary) free variable y and writes $\varphi \equiv \varphi(\overline{x}, y)$ with $y \notin \overline{x}$"

the inductive description φ. That is, the algorithms operate according to modalities that depend on the form (1), (2) or (3) of Definition 130 that φ matches.

1. **COUNT$(\varphi, \textbf{BIJ})$.** The following recursive procedure computes $card(\varphi(S))$ in time $O(|S|)$ for any $\varphi \in$ ID and $S \in$ BIJ. As mentioned above, it uses the inductive definition of ID.

Algorithm 20 COUNT(φ, S)

si arity$(\varphi) = 1$ **alors**
  calculer $\varphi(S)$
  return $card(\varphi(S))$
fin si
si $\varphi \equiv \psi(\overline{x}) \wedge y = \tau(\overline{x})$ **alors**
  return COUNT(ψ, S)
fin si
si $\varphi \equiv \psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i<m} y \neq \tau_i$ **alors**
  return COUNT$(\psi, S) *$ (COUNT$(\delta, S) - m$)
fin si

The last instruction of this procedure is justified by the fact that ψ fulfils the implication $\psi(\overline{x}) \rightarrow \text{PWD}_{i\in[m]}(\tau_i) \wedge \bigwedge_{i\in[m]} \delta(\tau_i)$.

2. **$\text{J}^{th}(\varphi, \textbf{BIJ}, <_{\textbf{lex}})$.** We design an algorithm that, given $\varphi \in$ ID and $S \in$ BIJ, performs a preprocessing step in time $O(|S|)$ in such a way that $\mathsf{jth}(\varphi, S, <_{\text{lex}}, j)$ can be afterwards computed in constant time for any $j \in \mathbb{N}$. Both the preprocessing and the output phase are recursive. The following easy considerations will make the proof of the algorithm immediate.

(1) If arity$(\varphi) = 1$, then computing and sorting $\varphi(S)$ can be done in time $O(|S|)$. Afterwards, $\mathsf{jth}(\varphi, S, <_{\text{lex}}, j)$ can be returned in constant time for any j.

(2) If $\varphi(\overline{x}, y) \equiv \psi(\overline{x}) \wedge y = \tau(\overline{x})$, then $\mathsf{jth}(\varphi, S, <_{\text{lex}}, j) = (\overline{a}, \tau(\overline{a}))$ where $\overline{a} = \mathsf{jth}(\psi, S, <_{\text{lex}}, j)$.

(3) If $\varphi(\overline{x}, y) \equiv \psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i\leq m} y \neq \tau_i(\overline{x})$, then the result rests on the two following remarks :

• If $(A, <_A)$ and $(B, <_B)$ are two linear orders and $<_{\text{lex}}$ is the associated lexicographic ordering for $A \times B$, then for any $j < card(A).card(B)$:

$$\mathsf{jth}(A \times B, <_{\text{lex}}, j) = (\mathsf{jth}(A, <_A, q), \mathsf{jth}(B, <_B, r)),$$

where $q = j \operatorname{div} card(B)$ and $r = j \bmod card(B)$.

• Let $(A, <)$ be a linear order and F a subset of A. Denote by $a_0, \ldots, a_{p-1}$ (resp. $f_0, \ldots, f_{m-1}$) the strictly increasing enumeration of A (resp. F) according to $<$. Then for any $r < p - m$:

$$\mathsf{jth}(A \setminus F, <, r) = \mathsf{jth}(A, <, s),$$

where s is the result of the following procedure :

$$s \leftarrow r;\ i \leftarrow 0;\ \textbf{while}\ (f_i \leq a_s \text{ and } i < m)\{i\text{++};\ s\text{++}\};\ \textbf{return}\ s.$$

The precomputation and output steps of our algorithm, denoted JTH, for the problem $\text{J}^{th}(\varphi, \text{BIJ}, <_{\text{lex}})$ are described below. We use some instruction of the form $T \leftarrow \mathsf{sort}\, X$. It means "sort the set X and store it in increasing order in a table $T[0, \ldots, card(X) - 1]$". The soundness of JTH proceeds from the above remarks. Besides, it is clearly a CONSTANT-TIME$_{lin}$ algorithm.

Algorithm

Algorithm 21 JTH.**precomp**(φ, S)

calculer $card(\varphi(S))$
si φ est une formule δ d'arité 1 **alors**
  calculer $\delta(S)$; $\Delta \leftarrow \mathsf{sort}\, \delta(S)$
fin si
si $\varphi \equiv \psi(\overline{x}) \wedge y = \tau(\overline{x})$ **alors**
  JTH.**precomp**(ψ, S)
fin si
si $\varphi \equiv \psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i<m} y \neq \tau_i$ **alors**
  JTH.**precomp**(δ, S) ; JTH.**precomp**(ψ, S)
fin si

□

Corollaire 115 *For each* $\varphi \in$ FO, ENUM$(\varphi, \text{BIJ}, <_{lex}) \in$ CONSTANT-DELAY$_{lin}$.

Preuve. Derives from Corollary 113, Lemma 18 and Proposition 114. □

Remarque 28 *This corollary improves Theorem 7 of [DG07] which states a similar result for some unspecified linear order* $<$ *instead of* $<_{lex}$.

We are not able to prove that, for each $\varphi \in$ FO, problem $\text{J}^{th}(\varphi, \text{BIJ}, <_{\text{lex}})$ belongs to the class CONSTANT-TIME$_{lin}$ and we conjecture it does not. However, a similar result holds for a less canonical linear order that depends on the input formula φ.

Définition 131 *Let* $\varphi(\overline{x}) \equiv \Upsilon_{i\in[s]}\varphi_i(\overline{x})$ *be an exclusive disjunction of inductive descriptions* $\varphi_i(\overline{x})$ *of signature* σ *and let* S *be a* σ*-structure. Let* $<^{\varphi}_{lex}$ *denote the "lexicographical" order on* $\varphi(S)$ *defined as follows : for any* $\overline{a}, \overline{b} \in \varphi(S)$ *:* $\overline{a} <^{\varphi}_{lex} \overline{b}$*, if*

- *either* $\overline{a} \in \varphi_i(S)$ *and* $\overline{b} \in \varphi_j(S)$ *for some* $i < j \leq s$,
- *or* $\overline{a}, \overline{b} \in \varphi_i(S)$ *for some* $i \leq s$ *and* $\overline{a} <_{lex} \overline{b}$.

Algorithm 22 JTH.**output**(φ, S, j)

si $j \geq card(\varphi(S))$ **alors**
  return error
sinon
  si φ est une formule δ d'arité 1 **alors**
    return $\Delta[j]$
  fin si
  si $\varphi \equiv \psi(\overline{x}) \wedge y = \tau(\overline{x})$ **alors**
    $\overline{a} \leftarrow$ JTH.**output**(ψ, S, j)
    return $(\overline{a}, \tau(\overline{a}))$
  fin si
  si $\varphi \equiv \psi(\overline{x}) \wedge \delta(y) \wedge \bigwedge_{i\in[m]} y \neq \tau_i$ **alors**
    $\beta \leftarrow card(\delta(S)) - m$
    $\overline{a} \leftarrow$ JTH.**output**$(\psi, S, j \operatorname{div} \beta)$
    $\Theta \leftarrow \mathsf{sort}\,\{\tau_1(\overline{a}), \ldots, \tau_m(\overline{a})\}$
    $i \leftarrow 0\,;\ s \leftarrow j \bmod \beta\,;$
    tant que $i < m$ et $\Theta[i] \leq \Delta[s]$ **faire**
      $\{i{+}{+},\ s{+}{+}\}$
    fin tant que
    return $(\overline{a}, \Delta[s])$
  fin si
fin si

Corollaire 116 *Let $\varphi \in \Upsilon\text{ID}$. Then $\text{J}^{th}(\varphi, \text{BIJ}, <^{\varphi}_{lex}) \in \text{CONSTANT-TIME}_{lin}$.*

Preuve. For any exclusive disjunction of inductive descriptions $\varphi \equiv \Upsilon_{i\in[1,s]}\varphi_i$ and any $S \in \text{BIJ}$, it holds : $\mathsf{jth}(\varphi, S, <^{\varphi}_{\text{lex}}, j) = \mathsf{jth}(\varphi_i, S, <_{\text{lex}}, k)$, where i is the smallest index such that $j < \sum_{u=1}^{i} card(\varphi_u(S))$ and where $k = j - \sum_{u=1}^{i-1} card(\varphi_u(S))$. Together with the algorithm JTH for $\text{J}^{th}(\varphi, \text{BIJ}, <_{\text{lex}})$ described above (see page 141) for $\varphi \in \text{ID}$, this remark justifies the following algorithm $\mathcal{A}$ for $\text{J}^{th}(\varphi, \text{BIJ}, <^{\varphi}_{\text{lex}})$ with $\varphi \in \Upsilon\text{ID}$:

Algorithm 23 $\mathcal{A}$**.precomp**(φ, S)

écrire φ sous la forme $\Upsilon_{i\in[s]}\varphi_i$, avec chaque $\varphi_i \in \text{ID}$
pour $i = 1$ à s **faire**
  JTH.**precomp**(φ_i, S)
  calculer $v_i = \sum_{j\in[i]} card(\varphi_j(S))$
fin pour

Algorithm 24 $\mathcal{A}$**.output**(φ, S, j)

si $j \geq v_s$ **alors**
  return error
sinon
  trouver $i \leq s$ tel que $v_{i-1} \leq j < v_i$
  return JTH.**output**$(\varphi_i, S, j - v_{i-1})$
fin si

It is easily seen that $\mathcal{A}$ runs in $\text{CONSTANT-TIME}_{lin}$. □

4.4 Complexity results on structures of bounded degree

Let S be a structure of domain D over a relational signature σ. We say that the *degree* of S is *bounded by* d if, for each symbol $R \in \sigma$, each $a \in D$ belongs to at most d tuples of R. We denote by DEG_d the class of relational structures of degree bounded by d. The complexity results we obtain for our query problems on bijective structures immediately imply the same results on structures of bounded degree because of the following lemma of [DG07] :

Lemme 117 (Durand, Grandjean) *Let $d > 0$ be a fixed integer. For each $(\varphi, S) \in \text{FO}\times\text{DEG}_d$, there exists $(\varphi', S') \in \text{FO}\times\text{BIJ}$ such that $\varphi(S) = \varphi'(S')$. Moreover, the size $|\varphi'|$ only depends on $|\varphi|$ and d, and (φ', S') is computable from (φ, d, S) in time $O(|S|)$, that means, in time $\leq f(|\varphi|, d).|S|$, for some function f.*

We now give a proof of this Lemma.

Interpreting a structure of bounded degree into a bijective structure

In this subsection, we present a linear **FO**-interpretation from DEG_d to BIJ.

Let $\mathcal{S} = \langle D; R_1, \ldots, R_q \rangle$ be a ρ-structure of domain D, of arity $m = max_{i\in[q]} arity(R_i)$ and of degree bounded by some constant d. One associates to $\mathcal{S}$ a bijective σ-structure $\mathcal{S}' = \langle D'; D, T_1, \ldots, T_q, g, f_1, \ldots, f_m \rangle$ of domain D' where $D, T_1, \ldots, T_q$ are pairwise disjoints unary relations (i.e. subsets of D') and $g, f_1, \ldots, f_m$ are permutations of D'. Structure $\mathcal{S}'$ is precisely defined as follows :

- D corresponds to the domain of $\mathcal{S}$.
- T_i ($i \in [q]$) is a set of elements each representing a tuple of R_i (hence, $card(T_i) = card(R_i)$). The new domain D' is the disjoint union : $D \cup (D \times \{1, \ldots, d\}) \cup T_1 \cup \cdots \cup T_q$. Let us use the following convenient abbreviations : $U = D \cup (D \times \{1, \ldots, d\})$ and $T = \bigcup_{i\in[q]} T_i$.
- g creates a cycle that relates d copies of each element x of the domain. More precisely, for each $x \in D$, it holds $g(x) = (x, 1)$, $g((x, i)) = (x, i + 1)$ for $1 \leq i < d$, and $g((x, d)) = x$. We also set $g(x) = x$ for all other x ($x \in T$).
- Each f_i is an involutive permutation and essentially represents a projection of T into D as follows. Let $R_i(x_1, \ldots, x_k)$ be true in $\mathcal{S}$ for some relation R_i of arity $k \leq m$ and some k-tuple $(x_1, \ldots, x_k) \in D^k$. Suppose $R_i(x_1, \ldots, x_k)$ is represented by element $t \in T_i$, then, for each $j \in [k]$, set $f_j(t) = (x_j, h)$ and set the reciprocal $f((x_j, h)) = t$ if $R(x_1, \ldots, x_k)$ is the h^{th} tuple in which x_j appears (with $h \leq d$). The construction is completed by loops $f_j(x) = x$ for all other $x \in D'$.

An instance of this reduction is given in the figure below. The original structure (digraph) of degree 3 is on the right side of the figure and the associated bijective structure is on the left side.

It is clear that, by construction, $\mathcal{S}'$ is a bijective structure and that we have the following interpretation Lemma.

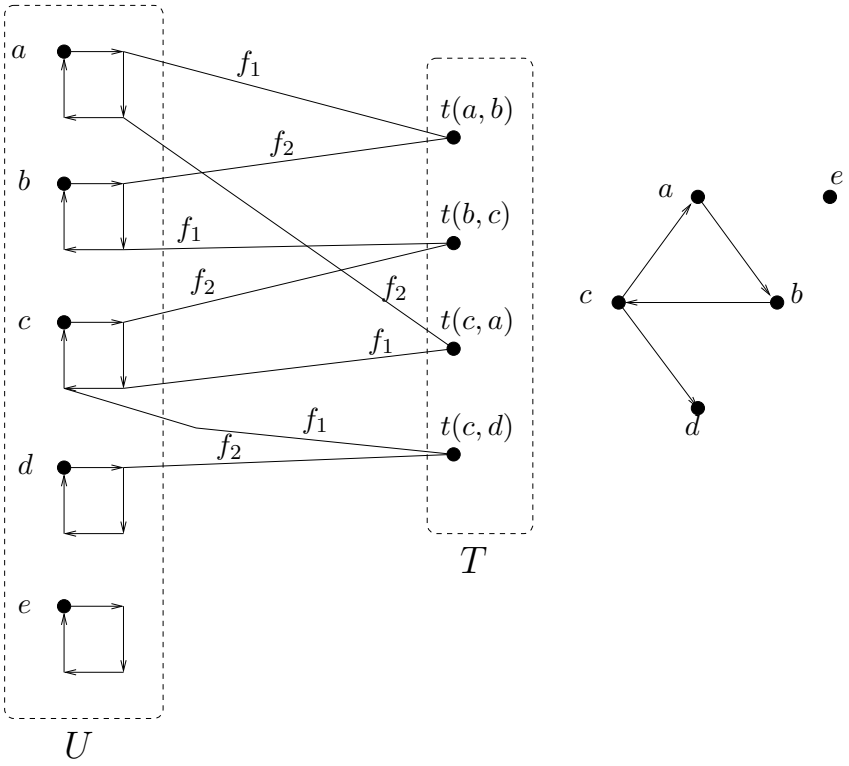
Lemme 118 *Let θ_i be the σ-formula below associated to any symbol $R_i \in \rho$ of arity k :*

$$\theta_i(x_1, \ldots, x_k) \equiv \exists t (T_i(t) \wedge \bigwedge_{j\in[k]} \bigvee_{h\in[d]} f_j(t) = g^h(x_j)).$$

Then, for all $(a_1 \ldots, a_k) \in D^k$:

$$(\mathcal{S}, a_1, \ldots, a_k) \models R_i(x_1, \ldots, x_k) \iff (\mathcal{S}', a_1, \ldots, a_k) \models \theta_i(x_1, \ldots, x_k).$$

To each first-order ρ-formula $\varphi(x_1, \ldots, x_p)$, one associates the σ-formula $\varphi''(x_1, \ldots, x_p)$ obtained by replacing each quantification $\exists v$ (resp. $\forall v$) by the relativized quantification $(\exists v D(v))$ (resp. $(\forall v D(v))$) (that can be written respectively as $\exists v(D(v) \wedge ...)$ and $\forall v(D(v) \rightarrow ...)$) and by replacing each subformula $R_i(x_1, \ldots, x_k)$ by $\theta_i(x_1, \ldots, x_k)$.


a
b
c
d
e
U
f_1
f_2
f_1
f_2
f_2
f_1
f_1
f_2
$t(a,b)$
$t(b,c)$
$t(c,a)$
$t(c,d)$
T
a
e
c
b
d

The following proposition and lemma express that our reduction is correct and linear in $|\mathcal{S}|$. Because of Lemma 118, Proposition 119 can be easily proved by induction on formula φ.

Proposition 119 (interpretation of $\mathcal{S}$ into $\mathcal{S}'$) *For all $(x_1 \ldots, x_p) \in D^p$:*

$$(\mathcal{S}, a_1, \ldots, a_p) \models \varphi(x_1, \ldots, x_p) \iff (\mathcal{S}', a_1, \ldots, a_p) \models \varphi''(x_1, \ldots, x_p).$$

In other words : $\varphi(\mathcal{S}) = \varphi''(\mathcal{S}') \cap D^p$. Then, setting $\varphi'(x_1, \ldots, x_p) \equiv \varphi''(x_1, \ldots, x_p) \wedge \bigwedge_{i\in[p]} D(x_i)$, it holds : $\varphi(\mathcal{S}) = \varphi'(\mathcal{S}')$

Lemme 120 *Computing $\mathcal{S}'$ from $\mathcal{S}$ can be done in linear time $O_{\rho,d}(|\mathcal{S}|)$.*

Preuve. As computing $\mathcal{S}'$ from $\mathcal{S}$ is easy, one has only to compare the size of the two structures. The size of $\mathcal{S}$ is :

$$|\mathcal{S}| = \Theta(|D| + \sum_{i\in[q]} card(R_i).arity(R_i)) = \Theta_\rho(|D| + \sum_{i\in[q]} card(R_i)).$$

For $\mathcal{S}'$, by construction, it holds that :

$$|D'| = (d+1).|D| + \sum_{i\in[q]} card(R_i) = \Theta_{d,\rho}(|\mathcal{S}|).$$

Hence, $|\mathcal{S}'| = \Theta(m|D'|) = \Theta_{d,\rho}(|\mathcal{S}|)$. □

So, we have proved Lemma 117. From Lemma 117 and Corollaries 115 and 116, we immediately deduce the following

Corollaire 121 *For each $d > 0$ and each $\varphi \in$ FO, we have :*

1. $\text{ENUM}(\varphi, \text{DEG}_d, <_{lex}) \in$ CONSTANT-DELAY$_{lin}$;
2. $\text{J}^{th}(\varphi, \text{DEG}_d, <^{\varphi,d}) \in$ CONSTANT-TIME$_{lin}$, *for some linear order $<^{\varphi,d}$;*

4.5 Conclusion

Suite aux deux résultats de ce chapitre, plusieurs questions restent encore ouvertes :

Peut-on trouver la j-ème solution d'une requête **FO** sur une structure de degré borné dans l'ordre lexicographique, et toujours avec la même complexité CONSTANT-DELAY$_{lin}$? Nous pensons pouvoir obtenir ce résultat pour un ordre lexicographique dont l'ordre du domaine sous-jacent est précalculé mais dépend de la formule traitée.

Frick et Grohe [FG04b] ont prouvé que l'on peut décider le problème $\text{MC}(\varphi, \text{DEG}_d)$ en temps $f(|\varphi|)\cdot|\mathcal{M}|$ pour une fonction f élémentaire (exponentielle triple). Peut-on obtenir aussi le même résultat de complexité pour la phase de précalcul de l'énumération d'un schéma d'énumération

de $\textsc{Enum}(\varphi, \textsc{deg}_d)$ de complexité $\textsc{Constant-Delay}_{lin}$? Le schéma présenté ici possède un précalcul en temps $f(|\varphi|)|\mathcal{M}|$ pour une fonction f *non élémentaire*.

Toute classe de structures de degré borné est trivialement de largeur arborescente localement bornée. On a donc envie de se poser la question suivante : peut-on généraliser les deux résultats de ce chapitre (Corollaire 121) aux classes de structures de largeur arborescente localement bornée ?

Chapitre 5

Classes de graphes localement décomposables, graphes d'intervalles et requêtes logiques

Sommaire

5.1 Introduction

Courcelle et al [CER90] ont défini une généralisation des classes de graphes de *largeur arborescente bornée* (classes notées TW) : les classes de graphes de *largeur de clique bornée* (classes

notées CW). Plus récemment, Frick et Grohe [FG01b] ont introduit une autre généralisation des classes TW : les classes de graphes (ou de structures) de *largeur arborescente localement bornée* (notée LTW). Nous nous intéressons ici à la notion qui généralise naturellement les deux classes CW et LTW : les classes de graphes de *largeur de clique localement bornée* (LCW).

En quoi cette notion est-elle intéressante et, en premier lieu, peut-on exhiber une classe naturelle de graphes qui soit LCW mais ne soit ni CW, ni LTW ? Nous démontrons que la classe des graphes d'intervalles unitaires, notée UIG, satisfait ces trois conditions. D'abord, il est évident que la classe UIG, qui contient les graphes complets, n'est pas, pour cette raison, de largeur arborescente localement bornée : les graphes complets ont un diamètre 1 et une largeur arborescente non bornée. Par ailleurs, Brandstadt et al. [BLM05] ont établi que la classe UIG n'est pas de largeur de clique bornée. Dans ce chapitre, nous démontrons que la classe UIG est de largeur de clique localement bornée, c'est-à-dire UIG $\in$ LCW. Notre résultat découle de la borne suivante : pour tout graphe d'intervalles unitaires G, on a $\mathrm{cw}(G) \leq \mathrm{diam}(G) + 2$.

Notons que Kante [Kan08] montre le résultat suivant : si une classe de graphes $\mathcal{G}$ est localement clique-décomposable alors pour tout $k > 0$, la classe $\mathcal{G}^k$ des graphes obtenus à partir d'un graphe $G \in \mathcal{G}$ en ajoutant des arêtes $\{x, y\}$ pour toute paire de sommet x, y à distance au plus k est elle-même localement clique-décomposable. C'est notamment le cas pour les graphes planaires, ce qui donne un second exemple (moins naturel) de classe de graphes localement clique-décomposable mais non localement arbre-décomposable.

En second lieu, nous généralisons une notion de Frick et Grohe [FG01b] et, pour cela, nous adaptons un résultat de ces auteurs pour notre généralisation. Plus précisément, Frick et Grohe ont introduit les classes de graphes *linéairement et localement arbre-décomposables* et ont prouvé qu'évaluer un énoncé **FO** fixé sur une telle classe de graphes se fait en temps linéaire. Nous définissons la notion de classe de graphes *linéairement et localement clique-décomposable*, notion qui généralise celle de Frick et Grohe ; nous affirmons que, par une adaptation de leur argument, on obtient la même complexité linéaire pour toute classe de graphes linéairement et localement clique-décomposable. Comme nous établissons par ailleurs que la classe UIG est linéairement et localement clique-décomposable, il s'en déduit que tout énoncé **FO** fixé s'évalue en temps linéaire sur tout graphe d'intervalles unitaires.

Enfin nous montrons que ce résultat de complexité n'est pas généralisable dans le sens suivant :

- le résultat ne peut pas s'étendre à **MSO** : nous prouvons qu'évaluer une requête **MSO** sur les graphes d'intervalles unitaires est aussi difficile que sur les structures quelconques ;
- le résultat ne peut pas s'étendre aux graphes d'intervalles (quelconques) : nous établissons qu'évaluer une requête **FO** sur les graphes d'intervalles est aussi difficile que sur les structures quelconques.

5.2 Préliminaires

Nous définissons d'abord les principales notions que nous étudions dans ce chapitre : d'un côté, les graphes d'intervalles et les graphes d'intervalles unitaires ; de l'autre, les classes de graphes de largeur de clique localement bornée.

Définition 132 (graphe d'intervalles, graphe d'intervalles unitaires) *Un graphe $G = (V, E)$ est appelé* graphe d'intervalles *(*IG : Interval Graph*) s'il existe un ensemble S d'intervalles réels $i = [b_i, e_i]$, $b_i < e_i$, et une fonction surjective $I : V \to S$ tels que, pour chaque paire de sommets distincts x, y de G, on ait $\{x, y\} \in E$ si et seulement si $I(x) \cap I(y) \neq \emptyset$. La paire (S, I) ou, plus simplement, la fonction I est appelée* représentation par intervalles *de G. Si de plus S est un ensemble d'intervalles unitaires (c'est-à-dire de longueur 1) alors G est appelé* graphe d'intervalles unitaires *(*UIG : Unit Interval Graph*).*

Remarque 29 *On utilisera la caractérisation suivante, prouvée dans [CK87, CKN+95] : un graphe G est un graphe d'intervalles unitaires si et seulement si il admet une représentation par intervalles I qui est* propre*, ce qui signifie qu'on n'a jamais $b_{I(y)} < b_{I(x)} \leq e_{I(x)} < e_{I(y)}$.*

Remarque 30 *On sait tester [CK87] si un graphe G est un graphe d'intervalles unitaires et, si oui, calculer une représentation de G par intervalles unitaires, ceci en temps linéaire.*

Définition 133 (largeur de clique localement bornée) *Une classe de graphes $\mathcal{G}$ est dite de largeur de clique localement bornée s'il existe une fonction calculable f telle que, pour tout graphe $G \in \mathcal{G}$, tout sommet $x \in V(G)$ et tout entier $k \geq 0$, on a $cw(N_G^k(x)) \leq f(k)$.*

5.3 Les graphes d'intervalles unitaires ont une largeur de clique localement bornée

Soit G un graphe d'intervalles unitaires et soit I une représentation de G. On suppose, sans perte de généralité, qu'il n'existe pas deux sommets distincts x et y tels que $I(x) = I(y)$. En effet, si une représentation I admet deux sommets x et y avec $I(x) = I(y)$, on peut construire une nouvelle représentation I' en décalant légèrement $I(y)$ sur la droite. Par conséquent, on peut définir un ordre linéaire $\leq_I$ des sommets de G tel que $x \leq_I y$ si et seulement si $b_I(x) \leq b_I(y)$.

5.3.1 Représentation en couches

Nous introduisons la représentation suivante qui sera pour nous un outil essentiel dans l'analyse des graphes d'intervalles unitaires. Remarquons que cette représentation est semblable à celle utilisée par Lozin [Loz07, Loz08].

Définition 134 (représentation en couches) *Etant donnée une représentation I du graphe d'intervalles unitaires $G = (V, E)$, on peut construire, en temps linéaire, une unique partition P de V telle que*

- *$P = (A_1, \ldots, A_k)$ où chaque "couche" A_i est l'ensemble $\{a_{i,1}, \ldots a_{i,j_i}\}$ tel que $(a_{1,1}, \ldots, a_{1,j_1}, a_{2,1}, \ldots, a_{2,j_2}, \ldots, a_{k,1}, \ldots, a_{k,j_k})$ est la liste strictement croissante des sommets de V selon l'ordre $\leq_I$;*
- *Pour tout $i \leq k$, A_i est une clique de G ;*
- *Pour tout $i < k$, $\{a_{i,1}, a_{i+1,1}\} \notin E$.*

On désigne par $c(x)$ la colonne ou couche de x, i.e. $c(x) = i$ si et seulement si $x \in A_i$. On définit une fonction (partielle) $b : V \to V$, appelée fonction "pont" ("bridge"), telle que, pour chaque sommet $x \in V$, $b(x)$ est le plus grand sommet y par rapport à $\leq_I$ tel que $\{x, y\} \in E$ et $c(y) = c(x) + 1$. La fonction b n'est pas définie en x si un tel y n'existe pas et on notera alors $b(x) = \perp$. Le tuple $\mathcal{L} = (V, \leq_I, b, c)$ est appelé représentation en couches de G ($\mathcal{L}$ pour "Layers").

On note $nc(\mathcal{L})$ (nombre de colonnes de $\mathcal{L}$) la cardinalité k de P.

Soit $\mathcal{L} = (V, \leq_I, b, c)$ une représentation en couches d'un graphe d'intervalles unitaires G. On définit la *distance de colonnes* entre deux sommet x et y :

$$d_c(x, y) = |c(x) - c(y)|$$

Il est essentiel de remarquer que la représentation en couches $\mathcal{L}$ peut être construite en temps linéaire à partir de la représentation I de $G = (V, E)$. La propriété suivante exprime que $\mathcal{L}$ contient toute l'information sur le graphe d'intervalles unitaires $G = (V, E)$.

Propriété 122 *Pour tous sommets $x, y \in V$, on a les implications suivantes :*

1. $\{x, y\} \in E \Rightarrow d_c(x, y) \leq 1$;
2. $d_c(x, y) = 0 \Rightarrow \{x, y\} \in E$;
3. $c(y) = c(x) + 1 \Rightarrow (\{x, y\} \in E \Leftrightarrow y \leq_I b(x))$.

Preuve. (1) : Par l'absurde : Supposons qu'il existe une arête $\{x, y\} \in E$ reliant un sommet $x \in A_i$ à un sommet $y \in A_j$ pour $j > i + 1$. Cela implique $I(x) \cap I(y) \neq \emptyset$ et donc, pour tous z, t tels que $x \leq_I z \leq_I t \leq_I y$, $I(z) \cap I(t) \neq \emptyset$. C'est vrai en particulier pour le premier élément $z = a_{i+1,1}$ de A_{i+1} et le premier élément $t = a_{i+2,1}$ de A_{i+2}, autrement dit : $\{a_{i+1,1}, a_{i+2,1}\} \in E$. Ce qui contredit la définition de la partition P.

(2) : Ce point reformule le fait que chaque A_i est une clique de G.

(3) : Supposons $c(y) = c(x) + 1$. L'implication $\{x, y\} \in E \Rightarrow y \leq_I b(x)$ se déduit de la définition de $b(x)$. L'implication réciproque découle du fait que si on a $x \leq_I y \leq_I b(x)$ alors, à cause de $\{x, b(x)\} \in E$, on a aussi $\{x, y\} \in E$. □

On a les propriétés suivantes :

Lemme 123 *Soient $G = (V, E)$ un graphe d'intervalles unitaires connexe et $\mathcal{L} = (V, \leq_I, b, c)$ une représentation en couches de G. On a les deux propriétés suivantes :*

1. *Pour chaque paire de sommets x et y, $d_c(x, y) \leq dist(x, y) \leq 2d_c(x, y) + 1$;*
2. *$nc(\mathcal{L}) - 1 \leq diam(G) \leq 2nc(\mathcal{L}) - 1$.*

Preuve. Soient x et y deux sommets tels que $\text{dist}(x, y) = k$. Considérons un chemin $P = (x_1 = x, x_2, \ldots x_k, x_{k+1} = y)$ de longueur k entre x et y. Puisque $\{x_i, x_{i+1}\} \in E$, on a $d_c(x_i, x_{i+1}) \leq 1$, pour chaque $i \leq k$, par l'assertion 1 de la propriété précédente. On a donc :

$$d_c(x, y) \leq \sum_{i=1}^{k} d_c(x_i, x_{i+1}) \leq k = \text{dist}(x, y)$$

Considérons maintenant deux sommets x et y tels que $d_c(x, y) = h$. On supposera $c(x) = j$ et $c(y) = j + h$. Notons a_{i,k_i}, pour tout $i \leq \text{nc}(\mathcal{L})$, le plus grand sommet selon $\leq_I$ tel que $c(a_{i,k_i}) = i$. Alors, du fait que chaque couche A_i est une clique de G et que G est connexe, la suite de sommets $(x, a_{j,k_j}, a_{j+1,1}, a_{j+1,k_{j+1}}, \ldots, a_{j+h,1}, y)$ forme un chemin de G de longueur au plus $2h + 1$ entre x et y. On en déduit $\text{dist}(x, y) \leq 2h + 1$. Cela prouve l'assertion (1). (En toute rigueur, il faut considérer aussi les cas où $x = a_{j,k_j}$ ou $y = a_{j+h,1}$ ou encore $k_l = 1$ pour un certain l, $j < l < j + h$: dans tous ces cas, le chemin est encore plus court.)

De cette assertion, on tire immédiatement l'assertion (2), du fait que l'on a $\max_{(x,y)\in V^2} d_c(x, y) = k - 1 = \text{nc}(\mathcal{L}) - 1$ et par la définition $\text{diam}(G) = \max_{(x,y)\in V^2} \text{dist}(x, y)$. □

On utilisera aussi le lemme suivant.

Lemme 124 *Soient deux sommets x, y de G vérifiant $c(x) = c(y)$, $x \leq_I y$ et $b(x) \neq \perp$. Alors on a aussi $b(y) \neq \perp$, $c(b(x)) = c(b(y))$ et $b(x) \leq_I b(y)$.*

Preuve. Des hypothèses, on déduit immédiatement $c(b(x)) = c(x) + 1$ et $\{y, b(x)\} \in E$. Enfin, toujours par définition de la fonction "pont" b, on déduit $b(y) \neq \perp$, $c(b(y)) = c(x) + 1 = c(b(x))$ et $b(x) \leq_I b(y)$. □

Définition 135 (arbre des ponts, ordre $\leq_{\mathcal{L}}$) *Soient $G = (V, E)$ un graphe d'intervalles unitaires connexe et $\mathcal{L} = (V, \leq_I, b, c)$ une représentation en couches de G. On définit le graphe orienté $T^{\mathcal{L}}$ où*

- *$V \cup \{\perp\}$ est l'ensemble des noeuds de $T^{\mathcal{L}}$;*
- *(x, y) est un arc de $T^{\mathcal{L}}$ si et seulement si $b(y) = x$.*

Par construction, $T^{\mathcal{L}}$ est un arbre, de fonction parent b et de racine $\perp$. On l'appelle arbre des ponts *de G.*

On désigne par $\leq_{\mathcal{L}}$ l'ordre total sur $V \cup \{\perp\}$ tel que

- *$\leq_{\mathcal{L}}$ est un ordre préfixe de l'arbre $T^{\mathcal{L}}$;*

– *pour deux enfants x et y d'un même noeud z dans $T^{\mathcal{L}}$, on prend $x \leq_{\mathcal{L}} y$ si et seulement si $x \leq_I y$.*

Dans les trois lemmes suivants, on supposera, sans le redire, que $G = (V, E)$ est un graphe d'intervalles unitaires connexe et que $\mathcal{L} = (V, \leq_I, b, c)$ est la représentation en couches de G d'arbre des ponts $T^{\mathcal{L}}$ et d'ordre associé $\leq_{\mathcal{L}}$.

Par construction, on voit facilement que :

Lemme 125 *Deux sommets distincts x et y de G de même couche $c(x) = c(y)$ ne sont jamais ancêtres l'un de l'autre dans l'arbre des ponts $T^{\mathcal{L}}$.*

Preuve. Pour tout sommet x tel que $b(x) \neq \perp$, on a $c(b(x)) = c(x) + 1 > c(x)$, et donc, pour tout ancêtre $y = b^n(x)$ de x dans $T^{\mathcal{L}}$, $n \geq 1$, on a $c(y) > c(x)$. □

Lemme 126 *Soient x et y deux sommets de G tels que $c(x) = c(y)$ et $x \leq_I y$. Alors on a aussi $x \leq_{\mathcal{L}} y$.*

Preuve. On prouve le lemme par induction sur la profondeur de x dans l'arbre $T^{\mathcal{L}}$ (distance de x à la racine).

Profondeur$(x) = 1$: Supposons d'abord $b(x) = \perp$, $c(x) = c(y)$ et $x \leq_I y$. Soit z le sommet ancêtre (au sens large) de y qui est enfant de $\perp$. On a $z \geq_I y \geq_I x$. Puisque x et z sont deux enfants de $\perp$ avec $x \leq_I z$, on a aussi $x \leq_{\mathcal{L}} z \leq_{\mathcal{L}} y$, par définition de l'ordre préfixe $\leq_{\mathcal{L}}$.

Profondeur$(x) > 1$: On suppose maintenant $b(x) \neq \perp$, $c(x) = c(y)$ et $x \leq_I y$. On en déduit $b(y) \neq \perp$, $c(b(x)) = c(b(y))$ et $b(x) \leq_I b(y)$ par le lemme 124. Par l'hypothèse de récurrence appliquée à $b(x)$ et $b(y)$, on obtient $b(x) \leq_{\mathcal{L}} b(y)$. Examinons deux cas.

Cas $b(x) = b(y)$: Puisque x et y sont deux enfants de $b(x)$ avec $x \leq_I y$, on déduit $x \leq_{\mathcal{L}} y$.

Cas $b(x) \neq b(y)$: Puisqu'on a $c(b(x)) = c(b(y))$, les noeuds $b(x)$ et $b(y)$ ne sont ni ancêtres ni descendants l'un de l'autre dans l'arbre $T^{\mathcal{L}}$. Puisqu'on a $b(x) \leq_{\mathcal{L}} b(y)$ pour l'ordre préfixe $\leq_{\mathcal{L}}$, tous les éléments du sous-arbre de racine $b(x)$ sont, pour cet ordre, avant tous ceux du sous-arbre de racine $b(y)$. En particulier, on obtient $x <_{\mathcal{L}} y$. Ce qui achève la preuve par récurrence. □

On note $v_1 <_{\mathcal{L}} v_2 <_{\mathcal{L}} \ldots <_{\mathcal{L}} v_n$ la liste de tous les sommets de G dans l'ordre $\leq_{\mathcal{L}}$. On désigne par $G_i = (V_i, E_i)$ le graphe induit de G où V_i est l'ensemble des i premiers sommets de G selon l'ordre $\leq_{\mathcal{L}}$.

Le lemme suivant caractérise le voisinage dans G_i de chaque sommet extérieur à G_i et donne une construction de G_{i+1} à partir de G_i.

Lemme 127 *Soit y un sommet de $V - V_i$. On a l'égalité*

$$N_G(y) \cap V_i = \{x \in V_i : c(y) \leq c(x) \leq c(y) + 1\}$$

Preuve. Pour tout sommet $x \in N_G(y)$, on a $c(y) - 1 \leq c(x) \leq c(y) + 1$, par la propriété 122. Il suffit donc de prouver les deux points suivantes :

1. $\{x \in V_i : c(y) \leq c(x) \leq c(y) + 1\} \subseteq N_G(y) \cap V_i$;
2. $\{x \in V_i : c(x) = c(y) - 1\} \cap (N_G(y) \cap V_i) = \emptyset$. (Autrement dit, l'ensemble $N_G(y) \cap V_i$, inclus dans l'ensemble $\{x \in V_i : c(y) - 1 \leq c(x) \leq c(y) + 1\}$, contient, parmi les éléments de cet ensemble, tous les x tels que $c(y) \leq c(x) \leq c(y) + 1$ et rien que ceux-ci.)

Point 1 : Evidemment, si on a $c(x) = c(y)$ alors x appartient à $N_G(y)$ puisque chaque couche est une clique de G. Il suffit de prouver l'égalité $\{x \notin N_G(y) : c(x) = c(y)+1\} \cap V_i = \emptyset$ (autrement dit, $\{x \in V_i : c(x) = c(y) + 1\} \subseteq N_G(y)$).

Soit x un sommet de V vérifiant $x \notin N_G(y)$ et $c(x) = c(y) + 1$. Supposons $b(y) = \perp$. Soit x' l'ancêtre (au sens large) de x qui est enfant de $\perp$. Puisqu'on a $y \leq_I x \leq_I x'$ et que y et x' sont des enfants de $\perp$, on a $y <_{\mathcal{L}} x'$ (par définition de l'ordre $\leq_{\mathcal{L}}$). Comme on a aussi $x' \leq_{\mathcal{L}} x$, on en déduit $y <_{\mathcal{L}} x$ et donc, comme y, le sommet x n'appartient pas à V_i.

Supposons maintenant $b(y) \neq \perp$. Alors $c(b(y)) = c(x)$ et $x >_I b(y)$ puisque $\{x, y\} \notin E$. Par conséquent, d'après le lemme 126, on a $b(y) <_{\mathcal{L}} x$ et donc $y <_{\mathcal{L}} x$. On en déduit encore que x n'appartient pas à V_i.

Point 2 : Soit un sommet $x \in N_G(y)$ tel que $c(x) = c(y) - 1$. On a donc $b(x) \neq \perp$, $c(b(x)) = c(y)$ et $y \leq_I b(x)$. Par le lemme 126, on en déduit $y \leq_{\mathcal{L}} b(x)$. Comme on a aussi $b(x) <_{\mathcal{L}} x$, on en déduit encore $y <_{\mathcal{L}} x$ et donc aussi $x \notin V_i$. Ce qui prouve le point 2 et achève la preuve du lemme. □

Remarque 31 *Par leurs définitions, l'arbre des ponts $T^{\mathcal{L}}$ de G et l'ordre total $\leq_{\mathcal{L}}$ sont calculables en temps linéaire à partir de la représentation par intervalles I de G.*

5.3.2 Clique-décomposition des graphes d'intervalles unitaires

Nous avons maintenant tous les outils nécessaires pour décomposer un graphe d'intervalles unitaires.

Lemme 128 *Soit $G = (V, E)$ un graphe d'intervalles unitaires connexe donné par sa représentation I et soit $\mathcal{L} = (V, \leq_I, c, b)$ la représentation en couches de G. Alors on peut calculer en temps linéaire une clique-décomposition de G de largeur $nc(\mathcal{L}) + 1$.*

Preuve. Notons d'abord qu'on peut calculer en temps linéaire la représentation en couches $\mathcal{L}$ de G. Nous allons prouver qu'il existe une clique-décomposition T de largeur $k + 1$ de G avec $k = \text{nc}(\mathcal{L})$ pour l'étiquetage $G' = (V, E, \lambda) \in CW_{k+1}$ du graphe G avec $\lambda : V \to [k + 1]$. L'étiquette $N = k + 1$ est utilisée comme étiquette temporaire (N pour "New"). Pour chaque entier $i \in [k]$, on désigne par $G'_i = (V_i, E_i, \lambda_i)$ l'étiquetage du graphe $G_i = (V_i, E_i)$ (graphe induit

$G[V_i]$ sur les i premiers sommets de G selon l'ordre $\leq_{\mathcal{L}}$) où l'on prend $\lambda_i(x) = c(x)$ pour chaque sommet x. On va prouver, par récurrence sur l'entier i, que le graphe G'_i est dans CW_{k+1}.

Cas $i = 1$: Clairement, on a $G'_1 = \bullet_{c(v_1)}$ où v_1 est l'unique sommet de G_1. La conclusion est immédiate.

Cas $i + 1 (i \geq 1)$: On suppose maintenant $G'_i \in CW_{k+1}$ et on désigne par y l'unique sommet de $V_{i+1} - V_i$: $y = v_{i+1}$. Par le lemme 127, on constate que le graphe G_{i+1} est obtenu en ajoutant à G_i le sommet $y \in V_{i+1} - V_i$ et l'ensemble d'arêtes $\{\{x, y\} \in E : x \in V_i \wedge (c(x) = c(y) \vee c(x) = c(y) + 1)\}$. D'où on déduit le terme T qui exprime G'_{i+1} dans la classe CW_{k+1} :

$$G'_{i+1} = \rho_{N,c(y)} \circ \eta_{N,c(y)+1} \circ \eta_{N,c(y)}(G'_i \uplus \bullet_N)$$

Cela se justifie par les étapes suivantes :

- on ajoute le sommet y avec l'étiquette N ;
- on relie y à tous les sommets $x \in V_i$ des colonnes $c(y)$ et $c(y) + 1$;
- enfin, on réétiquette y par sa colonne $c(y)$.

Ceci conclut la preuve par récurrence que G'_i est dans CW_{k+1}, pour tout $i \in [n]$ où $n = \text{card}(V)$. La clique-décomposition cherchée est obtenue pour $i = n$: $G = G_n$. De plus, on voit facilement que le terme (arbre) T obtenu pour G peut être calculé en temps linéaire à partir de $\mathcal{L}$ et $\leq_{\mathcal{L}}$, et donc à partir de la représentation I de G. □

Proposition 129 *Soit G un graphe d'intervalles unitaires connexe donné par sa représentation I. Il existe un algorithme qui calcule en temps linéaire une clique-décomposition de G de largeur $diam(G) + 2$.*

Preuve. C'est une conséquence immédiate des lemmes 123 et 128. □

Remarque 32 *On peut déduire assez facilement des résultats de Lozin [Loz08] la borne suivante, moins précise que celle de la proposition :*

$$cw(G) \leq 3 \cdot nc(\mathcal{L}) \leq 3 \cdot diam(G) + 3$$

Finalement, nous obtenons le résultat attendu pour la classe UIG.

Corollaire 130 *La classe UIG des graphes d'intervalles unitaires est de largeur de clique localement bornée.*

Preuve. Tout sous-graphe induit d'un graphe d'intervalles unitaires est un graphe d'intervalles unitaires. En conséquence, par la proposition précédente et le lemme 123, on obtient, pour tout sommet x de G,

$$\text{cw}(G[N^k_G(x)]) \leq \text{diam}(G[N^k_G(x)]) + 2 \leq 2k + 2.$$

□

5.4 Model-checking en temps linéaire pour les graphes d'intervalles unitaires

Afin de prouver ce résultat, nous mettons d'abord en place un arsenal d'outils (couvertures, etc) qui s'inspirent directement des concepts introduits par Frick et Grohe [FG01b] en les généralisant.

5.4.1 Classes de graphes linéairement et localement décomposables

Frick et Grohe [FG01b] ont introduit les notions suivantes que nous allons généraliser par la suite.

Définition 136 ((k,l)-couverture) *Soit $G=(V,E)$ un graphe. Une (k,l)-couverture de G est un ensemble $\mathcal{C}$ de parties de V qui vérifient les conditions suivantes :*

- *Pour toute partie $S \in \mathcal{C}$, on a $tw(G[S]) \leq l$;*
- *Pour tout sommet $x \in V$, il existe une partie $S \in \mathcal{C}$ qui contient la boule $N_G^k(x)$.*

Définition 137 (classe linéairement et localement arbre-décomposable) *Une classe de graphes $\mathcal{G}$ est dite linéairement et localement arbre-décomposable s'il existe une fonction f et un algorithme linéaire qui, pour tout entier $k>0$ et tout graphe $G \in \mathcal{G}$, calcule une $(k, f(k))$-couverture $\mathcal{C}_k$ de G.*

Avec ces notions, Frick et Grohe ont établi le résultat suivant.

Théorème 131 *[FG01b] Soit φ un $\{E\}$-énoncé* **FO** *et $\mathcal{G}$ une classe de graphes linéairement et localement arbre-décomposable. Alors le problème $\text{MC}(\varphi, \mathcal{G})$ est décidable en temps linéaire.*

En analysant soigneusement les preuves de Frick et Grohe [FG01b], on s'aperçoit que leurs preuves s'appliquent aussi à des notions plus générales que nous définissons maintenant.

Définition 138 (hiérarchie linéairement FO-décidable) *Soit $\mathcal{S}$ un ensemble de couples $(G,k) \in \text{GRAPH} \times \mathbb{N}$ qu'on appellera une hiérarchie. On dit que la hiérarchie $\mathcal{S}$ est linéairement* **FO***-décidable s'il existe un algorithme qui, pour tout énoncé $\varphi \in$* **FO** *et tout couple $(G,k) \in \mathcal{S}$, décide si on a $G \models \varphi$ en temps $O_{k,\varphi}(|G|)$.*

Remarque 33 *Intuitivement, l'expression $(G,k) \in \mathcal{S}$ signifie que G est au niveau k dans la hiérarchie $\mathcal{S}$ (par exemple, la hiérarchie de la largeur arborescente).*

Exemple 132 *La hiérarchie $\mathcal{S} = \{(G,k) : G \in \text{GRAPH}$ et $k \geq tw(G)\}$ est linéairement* **FO***-décidable par le théorème de Bodlaender [Bod96] qui calcule une décomposition arborescente de*

largeur k de G et le théorème de Courcelle [Cou90b, Cou92] qui, à partir de cette décomposition de G, décide si G satisfait un énoncé φ de **FO**.

A l'inverse, on ne sait pas si la hiérarchie $\mathcal{S} = \{(G,k) : G \in \text{GRAPH et } k \geq cw(G)\}$ est linéairement **FO***-décidable puisqu'on ne connaît pas d'équivalent du théorème de Bodlaender pour la largeur de clique (cependant, comme nous l'avons déjà énoncé, par le théorème 25 de Oum, il est possible de construire, en temps cubique, une clique-décomposition de largeur approchée de G [Oum05, HO07]).*

Ce contre-exemple motive la définition suivante.

Définition 139 (hiérarchie linéairement clique-décomposable) *On dit qu'une hiérarchie $\mathcal{S} \subseteq \text{GRAPH} \times \mathbb{N}$ est linéairement clique-décomposable s'il existe un algorithme qui, pour tout $(G,k) \in \mathcal{S}$, calcule une clique-décomposition de G de largeur au plus k en temps $O_k(|G|)$.*

Cette définition nous conduit au résultat suivant.

Lemme 133 *Toute hiérarchie linéairement clique-décomposable est linéairement* **FO***-décidable.*

Preuve. La proposition se déduit du théorème de Courcelle et al. [CMR00] suivant. Il existe un algorithme qui, étant donnés un graphe G, une clique-décomposition de G de largeur k et un énoncé **FO** (ou **MSO**) φ, décide en temps $O_{k,\varphi}(|G|)$ si on a $G \models \varphi$. □

Nous généralisons à présent les notions de (k,l)-couverture et de classe de graphes linéairement et localement arbre-décomposable (notions de Frick et Grohe rappelées précédemment).

Définition 140 ($(k,l,\mathcal{S})$-couverture) *Soient des entiers k, l et une hiérarchie $\mathcal{S} \subseteq \text{GRAPH} \times \mathbb{N}$. On appelle $(k,l,\mathcal{S})$-couverture d'un graphe $G = (V,E)$ un ensemble $\mathcal{C}$ de parties de V qui vérifient les deux conditions suivantes.*

- *Pour toute partie $S \in \mathcal{C}$, on a $(G[S], l) \in \mathcal{S}$;*
- *Pour tout sommet $x \in V$, il existe un ensemble $S \in \mathcal{C}$ qui contient la boule $N_G^k(x)$.*

Définition 141 (classe linéairement et localement $\mathcal{S}$-décomposable) *Une classe de graphes $\mathcal{G}$ est dite linéairement et localement $\mathcal{S}$-décomposable s'il existe une fonction f et un algorithme qui, pour tout entier $k > 0$ et tout graphe $G \in \mathcal{G}$, calcule en temps $O_k(|G|)$ une $(k, f(k), \mathcal{S})$-couverture $\mathcal{C}_k$ de G telle que $\sum_{S \in \mathcal{C}_k} |G[S]| = O_k(|G|)$.*

Définition 142 (classe linéairement et localement FO-décidable) *Une classe de graphes $\mathcal{G}$ est dite linéairement et localement* **FO***-décidable si elle est linéairement et localement $\mathcal{S}$-décomposable pour une hiérarchie $\mathcal{S}$ linéairement* **FO***-décidable.*

Remarque 34 *Une classe de graphes qui est linéairement et localement arbre-décomposable est linéairement et localement* **FO***-décidable. Plus précisément, la notion de classe (de graphes) linéairement et localement* **FO***-décidable généralise la notion de classe (de graphes) linéairement et localement arbre-décomposable sur le point suivant : on remplace, dans la définition de la* (k,l)*-couverture d'un graphe* G*, la propriété* $tw(G[S]) \leq l$ *par la propriété plus générale suivante : pour tout énoncé* $\varphi \in$ **FO***, on peut décider si* $G[S]$ *satisfait* φ *en temps* $O_{\varphi,l}(|G[S]|)$*.*

De fait, Frick et Grohe [FG01b] ont démontré le résultat suivant, ici réformulé.

Théorème 134 *Soient* φ *un* $\{E\}$*-énoncé* **FO** *et* $\mathcal{G}$ *une classe de graphes linéairement et localement* **FO***-décidable. Alors le problème* $\text{MC}(\varphi, \mathcal{G})$ *est décidable en temps linéaire.*

Nous nous intéressons à un cas particulier de classes linéairement et localement **FO**-décidables : les classes de graphes linéairement et localement clique-décomposables. Il s'agit d'une généralisation naturelle des classes de graphes linéairement et localement arbre-décomposables de Frick et Grohe.

Définition 143 (classe linéairement et localement clique-décomposable) *Une classe de graphes* $\mathcal{G}$ *est dite linéairement et localement clique-décomposable si elle est linéairement et localement* $\mathcal{S}$*-décomposable pour une hiérarchie* $\mathcal{S}$ *linéairement clique-décomposable.*

Lemme 135 *Toute classe de graphes linéairement et localement clique-décomposable est linéairement et localement* **FO***-décidable.*

Preuve. Découle du Lemme 133. □

5.4.2 Evaluation des requêtes FO sur les graphes d'intervalles unitaires

Afin d'établir que le problème d'évaluation des requêtes **FO** sur les graphes d'intervalles unitaires est calculable en temps linéaire, nous allons démontrer que la classe des graphes d'intervalles unitaires (UIG) est linéairement et localement $\mathcal{S}$-décomposable pour un ensemble $\mathcal{S}$ linéairement clique-décomposable. Cet ensemble $\mathcal{S}$ sera la hiérarchie, notée DUIG, que nous introduisons maintenant.

Définition 144 *On note DUIG la hiérarchie suivante, basée sur le diamètre dans la classe UIG :*

$$DUIG = \{(G,k) : G \in UIG \text{ et } k \geq diam(G) + 2\}$$

Lemme 136 *La hiérarchie DUIG est linéairement clique-décomposable.*

Preuve. C'est une conséquence directe du lemme 129. □

Théorème 137 *La classe des graphes d'intervalles unitaires UIG est linéairement et localement clique-décomposable.*

Preuve. Nous allons prouver que la classe UIG est linéairement et localement DUIG-décomposable. Soient un entier $k > 0$ et $G = (V, E)$ un graphe d'intervalles unitaires. Soit I une représentation par intervalles unitaires de G et soit $\mathcal{L} = (V, \leq_I, c, b)$ la représentation en couches associée à I. On désigne par $P_{k,i}$ l'ensemble de sommets $\{x \in V : |c(x) - i| \leq k\}$. Nous allons démontrer que la famille $\mathcal{C}_k = \{P_{k,i} : i \in [\mathrm{nc}(\mathcal{L})]\}$ est une $(k, 4k+3, \mathrm{DUIG})$-couverture de G et que celle-ci est calculable en temps $O_k(|G|)$.

- Comme $P_{k,i}$ est la réunion d'au plus $2k+1$ couches de $\mathcal{L}$, on déduit, par le lemme 123, la borne $\mathrm{diam}(G[P_{k,i}]) \leq 4k+1$. Par conséquent, on obtient $(G[P_{k,i}], 4k+3) \in \mathrm{DUIG}$.
- Soit x un sommet de G. Par le lemme 123, pour chaque sommet y tel que $\mathrm{dist}_G(x, y) \leq k$, on a $|c(x) - c(y)| \leq k$. Par conséquent, on obtient $N^k_G(x) \subseteq P_{k,i}$ pour $i = c(x)$.
- Chaque sommet x appartient à au plus $2k+1$ éléments de la couverture $\mathcal{C}_k$. Par conséquent, $\sum_i |G[P_{k,i}]| \leq (2k+1)|G|$. De plus, la couverture $\mathcal{C}_k$ peut facilement être calculée en temps linéaire.

□

Le corollaire suivant découle des théorèmes 137 et 134.

Corollaire 138 *Soit φ une $\{E\}$-formule* **FO**. *Le problème* MC(φ, *UIG*) *peut être décidé en temps linéaire.*

5.5 Requêtes logiques difficiles sur les graphes d'intervalles

Dans cette section, nous expliquons pourquoi le résultat de complexité de la section précédente ne peut être étendu ni aux graphes d'intervalles, ni aux requêtes **MSO**.

5.5.1 Evaluer une formule du premier ordre est aussi difficile sur les graphes d'intervalles que sur les graphes quelconques

Associer à un graphe quelconque un graphe d'intervalles

Nous allons maintenant associer à chaque graphe $G = (V, E)$ un graphe d'intervalles unitaires $G' = (V', E')$ qui codera le graphe G. Nous supposons d'abord que l'ensemble des sommets de G est numéroté $V = \{1, \ldots, n\}$ et nous représentons le graphe G par le mot $w(G) = \$\#l_1\#l_2\#\ldots l_n\#\$$ où $\#$ et $\$$ sont deux symboles spéciaux, appelés respectivement *délimiteur* et *extrémité*, et l_j est la liste $e_1, e_2 \ldots e_{d_j}$ des arêtes adjacentes au sommet j (d_j est le degré de j).

Exemple 139 *Pour $G = (V, E)$ où $V = \{1, 2, 3, 4\}$, $E = \{a, b, c, d\}$ et $a = 12$, $b = 13$, $c = 23$, $d = 34$, nous obtenons le mot $w(G) = \$\#ab\#ac\#bcd\#d\#\$$.*

Remarque 35 *Un symbole d'arête est introduit pour chaque arête de G : il représente l'arête dans $w(G)$. Dans notre exemple, la liste des symboles d'arête est a, b, c, d. Clairement, la taille du mot $w(G)$ est $|w(G)| = 2card(E) + n + 3 = O(|G|)$ et le temps de calcul de $w(G)$ à partir de G est aussi $O(|G|)$.*

Nous associons maintenant au mot $w(G)$ le graphe d'intervalles $G' = (V', E')$ où $V \subseteq V'$ et qui est donné par la représentation par intervalles I suivante :

1. A chaque symbole d'arête (resp. symbole délimiteur #, symbole d'extrémité \$) à la position p dans $w(G)$, nous associons exactement *deux* (resp. *trois*, *quatre*) sommets $x_i \in V'$, $1 \leq i \leq 2$ (resp. $1 \leq i \leq 3$, $1 \leq i \leq 4$), avec $I(x_i) = [p, p+1]$.
2. A chaque sommet $v \in V$ dont les deux symboles délimiteurs # sont aux positions p et p' dans $w(G)$, $p < p'$, on associe un sommet $x \in V'$ avec $I(x) = [p, p'+1]$
3. A chaque arête $e \in E$ dont les deux occurrences du symbole qui la représente apparaissent dans $w(G)$ aux positions respectives p et p' dans $w(G)$, $p < p'$, on associe un unique sommet $x \in V'$ avec $I(x) = [p, p'+1]$.
4. Il n'y a pas d'autres sommets dans $G' = (V', E')$ que ceux donnés par les clauses 1, 2 et 3.

Faisons d'abord quelques remarques préliminaires :

Remarque 36 *La représentation I du graphe d'intervalles G' est calculable en temps $O(|G|)$. En effet, le temps de calcul de I à partir de $w(G)$ est évidemment $O(|w(G)|) = O(|G|)$.*

Remarque 37 *Pour chaque sommet x de $G' = (V', E')$, les conditions suivantes 1, 2 et 3 sont équivalentes, par construction du graphe G' :*

1. *$I(x)$ est un intervalle unitaire ;*
2. *le sommet x est construit par la clause 1 de la définition ci-dessus de G' ;*
3. *le sommet x admet au moins un sommet jumeau y dans G'.*

Définir un graphe à l'intérieur de son graphe d'intervalles associé

En utilisant la logique du premier ordre, on veut interpréter le graphe $G = (V, E)$ à l'intérieur de son graphe d'intervalles associé $G' = (V', E')$. Dans ce but, nous allons définir une liste de $\{E'\}$-formules du premier ordre dont le sens est suggéré par leur nom, et dont les deux premières sont :

$$\text{Twin}(x, y) \equiv \forall z((z \neq x \wedge z \neq y) \rightarrow (E'(x, z) \leftrightarrow E'(y, z))$$

$$\text{Unit}(x) \equiv \exists y(x \neq y \wedge \text{Twin}(x, y))$$

Par l'équivalence $1 \Leftrightarrow 3$ de la Remarque 37, la formule $\mathrm{Unit}(x)$ exprime le fait que $I(x)$ est un intervalle unitaire.

La formule suivante exprime le fait que la paire d'intervalles $\{I(x), I(y)\}$ est de la forme $\{[p, p+1], [p+1, p+2]\}$:

$$\mathrm{AdjUnit}(x, y) \equiv \mathrm{Unit}(x) \wedge \mathrm{Unit}(y) \wedge E'(x, y) \wedge \neg\mathrm{Twin}(x, y)$$

Nous définissons les formules suivantes pour $k = 1, 2, 3, 4, 5$:

$$\mathrm{Twin}_{\geq k}(x) \equiv \exists x_1 \ldots \exists x_k (x = x_1 \wedge \bigwedge_{1 \leq i < j \leq k} \mathrm{Twin}(x_i, x_j))$$

et pour $k = 1, 2, 3, 4$:

$$\mathrm{Twin}_k(x) \equiv \mathrm{Twin}_{\geq k}(x) \wedge \neg\mathrm{Twin}_{\geq k+1}(x)$$

Par la clause 1 de la définition de G' et la Remarque 37, pour chaque sommet $x \in V'$, nous avons $\mathrm{Twin}_2(x)$ (resp. $\mathrm{Twin}_3(x)$, $\mathrm{Twin}_4(x)$) si et seulement si l'intervalle correspondant $I(x)$ est de la forme $[p, p+1]$ où p est la position du symbole d'arête (resp. symbole délimiteur $\#$, symbole d'extrémité \$) dans le mot $w(G)$. Pour cette raison, on note encore cette formule $\mathrm{Type}_e(x)$ (resp. $\mathrm{Type}_{\#}(x)$, $\mathrm{Type}_{\$}(x)$).

On constate que, pour tout $x \in V'$ qui n'est pas associé à un délimiteur $\#$, la formule suivante du premier ordre, écrite ici sous une forme abrégée suggestive, exprime, pour tout $y \in V'$, que l'on a l'inclusion $I(x) \subseteq I(y)$:

$$\mathrm{Inc}(x, y) \equiv N_{G'}[x] \subseteq N_{G'}[y]$$

La formule suivante exprime que $I(x)$ est une extrémité de $I(y)$, c'est-à-dire que $I(y)$ est de la forme $[p, q+1]$ et $I(x)$ est de la forme $[p, p+1]$ ou $[q, q+1]$.

$$\mathrm{Bound}(x, y) \equiv (\mathrm{Type}_{\#}(x) \vee \mathrm{Type}_e(x)) \wedge \mathrm{Inc}(x, y) \wedge \exists z(\mathrm{AdjUnit}(x, z) \wedge \neg\mathrm{Inc}(x, y))$$

En effet si $I(x)$ est $[p, p+1]$ (resp. $[q, q+1]$), alors pour le sommet z qui vérifie $I(z) = [p-1, p]$ (resp. $I(z) = [q+1, q+2]$), on a $I(z) \not\subseteq I(y)$; à l'inverse, si $I(x)$ est un intervalle unitaire $[r, r+1]$ avec $p < r < q$ alors les intervalles unitaires adjacents à $I(x)$, $[r-1, r]$ et $[r+1, r+2]$ sont tous deux inclus dans $I(y)$.

La formule suivante exprime que $I(a)$ et $I(b)$ sont les deux extrémités (distinctes) de $I(y)$.

$$\mathrm{Bounds}(a, b, y) \equiv \mathrm{Bound}(a, y) \wedge \mathrm{Bound}(b, y) \wedge \neg\mathrm{Twin}(a, b)$$

Pour finir, les $\{E'\}$-formules suivantes permettent de définir l'ensemble des sommets V et la relation E d'adjacence du graphe $G = (V, E)$ à l'intérieur du graphe $G' = (V', E')$.

$$\Delta(x) \equiv \exists a \exists b (\mathrm{Type}_{\#}(a) \wedge \mathrm{Type}_{\#}(b) \wedge \mathrm{Bounds}(a, b, x))$$

$$\Theta_E(x, y) \equiv \exists z \exists a \exists b (\mathrm{Type}_e(a) \wedge \mathrm{Type}_e(b) \wedge \mathrm{Bounds}(a, b, z) \wedge Inc(a, x) \wedge Inc(b, y))$$

C'est ce que dit le lemme suivant, conséquence immédiate de la définition du graphe G' et des remarques ci-dessus.

Lemme 140 *1. Pour chaque sommet $a \in V'$, nous avons $a \in V$ si et seulement si $(G', a) \models \Delta(x)$.*

2. Pour tous sommets $a, b \in V$, nous avons $\{a, b\} \in E$ si et seulement si $(G', a, b) \models \Theta_E(x, y)$.

3. Soit t la transformation $G \mapsto G'$ du graphe G en son graphe d'intervalles associé G'. Le triplet (t, Δ, Θ_E) est une **FO***-interprétation linéaire de la classe des graphes dans la classe des graphes d'intervalles.*

Preuve. Puisqu'on a prouvé les équivalences (1) et (2), il suffit de constater que la transformation $G \mapsto G'$ se fait en temps linéaire : il faut supposer pour cela que le graphe d'intervalles G' est calculé sous forme de sa représentation I. On voit facilement que l'on a $|I| = O(|G|)$ et que le temps de la transformation est $O(|I|) = O(|G|)$. □

On a donc prouvé le résultat suivant.

Théorème 141 *Il existe une* **FO***-interprétation linéaire de la classe des graphes, notée* GRAPH *(resp. de la classe des structures générales de signature fixée) dans la classe IG des graphes d'intervalles.*

Remarque 38 *Il existe une* **FO***-interprétation linéaire de la classe des structures générales de signature fixée dans la classe des graphes non orientés. En effet, il est assez facile de construire une* **FO***-interprétation linéaire de la classe des structures générales (d'une signature fixée) dans la classe des graphes non orientés colorés (sur les sommets) et, enfin, une* **FO***-interprétation linéaire de cette dernière classe dans la classe des graphes. Cette dernière interprétation utilise une configuration spéciale pour représenter chaque couleur.*

Les résultats suivants expriment qu'il est aussi difficile d'évaluer une formule **FO** (et, a fortiori, une formule **MSO**) sur un graphe d'intervalles que sur un graphe ou une structure quelconques.

Corollaire 142 *Les problèmes* MC(**FO**, *IG*) *et* ENUM(**FO**, *IG*) *sont aussi difficiles que les problèmes* MC(**FO**, GRAPH) *et* ENUM(**FO**, GRAPH)*, respectivement. Plus précisément,*

1. Comme p-MC(**FO**, GRAPH)*, le problème p-MC*(**FO**, *IG*) *est AW*[∗]*-complet ;*

2. *Si* ENUM(**FO**, *IG*) *est calculable en temps* $f(|\varphi|)|I|^c$ *(où* I *est la représentation du graphe d'intervalles donné en entrée), pour une certaine fonction* f *et une certaine constante* c*, alors le problème* ENUM(**FO**, GRAPH) *est calculable en temps* $O(f(O(|\varphi|))|G|^c)$*, pour les mêmes* f *et* c*.*

5.5.2 Evaluer une formule MSO est aussi difficile sur les graphes d'intervalles unitaires que sur les structures générales

Pour établir cela, nous allons procéder en deux étapes :

- Nous donnons d'abord une **MSO**-interprétation polynomiale de la classe des structures générales dans celle des grilles colorées ;
- enfin, nous exhibons une **FO**-interprétation linéaire de la classe des grilles colorées dans la classe des graphes d'intervalles unitaires donnés par leurs représentations par intervalles propres I.

Description d'une structure générale dans une grille colorée

Définition 145 (grille colorée) *On appelle grille colorée* $m \times n$ *un graphe coloré* $G = (V, E, \lambda)$ *avec* $V = [m] \times [n]$, $E = \{\{(i,j),(i',j')\} \in \mathcal{P}_2(V) : |i - i'| + |j - j'| = 1\}$ *et une fonction (de coloration)* $\lambda : V \to [k]$.

Lemme 143 *Il existe une* **MSO***-interprétation polynomiale de la classe des structures générales (de signature fixé) dans la classe des grilles colorées.*

Preuve. La remarque 38 permet de se ramener à la classe des graphes (non orientés). Soit un graphe quelconque $G = (V, E)$ avec $V = [n]$. On associe à G la grille colorée $G' = (V', E', \lambda)$ avec $V' = [n]^2$ et la coloration $\lambda : V' \to \{0, 1, 2\}$ définie comme suit :

- à tout sommet $i \in V$, on associe le sommet (i, i) de couleur 1 dans la grille G' : V est ainsi représenté par l'ensemble des sommets "diagonaux" de la grille ;
- à toute arête $\{i, j\} \in E$, on associe les deux sommets (i, j) et (j, i) de couleur 2 dans la grille G' ;
- tous les autres sommets de G' sont de couleur 0.

La transformation $G \mapsto G'$ est évidemment calculable en temps polynomial. Il reste à définir le graphe d'origine G dans la grille colorée G' à l'aide de formules **MSO** $\Delta(x)$ et $\Theta_E(x, y)$ à construire. Pour cela, on introduit plusieurs formules intermédiaires. Celles-ci sont données ci-dessous avec des abréviations facilement exprimables en logique du premier ordre.

La formule $\mathrm{path}(A, x, y)$ que nous allons donner exprime que l'ensemble de sommets A (A est une variable du second ordre monadique) est un chemin minimal (par inclusion des sommets) de G' entre les sommets x et y. Elle utilise la formule intermédaire suivante :

$$\text{path'}(A,x,y) \equiv \text{card}(N_{G'}(y) \cap A) = 1 \wedge A(x) \wedge \forall z((A(z) \wedge z \neq x \wedge z \neq y) \rightarrow \text{card}(N_{G'}(z) \cap A) = 2)$$

La formule path'(A,x,y) exprime que A est une réunion disjointe d'un chemin minimal de G' entre x et y et de $i \geq 0$ cycles minimaux (sans corde) de G'.

La formule suivante exprime que A est minimal pour l'inclusion par rapport à la propriété path'(A,x,y) et donc qu'on a $i = 0$: A est un chemin minimal de G' entre x et y.

$$\text{path}(A,x,y) \equiv \text{path'}(A,x,y) \wedge \forall A'(A' \subsetneq A \rightarrow \neg\text{path'}(A',x,y))$$

La formule square(x,y,z,t) exprime que (x,y,z,t) est un cycle de longueur 4 de G'.

$$\text{square}(x,y,z,t) \equiv E'(x,y) \wedge E'(y,z) \wedge E'(z,t) \wedge E'(t,x) \wedge x \neq z \wedge y \neq t$$

La formule line(A,x,y) exprime que A est un chemin en ligne droite (horizontale ou verticale) de la grille G' entre x et y.

$$\text{line}(A,x,y) \equiv \text{path}(A,x,y) \wedge \neg\exists u \exists v \exists w \exists z(A(u) \wedge A(v) \wedge A(w) \wedge \text{square}(u,v,w,z))$$

On définit le domaine initial V de G par la formule $\Delta(x) \equiv \lambda(x) = 1$.
On peut aisément voir que, pour tous sommets $a, b \in V$, on a $\{a,b\} \in E$ si et seulement si $(G', a, b) \models \Theta_E(x,y)$ pour la formule

$$\Theta_E(x,y) \equiv x \neq y \wedge \exists A \exists B \exists z(\lambda(z) = 2 \wedge \text{line}(A,x,z) \wedge \text{line}(B,y,z))$$

Ce qui achève la preuve du lemme.

□

Description d'une grille colorée dans un graphe d'intervalles unitaires

Cette **FO**-interprétation est un peu plus compliquée que la **MSO**-interprétation précédente. Elle utilise encore la notion de *position* dans le graphe G' construit et de *type* à chaque position. C'est le nombre de sommets de G' à la même position qui identifie le type (et la couleur éventuelle) de cette position.

Soit $G = (V, E, \lambda)$ une grille colorée $m \times n$ avec $V = [m] \times [n]$ et $\lambda : V \rightarrow [k]$.

On construit le graphe $G' = (V', E')$ de la manière suivante. D'abord, on associe à G un ensemble de "positions" $[3m+2] \times [n+2]$ qui forment une nouvelle grille. L'ensemble des sommets V' et l'ensemble des arêtes E' de G' sont définis comme suit.

Sommets de G : Pour chaque sommet $x = (i,j) \in [m] \times [n]$ de couleur $\lambda(x) = l$, on associe dans G'

- un sommet de "position" $(3i, j+1)$ et de type "Vertex" destiné à représenter x ;
- $l+1$ sommets de "position" $(3i-1, j+1)$ et de type "Color" destinés à représenter la couleur de x ;
- $l+1$ sommets de "position" $(3i+1, j+1)$ et de type "Color", sommets symétriques des précédents.

A chacune des positions suivantes (bords de la grille), on ajoute à G' $k+2$ sommets de type "Bord".

- aux positions $(i, 1)$ et $(i, n+2)$, pour $i \in [3m+2]$, positions de type "Bord horizontal" ;
- aux positions $(1, j)$ et $(3m+2, j)$, pour $j \in [n+2]$, positions de type "Bord vertical".

Arêtes de G' : On place une arête entre deux sommets distincts x et y de V' de "positions" respectives (i, j) et (i', j') si et seulement si on a $i = i'$, ou bien $i' = i+1$ et $j' \leq j$, ou bien $i'+1 = i$ et $j \leq j'$.

De la construction de G', on déduit immédiatement le lemme suivant :

Lemme 144 *Deux sommets de G' sont jumeaux si et seulement si ils ont la même "position".*

On construit un ordre linéaire de V', noté $\leq_I$, strictement croissant selon l'ordre lexicographique des positions. Autrement dit, pour tous sommets x et x' de positions respectives (i, j) et (i', j') avec $(i, j) <_{lex} (i', j')$, c'est-à-dire vérifiant $i < i'$, ou bien $i = i'$ et $j < j'$, on a $x <_I x'$.

On constate d'abord la propriété suivante.

Lemme 145 *Le voisinage fermé $N_{G'}[x]$ de tout sommet x de G' forme un intervalle, noté $[b(x), e(x)]$, de l'ordre $\leq_I$.*

Preuve. Soient x et x' deux sommets de G' de positions respectives (i, j) et (i', j'). Par construction de G', on a l'équivalence $x' \in N_{G'}[x] \Leftrightarrow (i-1, j) \leq_{lex} (i', j') \leq_{lex} (i+1, j)$. Cela implique, par construction de l'ordre $\leq_I$ (croissant dans l'ordre lexicographique des positions), que le voisinage fermé $N_{G'}[x]$ est un intervalle de l'ordre $\leq_I$ vérifiant l'égalité $N_{G'}[x] = \{x' \in V' : b(x) \leq_I x' \leq_I e(x)\}$ où $b(x)$ (resp. $e(x)$) est le sommet de position $\geq_{lex} (i-1, j)$ (resp. de position $\leq_{lex} (i+1, j)$) qui est minimal (resp. maximal) selon $\leq_I$ pour cette condition. □

On rappelle que, pour tout sommet $x \in V'$, on note $e(x)$ le plus grand sommet $y \in V'$ dans l'ordre $\leq_I$ tel que $\{x, y\} \in E'$.

Lemme 146 *La fonction $x \mapsto e(x)$ est croissante (au sens large) pour l'ordre $\leq_I$.*

Preuve. Soit $M = 3m+2$ (resp. $N = n+2$) le maximum des i (resp. j) pour toutes les positions (i, j) des sommets de G'. Pour un sommet $x \in V'$, on a deux cas : soit position$(x) = (i, j)$ avec $i < M$ et alors on a position$(e(x)) = (i+1, j)$; soit position$(x) = (M, j)$ et alors on a position$(e(x)) = (M, N)$. On en déduit que, pour tous sommets x et y de G', si on a position$(x) \leq_{lex}$ position(y)

alors on a aussi position$(e(x)) \leq_{lex}$ position$(e(y))$. Le lemme découle de la suite d'implications $x \leq_I y \Rightarrow$ position$(x) \leq_{lex}$ position$(y) \Rightarrow$ position$(e(x)) \leq_{lex}$ position$(e(y)) \Rightarrow e(x) \leq_I e(y)$. □

Lemme 147 *1. G' est un graphe d'intervalles unitaires, de représentation par intervalles propre $I : V' \to S$, $I(x) = [x, e(x)]$.*

2. Il existe un algorithme linéaire qui, pour toute grille colorée $G = (V, E, \lambda)$, calcule une représentation par intervalles propre I de G'.

Preuve. A tout sommet x de V', on associe l'intervalle $I(x) = [x, e(x)]$ sur la droite discrète associée à l'ordre $\leq_I$. On a les propriétés suivantes :

- Pour toute paire $x, y \in V'$ de sommets distincts, on a $\{x, y\} \in E'$ si et seulement si $I(x) \cap I(y) \neq \emptyset$: En effet, supposons $x <_I y$ (le cas $y <_I x$ est symétrique). On a les équivalences

$$\{x, y\} \in E' \Leftrightarrow (y \in [b(x), e(x)] \wedge y \geq_I x) \Leftrightarrow x \leq_I y \leq_I e(x) \Leftrightarrow [x, e(x)] \cap [y, e(y)] \neq \emptyset$$

- L'ensemble d'intervalles $\{I(x) : x \in V'\}$ est propre, c'est-à-dire il n'existe pas deux intervalles $I(x)$ et $I(y)$ tels que $x <_I y \leq_I e(y) <_I e(x)$ puisque la fonction $x \mapsto e(x)$ est croissante (au sens large) pour l'ordre $\leq_I$.

Par conséquent, on a établi que I, qui est une représentation par intervalles du graphe G', est une représentation propre : G' est donc un graphe d'intervalles unitaires, d'après la remarque 29. □

Description de la grille colorée G dans le graphe d'intervalles unitaires G'

Dans la section précédente, nous avons défini la formule suivante qui exprime que deux sommets sont jumeaux dans G', c'est-à-dire sont à la même position, d'après le lemme 144.

$$\text{Twin}(x, y) \equiv x \neq y \wedge \forall z((z \neq x \wedge z \neq y) \to (E'(x, z) \leftrightarrow E'(y, z))$$

Nous rappelons les formules suivantes pour $l = 2, \ldots, k + 3$:

$$\text{Twin}_{\geq l}(x) \equiv \exists x_1 \ldots \exists x_l (x = x_1 \wedge \bigwedge_{1 \leq i < j \leq l} \text{Twin}(x_i, x_j))$$

Pour $l = 1, \ldots, k + 2$, on a les formules :

$$\text{Twin}_l(x) \equiv \text{Twin}_{\geq l}(x) \wedge \neg \text{Twin}_{\geq l+1}(x)$$

Il est clair que l'on peut définir dans G' le domaine V de G. En effet, pour tout sommet $a \in V'$, on a, par construction de G', $a \in V$ si et seulement si $(G', a) \models \Delta(x)$ avec $\Delta(x) \equiv \text{Twin}_1(x)$.

Pour exprimer qu'un sommet de G' est de type "Bord", on définit la formule suivante :

$$\text{Type}_B(x) \equiv \text{Twin}_{k+2}(x)$$

Définition 146 (sommets adjacents horizontalement, adjacents verticalement) *Soient x et y deux sommets de G'. On dit que x et y sont adjacents horizontalement (resp. verticalement) si x est à la position (i,j) et y est à la position (i',j') avec $|i-i'|=1$ et $j=j'$ (resp. $i=i'$ et $|j-j'|=1$).*

Le lemme suivant donne une caractérisation de l'adjacence verticale.

Lemme 148 *Soient x et y deux sommets de G' vérifiant (1) et (2) :*

1. *l'un des deux sommets n'est pas de type "Bord horizontal" ;*
2. *aucun des deux sommets x et y n'est de type "Bord vertical".*

Alors, les sommets x et y sont adjacents verticalement si et seulement si x et y n'ont pas la même position et l'ensemble $N_{G'}[x] - N_{G'}[y]$ est constitué de sommets qui sont tous à la même position.

Preuve. $\Rightarrow$: Soient deux sommet x, y de G' vérifiant (1) et (2). Supposons position$(x) = (i,j)$ et position$(y) = (i, j+1)$ (resp. $(i, j-1)$) avec $2 \leq i \leq M-1$. Alors tous les sommets de $N_{G'}[x] - N_{G'}[y]$ sont à la position $(i-1, j)$ (resp. $(i+1, j)$).

$\Leftarrow$: On va prouver la contraposée de cette implication. Supposons que les sommets x et y ne sont pas adjacents verticalement et ont des positions différentes. On doit montrer que les sommets de $N_{G'}[x] - N_{G'}[y]$ ne sont pas tous à la même position. On a deux cas :
Premier cas : position$(y) <_{lex}$ position(x) : Comme x et y ne sont pas adjacents verticalement, leurs positions respectives ne sont pas consécutives dans l'ordre lexicographique. Notons position$(x) = (i_x, j_x)$ et position$(y) = (i_y, j_y)$. Soit z un sommet avec position$(z) = (i_z, j_z)$, $(i_y, j_y) <_{lex} (i_z, j_z) <_{lex} (i_x, j_x)$ et (i_z, j_z) étant le prédécesseur immédiat de (i_x, j_x) dans l'ordre $<_{lex}$. On a $i_z \leq i_x < N$. On considère deux sommets x' de position $(i_x + 1, j_x)$ et z' de position $(i_z + 1, j_z)$, donc de positions différentes. On constate l'inclusion $\{x', z'\} \subseteq N_{G'}[x] - N_{G'}[y]$.
Second cas : position$(x) <_{lex}$ position(y) : Le raisonnement est symétrique (inverser gauche avec droite, haut avec bas, et inverser l'ordre $\leq_{lex}$).

On a donc prouvé la contraposée, ce qui achève la preuve du lemme. □

Justifiée par le lemme précédent, la formule suivante exprime le fait que deux sommets x, y qui vérifient les conditions (1) et (2) du lemme 148 sont adjacents verticalement :

$$E_v(x,y) \equiv \neg\text{Twin}(x,y) \wedge \forall u \forall v(\{u,v\} \subseteq N_{G'}[x] - N_{G'}[y] \Rightarrow \text{Twin}(u,v))$$

L'adjacence horizontale est caractérisée par le lemme suivant.

Lemme 149 *Soient deux sommets x et y de G' qui ne sont pas de type "Bord". Alors on a l'équivalence entre (1) et (2).*

1. *x et y sont adjacents horizontalement ;*
2. *$\{x, y\} \in E'$ et il existe un sommet $z \notin N_{G'}[y]$ adjacent verticalement à x.*

Preuve. (1) $\Rightarrow$ (2) : Supposons que x et y sont adjacents horizontalement dans G' avec position$(x) = (i, j)$ et position$(y) = (i + 1, j)$ (resp. $(i - 1, j)$). On a évidemment $\{x, y\} \in E'$. Soit z un sommet de position $(i, j - 1)$ (resp. $(i, j + 1)$). Par construction, on a $z \notin N_{G'}[y]$ et z est adjacent verticalement à x.

(2) $\Rightarrow$ (1) : Supposons qu'on a l'arête $\{x, y\} \in E'$ et un sommet $z \notin N_{G'}[y]$ adjacent verticalement à x. On note position$(x) = (i, j)$. On a donc position$(z) = (i, j + 1)$ (resp. $(i, j - 1)$). Les sommets de $N_{G'}[x] - N_{G'}[z]$ dont y fait partie, par hypothèse, sont les sommets de position $(i - 1, j)$ (resp. $(i + 1, j)$). Donc x et y sont adjacents horizontalement. □

Justifiée par le lemme précédent, la formule suivante exprime le fait que deux sommets x, y qui ne sont pas de type "Bord" sont adjacents horizontalement :

$$E_h(x, y) \equiv E'(x, y) \wedge \exists z(E_v(x, z) \wedge z \notin N_{G'}[y])$$

On voit facilement (par construction de G') que, pour tout sommet $a \in V$, on a $\lambda(a) = i$ si et seulement si $(G', a) \models \Theta_{\lambda_i}(x)$, pour la formule

$$\Theta_{\lambda_i}(x) \equiv \exists y(E_h(x, y) \wedge \text{Twins}_{i+1}(y))$$

et que pour tous sommets $a, b \in V$, on a $\{x, y\} \in E$ si et seulement si $(G', a, b) \models \Theta_E(x, y)$ pour la formule

$$\Theta_E(x, y) \equiv E_v(x, y) \vee \exists u \exists v(E_h(x, u) \wedge E_h(u, v) \wedge E_h(v, y))$$

On ainsi établi que les formules Δ, Θ_E et Θ_{λ_i} pour $i \in [k]$, satisfont les conditions d'une interprétation de la classe des grilles colorées dans la classe des graphes d'intervalles unitaires. A cause du lemme 147, on a donc démontré le résultat suivant :

Lemme 150 *Il existe une* **FO***-interprétation linéaire de la classe des grilles colorées dans la classe des graphes d'intervalles unitaires, donnés par leurs représentations propres.*

Des Lemmes 143 et 150, on tire par transitivité le résultat suivant.

Théorème 151 *Il existe une* **MSO***-interprétation polynomiale de la classe des structures générales dans la classe des graphes d'intervalles unitaires.*

De ce théorème, on déduit le même résultat de complexité pour la classe UIG que pour la classe des structures générales. C'est le corollaire suivant.

Corollaire 152 *Pour chaque niveau $\Sigma_k P$ ($k \geq 1$) de la hiérarchie polynomiale, on peut trouver un $\{E\}$-énoncé* **MSO** φ *tel que* $\text{MC}(\varphi, \mathit{UIG})$ *est $\Sigma_k P$-complet.*

5.6 Conclusion

En donnant des restrictions sur la notion de classe de graphes linéairement et localement arbre-décomposable, Frick [Fri04] a prouvé qu'on peut évaluer une requête $\varphi(\mathcal{M})$ de **FO** sur une structure $\mathcal{M}$ d'une telle classe en temps $O(|\mathcal{M}| + |\varphi(\mathcal{M})|)$. Au delà de ce résultat, une question naturelle se pose : peut-on énumérer les solutions de $\varphi(\mathcal{M})$ à délai constant ?

Par ailleurs, on note que la définition de Frick s'adapte aux classes de graphes linéairement et localement clique-décomposables et s'applique alors à la classe UIG. On peut encore se poser la question : peut-on énumérer à délai constant les solutions de $\varphi(\mathcal{M})$ pour $\mathcal{M}$ appartenant à une telle classe (ou seulement $\mathcal{M} \in \text{UIG}$) et une formule φ de **FO** ?

Chapitre 6

Conclusion

Nous récapitulons ici d'une part ce que nous estimons être les apports principaux de cette thèse, d'autre part quelques questions ouvertes suscitées par ce travail et qui s'ajoutent à celles déjà présentées dans les chapitres précédents.

Nos contributions se situent principalement sur deux plans :

Les méthodes ques nous avons introduites ou systématisées

- des méthodes logiques : traductions et interprétations, réductions exactes, élimination des quantificateurs et normalisation sur les structures finies, aux chapitres 2 et 4 ;
- des concepts et des méthodes combinatoires : méthode de couvertures, représentants et arbres de choix, algorithmes sur les hypergraphes au chapitre 2 ; représentation en couches, arbre et ordre associés, pour les graphes d'intervalles unitaires, au chapitre 5.

Les résultats de complexité que nous avons obtenus pour les problèmes de requêtes

- des bornes optimales dans plusieurs cas : requêtes **FO** sur les structures de degré borné, requêtes **MSO** sur les structures de largeur arborescente bornée, requêtes $\mathbf{CCQ}^{\neq}$;
- des résultats de classification : classification dichotomique des requêtes acycliques avec ou sans inégalités, requêtes logiques sur les graphes d'intervalles, etc.

6.1 Nos principales contributions

Un cadre de complexité pour l'énumération

En premier lieu, nos résultats s'inscrivent dans un cadre de classes de complexité que nous avons introduites :

- les classes $\mathrm{enum}(f, g)$ pour les problèmes d'énumération ;
- les classes $\mathrm{dyn}(f, g)$ pour les problèmes "dynamiques", dans un sens que nous avons défini.

Nous avons, pour ces classes, défini une notion de *réduction adaptée*, la réduction exacte. Nous avons démontré, pour cette réduction et ces classes, des propriétés de clôture. Ces classes sont robustes à la fois parce que leur définition est assez indépendante de la définition du modèle (pourvu qu'il utilise un accès par adresse) et à cause de leurs propriétés de clôture.

Généralité/optimalité

Notre problématique de base est la question de l'énumération de requêtes logiques avec diverses restrictions sur la logique ou les structures considérées. L'un des intérêts de ce point de vue est de permettre d'obtenir facilement, pour divers problèmes combinatoires difficiles (**NP**-complets), des bornes supérieures de complexité pour l'énumération de leurs solutions à partir d'un résultat général. C'est ce que nous avons fait tout particulièrement au chapitre 2, en examinant de la façon la plus fine possible la complexité de l'évaluation d'une requête $\varphi(\mathcal{M})$, non seulement en fonction de la taille de la structure $\mathcal{M}$, mais aussi en fonction des paramêtres, nombre d'inégalités et nombre de variables, principalement, liés à la formule φ ; c'est aussi pour cela que nous avons introduit les formules $\mathbf{F\text{-}ACQ}^{all\neq}$. Dans les deux cas (problème d'énumération et problème dynamique) et de façon très similaire, nous avons mis à jour la nécessité et l'importance de la notion de prétraitement (ou précalcul ou phase statique) mesuré par la fonction f. La composition de ce prétraitement et du calcul proprement dit, mesuré par la fonction g, constitue le schéma d'énumération (resp. schéma dynamique), notion que nous avons introduite et dont nous avons montré l'intérêt et la robustesse. La contrepartie de la généralité de notre méthode est qu'elle ne donne pas nécessairement la borne que peut donner une méthode ad hoc pour un problème combinatoire donné.

Méthodes logiques et combinatoires

Une méthode de base que nous avons utilisée dans l'évaluation de requêtes logiques est une forme d'*élimination des quantificateurs* (chapitres 2 et 4) : typiquement, nous ramenons, en *temps linéaire*, l'évaluation d'une requête $\varphi(\mathcal{M})$ à celle d'une requête $\varphi'(\mathcal{M}') = \varphi(\mathcal{M})$ où φ' est une formule ayant une variable de moins que φ et qu'on applique sur une nouvelle structure $\mathcal{M}'$. Nous avons construit, tout particulièrement au chapitre 2, des outils combinatoires (couvertures, représentants, arbres de choix, etc), qui sont utiles à la fois pour cette élimination des quantificateurs et pour l'énumération des solutions des requêtes. On peut penser que l'on pourrait se servir de ces outils dans d'autres applications.

Classification des requêtes conjonctives

L'une des principales contributions de la thèse porte sur la classification de la complexité des formules conjonctives (chapitre 2). Ceci est à comparer avec la caractérisation suivante obtenue

par Grohe, Schwentick et Segoufin [GSS01] (sous l'hypothèse $\mathbf{P} \neq \mathbf{NP}$) : Etant donnée une classe de graphes $\mathcal{G}$, le problème MC($\mathbf{CQ}_{\mathcal{G}}$) est décidable en temps polynomial si et seulement si $\mathcal{G}$ est une classe de graphes de largeur arborescente bornée. Ici, $\mathbf{CQ}_{\mathcal{G}}$ représente la classe des formules conjonctives dont le graphe de Gaifman est dans $\mathcal{G}$. De même, si on note $\mathbf{CQ}_{H,S}$ (resp. ${\mathbf{CQ}^{\neq}}_{H,S}$) la classe des requêtes conjonctives $\mathbf{CQ}$ (resp. $\mathbf{CQ}^{\neq}$) dont l'hypergraphe est $H = (V, E)$ et dont l'ensemble des variables libres est $S \subseteq V$, nous avons démontré (sous l'hypothèse que le produit de matrice n'est pas calculable en temps linéaire) l'équivalence suivante : pour tout hypergraphe acyclique $H = (V, E)$ et tout ensemble de sommets $S \subseteq V$, le problème ENUM($\mathbf{CQ}_{H,S}$) (resp. ENUM(${\mathbf{CQ}^{\neq}}_{H,S}$)) est dans CONSTANT-DELAY$_{lin}$ si et seulement si l'hypergraphe H est S-connexe par $\subseteq$-extension (voir la définition de cette notion au chapitre 2).

Constantes multiplicatives du temps linéaire

Notre méthode de calcul d'une requête **MSO** φ sur une structure $\mathcal{M}$ de largeur arborescente bornée (chapitre 3) nous a permis de mettre en évidence un contraste étonnant entre les deux fais suivants : alors qu'il a été prouvé par Frick et Grohe [FG04b] (sous l'hypothèse $\mathbf{P} \neq \mathbf{NP}$) que, pour toute constante d et toute fonction élémentaire f, il n'existe aucun algorithme qui décide, pour toute formule φ de **MSO**, la question $\varphi(\mathcal{M}) \neq \emptyset$ en temps $f(|\varphi|)|\mathcal{M}|^d$, l'algorithme d'énumération que nous avons construit pour le calcul de $\varphi(\mathcal{M})$ exécute d'abord un précalcul de temps $f(|\varphi|)|\mathcal{M}|$, pour une fonction f non élémentaire, puis énumère les solutions $S \in \varphi(\mathcal{M})$, à délai borné par $c|S|$, pour une constante c *indépendante de la formule* φ. Puisqu'elle n'intègre aucun paramètre caché, cette constante multiplicative du délai est petite intrinsèquement et fait fortement contraste avec la constante $f(|\varphi|)$, intrinsèquement énorme.

Complexité optimale

La principale originalité du chapitre 4 est d'établir le premier résultat optimal pour le calcul de la j-ème solution sur une classe générale de requêtes logiques (les requêtes **FO** sur les structures de degré borné) : un temps de calcul constant après un précalcul linéaire. Nous estimons que c'est un cas plutôt rare. En particulier, on ne peut pas calculer le nombre de k-chemins (chemins simples de longueur k) d'un graphe G en temps $O_k(|G|^c)$ pour une constante c (indépendante de k), sous l'hypothèse communément admise $\mathbf{FPT} \neq \#W[1]$ [FG04a]. Comme le problème du k-chemin s'exprime par une formule $\mathbf{CCQ}^{\neq}$, le calcul de la j-ème solution d'une requête $\mathbf{CCQ}^{\neq}$ n'est pas calculable après prétraitement en temps linéaire, ni même $O(|\mathcal{M}|^c)$, pour une constante c indépendante de la formule φ, ceci sous la même hypothèse de complexité. Cela contraste avec le fait qu'on sait énumérer à délai constant, après un précalcul linéaire, l'ensemble des solutions $\varphi(\mathcal{M})$ d'une telle requête.

Complexité quasi-optimale et recherche dichotomique

De façon intermédiaire, on a prouvé que l'on pouvait calculer la j-ème solution S d'une requête **MSO** $\varphi(T)$ sur un arbre T en temps $O_\varphi(|S| \log |T|)$n après prétraitement linéaire (chapitre 3). De même, il est facile de constater que l'on obtient un résultat comparable pour une requête **CCQ** (sans inégalité) : un temps $O_\varphi(\log |\mathcal{M}|)$ après prétraitement $O_\varphi(|\mathcal{M}|)$. Dans les deux cas, le terme logarithmique est dû à une recherche dichotomique qui parait difficilement évitable. Peut-on justifier ce facteur par un argument de la forme suivante : si on savait calculer la j-ème solution (d'une requête **CCQ**) en temps constant alors on pourrait tester en temps constant la présence d'un élément donné dans un tableau trié.

Enfin, au chapitre 5, nous avons exhibé une nouvelle classe de graphes, la classe des graphes d'intervalles unitaires, notée UIG, pour laquelle chaque énoncé **FO** est décidable en temps linéaire. Nous avons aussi montré que ce résultat ne peut s'étendre ni à **MSO** ni sur la classe des graphes d'intervalles quelconques. L'intérêt de ce chapitre est aussi qu'on y établit que la classe des graphes d'intervalles unitaires a une largeur de clique localement bornée. C'est à partir de cette propriété et en utilisant une adaptation d'une méthode de Frick et Grohe [FG01b] que nous avons prouvé notre résultat de complexité linéaire sur UIG. Rappelons qu'à notre connaissance UIG est la seule classe naturelle de graphes à être prouvée de largeur de clique localement bornée sans être ni de largeur de clique bornée ni de largeur arborescente localement bornée, là aussi de façon prouvée.

6.2 Récapitulatif des complexités des problèmes de requêtes

Nous récapitulons ici dans un tableau les complexités des problèmes de requêtes étudiés dans cette thèse, dans les quatre versions suivantes : model-checking, énumération, comptage et j-ème solution. Intuitivement, et le tableau le confirme, ces quatre problèmes sont de difficulté croissante. Seul cas qui peut être discuté, nous pensons qu'en général, le problème de comptage est plus difficile que celui de l'énumération. Pour chaque résultat de la littérature ou qui en est une conséquence directe, le tableau donne la référence de l'article où il est prouvé. Les résultats sans référence sont démontrés dans cette thèse ou sont des conséquences immédiates de nos résultats. Nous omettons, dans ce tableau, les requêtes **FO** étudiées au chapitre 5, sur les graphes d'intervalles unitaires principalement, puisqu'ils n'étudient que le problème de leur évaluation. La complexité des deux problèmes marqués avec " ?" (comptage et j-ème solution pour les requêtes **ACQ**) nous semble une question ouverte ; nous conjecturons qu'ils sont aussi difficiles que pour $\mathbf{ACQ}^{\neq}$.

Le tableau ne mentionne non plus deux autres ensembles de résultats de la thèse :

– des résultats sur la complexité combinée des requêtes présentés aux chapitres 2 et 3 ;

– des résultats sur la complexité des requêtes **FO** ou **MSO** sur les graphes de largeur de clique bornée ou de largeur de clique localement bornée.

	$\text{MC}(\varphi)$	$\text{Enum}(\varphi)$	$\text{Count}(\varphi)$	$\text{Jth}(\varphi)$
FO ($\deg(\mathcal{M}) \leq k$)	linéaire [See96]	$\text{Constant-Delay}_{lin}$ [DG07]	linéaire	$\text{Constant-Time}_{lin}$
MSO ($\text{tw}(\mathcal{M}) \leq k$)	linéaire [Cou87]	Lin-Delay_{lin}	linéaire [ALS91]	$\text{dyn}(\lvert\mathcal{M}\rvert, \lvert S\rvert \log \lvert\mathcal{M}\rvert)$
CCQ	linéaire [Yan81]	$\text{Constant-Delay}_{lin}$	linéaire	$\text{dyn}(\lvert\mathcal{M}\rvert, \log \lvert\mathcal{M}\rvert)$
$\mathbf{CCQ}^{\neq}$	linéaire	$\text{Constant-Delay}_{lin}$	#W[1]-complet [FG04a]	#W[1]-dur [FG04a]
ACQ	linéaire [Yan81]	délai $O(\lvert\mathcal{M}\rvert)$	?	?
$\mathbf{ACQ}^{\neq}$	linéaire	délai $O(\lvert\mathcal{M}\rvert)$	#W[1]-complet [FG04a]	#W[1]-dur [FG04a]
$\mathbf{CQ}(\text{ctw}(\varphi) \leq k)$	$O(\lvert\text{Dom}(\mathcal{M})\rvert^{k+1})$ [CR00]	$\text{enum}(\lvert\text{Dom}(\mathcal{M})\rvert^{k+1}, 1)$	$O(\lvert\text{Dom}(\mathcal{M})\rvert^{k+1})$	$\text{dyn}(\lvert\text{Dom}(\mathcal{M})\rvert^{k+1}, \log \lvert\mathcal{M}\rvert)$
$\mathbf{CQ}^{\neq}(\text{ctw}(\varphi) \leq k)$	$O(\lvert\text{Dom}(\mathcal{M})\rvert^{k+1})$	$\text{enum}(\lvert\text{Dom}(\mathcal{M})\rvert^{k+1}, 1)$	#W[1]-complet [FG04a]	#W[1]-dur [FG04a]
$\mathbf{CQ}$, $\mathbf{CQ}^{\neq}$	W[1]-complet [PY99]	W[1]-dur [PY99]	#W[1]-complet [FG04a]	#W[1]-dur [FG04a]

6.3 Quelques questions ouvertes ou prolongements

Dans les divers chapitres du manuscrit, nous avons présenté plusieurs questions ouvertes et conjectures spécifiques à ces chapitres. Nous concluons maintenant en énonçant d'autres questions ouvertes ou en suggérant des pistes de recherche de portée plus générale.

Classes de complexité pour l'énumération

Il serait intéressant de trouver des problèmes complets pour les classes de complexité d'énumération, les classes enum(f, g) et leurs variantes. Notons que, même si la réduction exacte s'est avérée utile pour établir nos bornes supérieures de complexité, elle nous semble trop stricte pour pouvoir prouver des résultats de complétude sous cette réduction. On peut aussi se poser la question de démontrer des théorèmes de hiérarchie non triviaux pour les classes enum(f, g).

Temps global / délai d'énumération

Il serait pertinent d'exhiber un problème, naturel ou non, dont l'ensemble des solutions soit énumérable, dans un ordre fixé ou non, en temps total linéaire (en l'entrée + la sortie), sans qu'il soit possible de le faire à délai $O(|S|)$, pour chaque solution S, après un précalcul en temps linéaire. Par exemple, peut-on trouver un problème naturel qui soit énumérable à délai constant et précalcul linéaire, pour un certain ordre, mais ne le soit pas (modulo une hypothèse de complexité) pour l'ordre inverse ? Un tel résultat serait à comparer avec le théorème suivant de Johnson, Papadimitriou et Yannakakis [JPY88] : on peut énumérer à délai polynomial et dans l'ordre lexicographique tous les stables maximaux d'un graphe, mais, si $\mathbf{P} \neq \mathbf{NP}$, on ne peut pas le faire dans l'ordre inverse.

Comptage / énumération

Le problème du comptage des solutions d'un problème est dans certains cas plus "difficile" que le problème de l'énumération de ses solutions : par exemple, comme on l'a évoqué ci-dessus, l'énumération des solutions d'une requête $\mathbf{CCQ}^{\neq}$ est de complexité "minimale" Constant-Delay$_{lin}$ alors que, sous l'hypothèse $\mathbf{FPT} \neq \#W[1]$, le comptage des solutions d'une telle requête n'est pas calculable par un algorithme **FPT** (le paramètre étant la taille de la formule).

Réciproquement, quand le problème du comptage des solutions d'un probème est facile, le problème de l'énumération de ces solutions est-il lui aussi facile ? Nous pensons que c'est largement vrai dans la pratique, en nous appuyant sur les trois classes de requêtes suivantes que nous avons étudiées dans cette thèse : les requêtes **MSO** sur les structures de largeur arborescente bornée, les requêtes **FO** sur les structures de degré borné, les requêtes **CCQ**. Dans les trois cas, le problème de comptage est linéaire et le problème de l'énumération est dans Lin-Delay$_{lin}$:

à chaque fois, les algorithmes de comptage et d'énumération sont efficaces car ils s'appuient sur des *propriétés structurelles fortes* de l'ensemble des solutions ; intuitivement et de façon très simplifiée, l'ensemble des solutions est une composition d'unions disjointes et de produits cartésiens. Dans ce contexte, il est pertinent de se poser la question suivante : contrairement à notre intuition, peut-on mettre en évidence un problème naturel pour lequel on peut trouver un algorithme efficace pour compter ses solutions mais pour lequel la question de leur énumération est difficile ?

Classes de requêtes à délai constant

Il serait intéressant d'exhiber de nouvelles classes de requêtes dans CONSTANT-DELAY$_{lin}$: un candidat possible est la classe des requêtes **FO** sur les classes de graphes linéairement et localement arbre-décomposables (avec certaines restrictions) qui sont déjà, comme l'a prouvé Frick [Fri04], calculables en temps total linéaire en la taille cumulée de l'entrée et de la sortie.

Bibliographie

Bibliographie

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AHU82] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, 1982.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2) :308–340, 1991.

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4) :844–856, 1995.

[Bag06] Guillaume Bagan. Mso queries on tree decomposable structures are computable with linear delay. In *Proc. of the Annual Conference of the European Association for Computer Science Logic (CSL)*, pages 167–181, 2006.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[BDGO08] Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *RAIRO*, 42 :147–164, 2008.

[Ber73] Claude Berge. *Graphs and hypergraphs.* North Holland, Amsterdam, 2nd edition, 1973.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3) :479–513, 1983.

[BG04] Régis Barbanchon and Etienne Grandjean. The minimal logically-defined np-complete problem. In Volker Diekert and Michel Habib, editors, *STACS 2004, 21st Annual Symposium on Theoretical Aspects of Computer Science, Montpellier, France, March 25-27, 2004, Proceedings*, volume 2996 of *Lecture Notes in Computer Science*, pages 338–349. Springer, 2004.

[BK96] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2) :358–402, 1996.

[BLM05] Andreas Brandstädt, Hoàng-Oanh Le, and Raffaele Mosca. Chordal co-gem-free and (5, gem)-free graphs have bounded clique-width. *Discrete Applied Mathematics*, 145(2) :232–241, 2005.

[Bod93] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11 :1–21, 1993.

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6) :1305–1317, 1996.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[CER90] Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Context-free handle-rewriting hypergraph grammars. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, 4th International Workshop, Bremen, Germany, March 5-9, 1990, Proceedings*, pages 253–268, 1990.

[Ces01] Marco Cesati. Compendium of parameterized problems, 2001.

[CH97] Nadia Creignou and Jean-Jacques Hébrard. On generating all solutions of generalized satisfiability problems. *ITA*, 31(6) :499–511, 1997.

[CK87] D.G. Corneil and P.A. Kamula. Extensions of permutation and interval graphs. Combinatorics, graph theory, and computing, Proc. 18th Southeast. Conf., Boca Raton/Fl. 1987, Congr. Numerantium 58, 267-275 (1987)., 1987.

[CKN+95] Derek G. Corneil, Hiryoung Kim, Sridhar Natarajan, Stephan Olariu, and Alan P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55(2) :99–104, 1995.

[CKSU05] Henry Cohn, Robert D. Kleinberg, Balázs Szegedy, and Christopher Umans. Group-theoretic algorithms for matrix multiplication. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 379–388. IEEE Computer Society, 2005.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, 2-4 May 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977.

[CM93] Bruno Courcelle and Mohamed Mosbah. Monadic second-order evaluations on tree-decomposable graphs. *Theor. Comput. Sci.*, 109(1&2) :49–82, 1993.

[CMR00] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory Comput. Syst.*, 33(2) :125–150, 2000.

[CO00] Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101(1-3) :77–114, 2000.

[Col07] Thomas Colcombet. Factorisation forests for infinite words. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings*, volume 4639 of *Lecture Notes in Computer Science*, pages 226–237. Springer, 2007.

[Cou87] Bruno Courcelle. An axiomatic definition of context-free rewriting and its application to nlc graph grammars. *Theor. Comput. Sci.*, 55(2-3) :141–181, 1987.

[Cou90a] Bruno Courcelle. Graph rewriting : An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 193–242. 1990.

[Cou90b] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1) :12–75, 1990.

[Cou92] Bruno Courcelle. The monadic second-order logic of graphs iii : tree-decompositions, minor and complexity issues. *ITA*, 26 :257–286, 1992.

[Cou06] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Maths*, 2006. To appear.

[CR00] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2) :211–229, 2000.

[CR05] Derek G. Corneil and Udi Rotics. On the relationship between clique-width and treewidth. *SIAM J. Comput.*, 34(4) :825–847, 2005.

[DF99] Rod G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.

[DFT96] Rod G. Downey, Michael R. Fellows, and Udayan Taylor. The parameterized complexity of relational database queries and an improved characterization of w [1]. In *Combinatorics, Complexity, and Logic – Proceedings of DMTCS '96*, pages 194–213. Springer-Verlag, 1996.

[DG07] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Transactions on Computational Logic*, 2007.

[DGK07] Anuj Dawar, Martin Grohe, and Stephan Kreutzer. Locally excluding a minor. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 270–279. IEEE Computer Society, 2007.

[Die05] Rod G. Diestel. *Graph Theory*. Springer-Verlag, 2005.

[DO06] Arnaud Durand and Frédéric Olive. First-order queries over one unary function. In *Proc. of the Annual Conference of the European Association for Computer Science Logic (CSL)*, volume 8, pages 334–348, 2006.

[EF99] Heinz-Dieter Ebbinghaus Ebbinghaus and Jorg Flum. *Finite Model Theory*. Springer Verlag, 2nd edition, 1999.

[EFT06] Heinz-Dieter Ebbinghaus, Jorg Flum, and Wolfgang Thomas. *Mathematical Logic*. Springer Verlag, 2006.

[Epp00] David Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3) :275–291, 2000.

[FFG02] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6) :716–752, 2002.

[FG01a] Jörg Flum and Martin Grohe. Fixed-parameter tractability, definability, and model-checking. *SIAM J. Comput.*, 31(1) :113–145, 2001.

[FG01b] Markus Frick and Martin Grohe. Deciding first-order properties of locally tree-decomposable structures. *J. ACM*, 48(6) :1184–1206, 2001.

[FG04a] Jörg Flum and Martin Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4) :892–922, 2004.

[FG04b] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1-3) :3–31, 2004.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[FGK03] Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees (extended abstract). In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 188–. IEEE Computer Society, 2003.

[FKN+04] Michael R. Fellows, Christian Knauer, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Ulrike Stege, Dimitrios M. Thilikos, and Sue Whitesides. Faster fixed-parameter tractable algorithms for matching and packing problems. In

Susanne Albers and Tomasz Radzik, editors, *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, Lecture Notes in Computer Science, pages 311–322. Springer, 2004.

[Fri04] Markus Frick. Generalized model-checking over locally tree-decomposable classes. *Theory Comput. Syst.*, 37(1) :157–191, 2004.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability. A guide to the theory of NP-completness*. Freeman, New York, NY, 1979.

[GKS06] Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2) :238–272, 2006.

[GLS01] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3) :431–498, 2001.

[GLS02] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3) :579–627, 2002.

[GM06] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *SODA*, pages 289–298, 2006.

[GMS07] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions : np-hardness and tractable variants. In *PODS*, pages 13–22, 2007.

[GO04] Etienne Grandjean and Frédéric Olive. Graph properties checkable in linear time in the number of vertices. *J. Comput. Syst. Sci.*, 68(3) :546–597, 2004.

[GR99] Martin Charles Golumbic and Udi Rotics. On the clique-width of perfect graph classes. In Peter Widmayer, Gabriele Neyer, and Stephan Eidenbenz, editors, *Graph-Theoretic Concepts in Computer Science, 25th International Workshop, WG '99, Ascona, Switzerland, June 17-19, 1999, Proceedings*, pages 135–147. Springer, 1999.

[Gra94a] Etienne Grandjean. Invariance properties of rams and linear time. *Computational Complexity*, 4 :62–106, 1994.

[Gra94b] Etienne Grandjean. Linear time algorithms and np-complete problems. *SIAM J. Comput.*, 23(3) :573–597, 1994.

[Gra96] Etienne Grandjean. Sorting, linear time and the satisfiability problem. *Ann. Math. Artif. Intell.*, 16 :183–236, 1996.

[Gro01] Martin Grohe. Generalized model-checking problems for first-order logic. In Afonso Ferreira and Horst Reichel, editors, *STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science, Dresden, Germany, February 15-17, 2001, Proceedings*, volume 2010 of *Lecture Notes in Computer Science*, pages 12–26. Springer, 2001.

[GS02] Etienne Grandjean and Thomas Schwentick. Machine-independent characterizations and complete problems for deterministic linear time. *SIAM J. Comput.*, 32(1) :196–230, 2002.

[GSS01] Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable ? In *STOC*, pages 657–666, 2001.

[HO07] Petr Hlinený and Sang Il Oum. Finding branch-decompositions and rank-decompositions. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, volume 4698 of *Lecture Notes in Computer Science*, pages 163–174. Springer, 2007.

[Imm06] Neil Immerman. *Descriptive Complexity*. Springer Verlag, 2006.

[JPY88] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3) :119–123, 1988.

[Kan08] M. M. Kante. *Structurations des graphes : quelques applications algorithmiques*. PhD thesis, Université de Bordeaux 1, 2008.

[Lib04] Leonid Libkin. *Elements of finite model theory*. Springer, 2004.

[Lin08] Steven Lindell. A normal form for first-order logic over doubly-linked data structures. *Int. J. Found. Comput. Sci.*, 19(1) :205–217, 2008.

[Loz07] Vadim V. Lozin. Clique-width of unit interval graphs, 2007.

[Loz08] Vadim V. Lozin. From tree-width to clique-width : Excluding a unit interval graph. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, volume 5369 of *Lecture Notes in Computer Science*, pages 871–882. Springer, 2008.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50, 1988.

[Mon85] B. Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25 :239–254, 1985.

[Mö90] R. H. Möhring. Graph problems related to gate matrix layout and pla folding. *Computational Graph Theory*, 1990.

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[NPTT05] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-ary queries by tree automata. In Gavin M. Bierman and Christoph Koch, editors, *Database*

Programming Languages, 10th International Symposium, DBPL 2005, Trondheim, Norway, August 28-29, 2005, Revised Selected Papers, volume 3774 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2005.

[Oum05] Sang Il Oum. Approximating rank-width and clique-width quickly. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, volume 3787 of *Lecture Notes in Computer Science*, pages 49–58. Springer, 2005.

[Pap94] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[PV90] Jürgen Plehn and Bernd Voigt. Finding minimally weighted subgraphs. In Rolf H. Möhring, editor, *Graph-Theoretic Concepts in Computer Science, 16rd International Workshop, WG '90, Berlin, Germany, June 20-22, 1990, Proceedings*, volume 484 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 1990.

[PY99] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3) :407–427, 1999.

[Rab64] Michael O. Rabin. Simple method for undecidability proofs and some applications. *Logic, Methodology and Philosophy of Science II*, pages 58–68, 1964.

[Rei01] Klaus Reinhardt. The complexity of translating logic to finite automata. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata, Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 231–238. Springer, 2001.

[RS84] Neil Robertson and Paul D. Seymour. Graph minors. iii. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1) :49–64, 1984.

[See96] Detlef Seese. Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science*, 6(6) :505–526, 1996.

[Seg08] Luc Segoufin. personal communication, 2008.

[STU97] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.*, 26(3) :678–692, 1997.

[Tuz92] Zsolt Tuza. Narrowness, pathwidth, and their application in natural language processing. *Discrete Applied Mathematics 36*, 1992.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1) :57–81, 1968.

[TY84] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579, 1984.

[TY85] Robert Endre Tarjan and Mihalis Yannakakis. Addendum : Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 14(1) :254–255, 1985.

[Uno98] Takeaki Uno. A new approach for speeding up enumeration algorithms. In Kyung-Yong Chwa and Oscar H. Ibarra, editors, *Algorithms and Computation, 9th International Symposium, ISAAC '98, Taejon, Korea, December 14-16, 1998, Proceedings*, pages 287–296. Springer, 1998.

[Uno01] Takeaki Uno. A fast algorithm for enumerating bipartite perfect matchings. In Peter Eades and Tadao Takaoka, editors, *Algorithms and Computation, 12th International Symposium, ISAAC 2001, Christchurch, New Zealand, December 19-21, 2001, Proceedings*, pages 367–379. Springer, 2001.

[Var82] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, 5-7 May 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982.

[Yan81] M. Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94, 1981.

Résumé

Cette thèse est consacrée à l'évaluation de requêtes logiques du point de vue de l'énumération. Nous étudions quatre classes de requêtes. En premier lieu, nous nous intéressons aux formules conjonctives acycliques avec inégalités. Nous prouvons que de telles requêtes logiques peuvent être évaluées à délai linéaire en la taille de la structure. Nous exhibons ensuite la sous-classe des formules connexe-acycliques pour lesquelles l'évaluation de requêtes s'effectue à délai constant après prétraitement linéaire. Nous montrons que cette classe est maximale pour ce résultat dans le sens suivant : si le produit de deux matrices booléennes ne peut pas être calculé en temps linéaire alors toute requête conjonctive acyclique est évaluable à délai constant après prétraitement linéaire si et seulement si elle est connexe-acyclique. En second lieu, nous démontrons que toute requête MSO sur une classe de structures de largeur arborescente bornée peut être évaluée à délai linéaire en la taille de chaque solution produite après un prétraitement linéaire en la taille de la structure. En troisième lieu, nous prouvons que, pour chaque requête en logique du premier ordre sur des structures de degré borné, il est possible de trouver en temps constant la j-ème solution dans un certain ordre après un prétraitement linéraire. Enfin, nous établissons que les graphes d'intervalles unitaires ont une largeur de clique localement bornée. D'où nous déduisons que tout énoncé du premier ordre sur ces graphes est décidable en temps linéaire ; là encore, nous démontrons une certaine maximalité de ce résultat.

Mots-clés: Complexité algorithmique, requête logique, logique du premier ordre ou du second ordre, base de données, requête conjonctive, model-checking, énumération, temps linéaire, délai constant, largeur arborescente, largeur de clique, hypergraphe, graphe d'intervalles unitaires.

Abstract

This thesis is dedicated to the evaluation of logical queries from the enumeration point of view. First, we deal with acyclic conjunctive formulas with inequalities ; we show that such a query can be evaluated with linear delay in the size of the structure : this improves a result by Papadimitriou and Yannakakis. Then, we exhibit a subclass of acyclic formulas, so-called connex-acyclic formulas. Such queries can be evaluated with constant delay after some linear time preprocessing. We show that this result is maximal in the following sense : if the product of two boolean matrices cannot be computed in linear time then any acyclic query is computable with constant delay after some linear time preprocessing if and only if it is connex-acyclic. Second, we prove that any MSO query over a class of bounded treewidth structures can be evaluated with a linear delay in the size of each solution after some linear preprocessing in the size of the structure. Third, we show that for each first-order query over bounded degree structures, one can compute the j-th solution of the query in constant time after some linear time preprocessing. Finally, we prove that unit interval graphs are of bounded local cliquewidth. Hence, we deduce that any first-order statement over these graphs is decidable in linear time ; also, we show that this result is somehow maximal.

Keywords: Computational complexity, logical query, first-order or second-order logic, database, conjunctive query, model-checking, enumeration, linear time, constant delay, treewidth, cliquewidth, hypergraph, unit interval graph.