UNIVERSITÉ DE MONTRÉAL

TECHNIQUES POUR L'EXPLORATION DE DONNÉES STRUCTURÉES ET POUR LA DÉCOUVERTE DE CONNAISSANCES EN THÉORIE DES GRAPHES

CHRISTIAN DESROSIERS
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(MATHÉMATIQUES DE L'INGÉNIEUR)
JUIN 2008

Your file Votre référence
ISBN: 978-0-494-46096-2
Our file Notre référence
ISBN: 978-0-494-46096-2

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

TECHNIQUES POUR L'EXPLORATION DE DONNÉES STRUCTURÉES ET POUR LA DÉCOUVERTE DE CONNAISSANCES EN THÉORIE DES GRAPHES

présentée par: DESROSIERS Christian

en vue de l'obtention du diplôme de: Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de:

M. GAGNON Michel, ing., Ph.D., président

M. HERTZ Alain, ing., Doct. ès Sc., membre et directeur de recherche

M. GALINIER Philippe, Doct., membre et codirecteur de recherche

M. VALTCHEV Petko, Ph.D., membre

M. KARYPIS George, Ph.D., membre externe

A mes parents

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur, Alain Hertz, pour ses conseils, sa disponibilité et sa bonne humeur lors des réunions. Son dévouement à sa profession a été une grande source d'inspiration et de motivation pour ma thèse. Je remercie également mon codirecteur, Philippe Galinier, pour son esprit vif et ses remarques fort utiles qui ont permis l'avancement de ma thèse. J'aimerais aussi remercier Pierre Hansen, qui a agi comme second codirecteur durant ma thèse. Son immense savoir, ainsi que les discussions philosophiques que nous avons eues, m'ont aidé à bien saisir l'essence de la découverte scientifique.

J'aimerais également remercier les professeurs Michel Gagnon, Petko Valtechev, George Karypis et Ettore Merlo d'avoir accepté de faire partie de mon jury de thèse. Merci en particulier au professeur Gagnon pour sa gentillesse lors de nos rencontres.

Je suis aussi très reconnaissant envers le Fonds québécois de la recherche sur la nature et les technologies (FQRNT) pour m'avoir octroyé une bourse de recherche, ainsi qu'envers la compagnie Irosoft et son président Alain Lavoie pour leur contribution financière. Je remercie du même coup mon directeur et codirecteur de recherche, les professeurs Hertz et Galinier, ainsi que le professeur Hansen, pour leur aide financière.

Je souhaite également remercier ma femme, June, pour son inspiration, ainsi que les membres de ma famille, Marieve, Philippe, Louise et Berthold, pour leur bons conseils et encouragements. Je remercie aussi tous mes amis, en particulier Jake, Amy, Anne-Marie, Luc, Thuy-Ly et Martin, ainsi que mes collègues de travail, Sandrine, Rim, Jean, Nizar, Imed, Daniel et Alain, pour les conseils précieux et tous les moments de détente.

Enfin, je remercie l'hiver québécois et sa rigueur, sans lesquels la tentation de laisser ma thèse au profit d'une activité de plein air aurait été trop forte.

RÉSUMÉ

L'automatisation de la découverte de nouvelles connaissances est un domaine fascinant de recherche allant de pair avec le progrès scientifique. Cette thèse porte sur deux disciplines de ce domaine ayant connu un vaste succès au cours des dernières années : l'exploration de données et la génération automatisée de conjectures en mathématiques. Nous présentons, dans un premier temps, des contributions à un problème important de l'exploration de données, connu sous le nom de la découverte des patrons fréquents. Ce problème, jouant un rôle clé dans plusieurs domaines tels que la bioinformatique, la chimie computationnelle ainsi que le Web, consiste à trouver les patrons que l'on retrouve fréquemment dans une base de données. Nous introduisons, dans cette thèse, des techniques permettant d'améliorer les méthodes existantes pour ce problème. En particulier, nous proposons un nouvel algorithme, appelé SYGMA, permettant de trouver efficacement les sous-graphes fréquents d'une base de graphes ayant peu d'étiquettes différentes. Nous présentons également une nouvelle approche à l'exploration des patrons fréquents, utilisant des connaissances de fond sur les patrons fréquents pour définir la topologie de l'espace de recherche de sorte à limiter le nombre de calculs coûteux. Enfin, nous montrons, à l'aide d'expériences numériques sur des instances générées et provenant d'applications réelles, l'efficacité de nos méthodes par rapport aux méthodes existantes.

Cette thèse présente, par ailleurs, des contributions significatives au problème de la génération de conjectures en théorie des graphes. Plus spécifiquement, nous introduisons des méthodes innovatrices permettant d'obtenir de manière automatique des conjectures portant sur la caractérisation par sous-graphes interdits (CSI). Étant donné une classe de graphes $\mathcal{C}$, une CSI est un ensemble de graphes $\mathcal{H}$ tel qu'un graphe G appartient à $\mathcal{C}$ si et seulement s'il ne contient aucun graphe de $\mathcal{H}$ comme sous-graphe induit. Les CSI jouent un rôle essentiel en théorie des graphes, étant au coeur de plusieurs résultats fondamentaux dans ce domaine, et permettant aussi de développer des algorithmes efficaces

pour reconnaître les graphes d'une classe donnée. Nous proposons, dans cette thèse, cinq algorithmes pour ce problème : deux algorithmes pour obtenir des conditions suffisantes pour avoir une CSI, deux algorithmes générant des conditions nécessaires pour une CSI et, enfin, un dernier algorithme permettant d'obtenir de vraies CSI. Nous utilisons ensuite ces algorithmes pour reproduire des résultats connus de la théorie des graphes, ainsi que pour en trouver de nouveaux.

ABSTRACT

The automation of the process of knowledge discovery is a fascinating field of research that has a deep impact on the progress of science. This thesis focuses on two disciplines of that field, which have enjoyed much success in the last few years : datamining and the automated generation of conjectures in mathematics. We first present contributions to an important problem in datamining, known as frequent pattern discovery. This problem, which plays a key role in various fields such as bioinformatics, computational chemistry and the Web, consists in finding the patterns that are frequently found in a database. We introduce, in this thesis, techniques that improve existing methods for this problem. In particular, we propose a new algorithm, called SYGMA, that efficiently finds the frequent subgraphs of a database containing graphs with a limited number of different labels. We also present a novel approach to frequent pattern mining, that uses background knowledge on the frequent patterns to remap the search space in a way that minimizes the number of time-expensive computations. Finally, we show, through numerical experiments on generated and real-life instances, the efficiency of our methods compared to existing ones.

This thesis also presents significant contributions to the problem of automated conjecture generation in the field of graph theory. More precisely, we introduce innovative methods that find, in an automated way, conjectures on a forbidden subgraph characterization (FSC). Given a class of graphs $\mathcal{C}$, an FSC is a set of graphs $\mathcal{H}$ such that a graph G belongs to $\mathcal{C}$ if and only if it does not contain any graph of $\mathcal{H}$ as an induced subgraph. FSCs play an essential role in graph theory, being at the centre of many famous results in that field, and allowing also to develop efficient algorithms to recognize the graphs of a given class. We propose, in this thesis, five algorithms for this problem : two algorithms to find sufficient conditions to have an FSC, two algorithms generating necessary conditions to have an FSC, and, finally, an algorithm that obtains actual FSCs. We then use these algorithms to reproduce known results of graph theory, as well as to find new ones.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES NOTATIONS ET DES SYMBOLES

AGM : Apriori Graph Mining

CSI : Caractérisation par Sous-graphes Interdits

DFS : Depth-First Search

DSPM : Diagonally Subgraph Pattern Mining

FFSM : Fast Frequent Subgraph Mining

FSC : Forbidden Subgraph Characterization

FSG : Frequent Sub-Graph mining

NFSC : Necessary conditions to a Forbidden Subgraph Characterization

PLI : Programmation Logique Inductive

PTE : Predictive Toxicology Evaluation

SFSC : Sufficient conditions to a Forbidden Subgraph Characterization

SYGMA : Symmetry-free Graph Mining Algorithm

VNS : Variable Neighborhood Search

$\text{sup}(X, \mathcal{D})$: support du patron X dans la base de données $\mathcal{D}$

$\text{freq}(X, \mathcal{D})$: fréquence du patron X dans la base de données $\mathcal{D}$

$\text{maxCS}(\mathcal{L})$: sous-graphes communs maximaux de l'ensemble $\mathcal{L}$

$\text{minCNS}(\mathcal{L})$: non sous-graphes communs minimaux de l'ensemble $\mathcal{L}$

$\text{emb}(H, G)$: nombre d'isomorphismes de G à un sous-graphe de H

$\mathcal{G}_{\mathcal{H}}$: graphes n'ayant aucun graphe de $\mathcal{H}$ comme sous-graphe induit

$V(G)$: ensemble des sommets du graphe G

$E(G)$: ensemble des arêtes du graphe G

$L(G)$: ensemble des étiquettes du graphe G

$l_v(u)$: étiquette du sommet u

$l_e(u, v)$: étiquette de l'arête (u, v)

$deg(v)$:	degré du sommet v
$D(G)$:	diamètre du graphe G
$n_1(G)$:	nombre de sommets pendants du graphe G
$\alpha(G)$:	taille du plus grand ensemble stable du graphe G
$r(G)$:	rayon du graphe G
$i(G)$:	taille du plus petit ensemble stable maximal du graphe G
$\gamma(G)$:	taille du plus petit ensemble dominant du graphe G
$\Gamma(G)$:	taille du plus grand ensemble dominant minimal du graphe G
$IR(G)$:	taille du plus grand ensemble irredondant du graphe G
$ir(G)$:	taille du plus petit ensemble irredondant maximal du graphe G
$\rho(G)$:	plus petit nombre de chaînes nécessaires pour recouvrir le graphe G
$Orb(v)$:	orbites du sommet v
$Orb(u,v)$:	orbites de la paire de sommets (u,v)
$Aut(G)$:	groupe d'automorphisme du graphe G
$\rho_g(\varphi)$:	généralisations du concept φ
$\rho_s(\varphi)$:	spécialisations du concept φ

INTRODUCTION

La découverte de connaissances est un processus complexe dont les fondements sont étudiés depuis fort longtemps. Récemment, l'arrivée de technologies permettant d'échanger et de stocker de plus en plus d'information, ainsi que l'augmentation remarquable de la puissance des ordinateurs, a permis l'apparition de systèmes informatiques automatisant, complètement ou en partie, le processus de découverte. À ce titre, deux disciplines ayant connu beaucoup de succès sont l'exploration de données et la génération automatisée de concepts en mathématiques. Les contributions présentées dans cette thèse portent sur deux problèmes importants de ces disciplines, soit la découverte des patrons fréquents et la génération automatisée de conjectures en théorie des graphes.

1.1 La découverte des patrons fréquents

L'exploration de données est une discipline située au croisement de l'informatique et des mathématiques, et dont le but est d'extraire, de manière automatisée, des connaissances utiles à partir d'une grande quantité de données. Un des problèmes importants de cette discipline est la découverte des patrons fréquents :

Définition 1 (Découverte des patrons fréquents). *Soit $\mathcal{D}$ une base de données, le* ***support*** *d'un patron X dans $\mathcal{D}$, noté $\sup(X, \mathcal{D})$ est le nombre d'objets de $\mathcal{D}$ contenant X. De même, la* ***fréquence*** *de X, notée $\text{freq}(X, \mathcal{D})$, correspond au ratio des objets de $\mathcal{D}$ contenant X, soit*

$$\text{freq}(X, \mathcal{D}) = \frac{\sup(X, \mathcal{D})}{|\mathcal{D}|}.$$

Étant donné un seuil minimum de fréquence, noté f_{min}, le problème de la découverte des patrons fréquents correspond à trouver l'ensemble $\mathcal{F}$ contenant les patrons dont la

fréquence dans $\mathcal{D}$ est supérieure ou égale à f_{min}, i.e.

$$\mathcal{F} = \{X \subseteq \mathcal{I} \mid \text{freq}(X, \mathcal{D}) \geq f_{min}\}.$$

*On appelle **patron fréquent** tout patron de l'ensemble $\mathcal{F}$.*

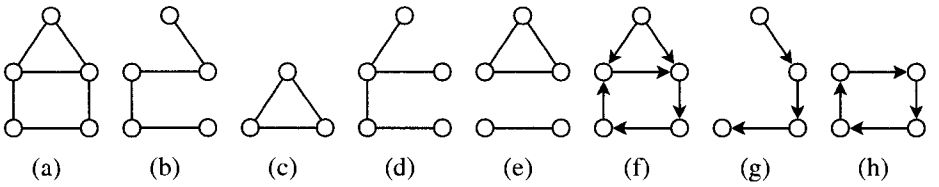
Un exemple bien connu de la découverte des patrons fréquents est la *découverte d'ensembles fréquents d'items* (Agrawal et al., 1993), pouvant être définie comme suit. Soit $\mathcal{I}$ un ensemble d'items, et soit $\mathcal{D}$ un ensemble de *transactions*, tel que chaque transaction $T \in \mathcal{D}$ est un ensemble d'items, i.e. $T \subseteq \mathcal{I}$. On dit qu'une transaction T contient un ensemble d'items $X \subseteq \mathcal{I}$ si $X \subseteq T$. Le support d'un ensemble d'items X correspond, dans ce cas, au nombre de transactions de $\mathcal{D}$ contenant X. Le but est alors de trouver tous les ensembles d'items X qui sont sous-ensembles d'un nombre suffisant de transactions, fixé par le seuil minimum de fréquence. On retrouve souvent ce problème dans le domaine du marketing et de la vente, où il est mieux connu sous le nom d'*analyse de panier de marché*. Dans ce contexte, on cherche typiquement à déterminer les habitudes d'achat des clients d'une compagnie, afin de développer des stratégies de vente. Ainsi, chaque transaction de la base de données représente un ensemble de produits achetés simultanément par un même client, et le but est d'identifier les produits qui sont souvent achetés ensemble. Ces ensembles fréquents d'items servent ensuite à obtenir des règles d'association du genre :

> "les clients qui achètent les produits A et B achètent également les produits C et D".

Une autre spécialisation importante de la découverte des patrons fréquents porte sur les graphes.

Définition 2 (Graphe). *Un **graphe** $G = (V, E)$ se compose d'un ensemble de sommets V et d'un ensemble d'arêtes $E \subseteq V \times V$ reliant deux sommets. De façon équivalente, on dit que $V(G)$ et $E(G)$ sont des ensembles contenant respectivement les sommets et les*

arêtes du graphe G. Par ailleurs, on dit qu'un graphe est ***dirigé*** *ou* ***orienté*** *si ses arêtes possèdent un sommet initial u et un sommet terminal v, i.e. $(u, v) \neq (v, u)$. De plus, un graphe dont deux mêmes sommets sont reliés par plusieurs arêtes est appelé* ***multi-graphe****, et on appelle* ***pseudo-graphe*** *un graphe ayant une arête reliant un sommet à lui-même. Un* ***graphe simple*** *est un graphe qui n'est ni un multi-graphe, ni un pseudo-graphe. Une* ***chaîne****, ou* ***chemin*** *dans le contexte des graphes dirigés, est une séquence de sommets $v_1, v_2, \ldots, v_n \in V$ reliés par des arêtes, i.e. $(v_i, v_{i+1}) \in E, 1 \leq i \leq n-1$. Un* ***cycle*** *(****circuit****) est une chaîne (chemin) dont les extrémités coïncident, i.e. $v_1 = v_n$. Un graphe est* ***connexe*** *s'il existe, pour toute paire de sommets, une chaîne ayant pour extrémités ces deux sommets. Finalement, un* ***arbre*** *est un graphe connexe non-dirigé n'ayant pas de cycle.*



Graphe	Dirigé	Connexe	Chaîne	Cycle	Chemin	Circuit	Arbre
(*a*)		×					
(*b*)		×	×				×
(*c*)		×		×			
(*d*)		×					×
(*e*)							
(*f*)	×	×					
(*g*)	×	×			×		
(*h*)	×	×				×	

Figure 1.1 Quelques exemples de graphes.

Puisqu'il ne sera question dans cette thèse que de graphes simples, nous appellerons graphe, à partir de maintenant, tout graphe simple. La figure 1.1 montre quelques exemples de graphes. On constate que certains de ces graphes peuvent être produits en retirant des sommets et des arêtes d'un autre graphe. On appelle *sous-graphes* de tels graphes.

Définition 3 (Sous-graphe). *Soit $G = (V, E)$ un graphe, le graphe $G' = (V', E')$ est un* ***sous-graphe*** *de G s'il peut être produit en retirant des sommets et des arêtes de G, i.e. si*

$$V' \subseteq V \quad \text{et} \quad \forall u, v \in V',\ (u, v) \in E' \Rightarrow (u, v) \in E.$$

On dit, par ailleurs, que G' est un ***sous-graphe induit*** *de G s'il peut être produit en retirant uniquement des sommets de G ainsi que les arêtes incidentes à ces sommets, i.e.*

$$V' \subset V \quad \text{et} \quad \forall u, v \in V',\ (u, v) \in E \Rightarrow (u, v) \in E'.$$

Dans la figure 1.1, les graphes (*b*) à (*e*) sont tous des sous-graphes de (*a*), alors que les graphes (*g*) et (*h*) sont des sous-graphes de (*f*). Par contre, seul le graphe (*c*) est un sous-graphe induit de (*a*), tandis les graphes (*g*) et (*h*) sont tous les deux des sous-graphes induits de (*f*).

Alors qu'ils permettent de décrire la topologie d'une structure, les graphes ne permettent pas de représenter entièrement des structures plus riches en information telles que les molécules et les documents structurés (e.g. documents *XML*). Les *graphes étiquetés* enrichissent les graphes en donnant aux sommets et aux arêtes un attribut appelé *étiquette*.

Définition 4 (Graphe étiqueté). *Un* ***graphe étiqueté*** *est un quintuplet $G = (V, E, L, l_v, l_e)$, où V est un ensemble de sommets, E un ensemble d'arêtes, L est un ensemble d'étiquettes, et les fonctions $l_v : V \rightarrow L$ et $l_e : E \rightarrow L$ sont des injections associant respectivement chaque sommet et chaque arête de G à une étiquette de L. Pour simplifier, on peut supposer que les étiquettes sont des nombres entiers, et écrire $G = (V, E, l_v, l_e)$, où $l_v : V \rightarrow \mathbb{N}$ et $l_e : E \rightarrow \mathbb{N}$. Un graphe étiqueté $G' = (V', E', l'_v, l'_e)$ est un sous-graphe de G s'il peut être produit en retirant des sommets et des arêtes de G, i.e. si*

1. $V' \subseteq V$.
2. $\forall u \in V',\ l'_v(u) = l_v(u)$.

3. $\forall u, v \in V',\ (u,v) \in E' \Rightarrow (u,v) \in E$.

4. $\forall e \in E',\ l'_e(e) = l_e(e)$.

De plus, G' est un sous-graphe ***induit*** *de G s'il peut être produit en retirant uniquement des sommets de G, i.e. si*

1. $V' \subset V$.

2. $\forall u \in V',\ l'_v(u) = l_v(u)$.

3. $\forall u, v \in V',\ (u,v) \in E \Rightarrow (u,v) \in E'$.

4. $\forall e \in E',\ l'_e(e) = l_e(e)$.

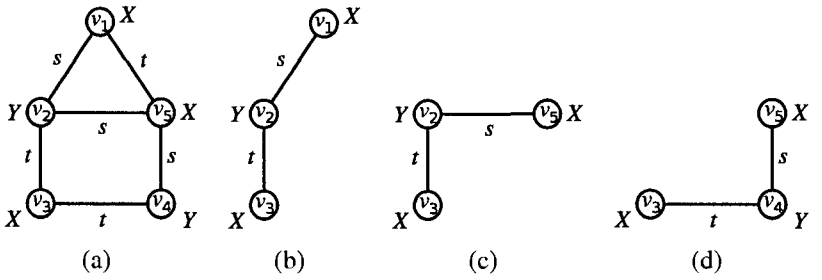


Figure 1.2 Quelques exemples de graphes étiquetés.

On montre, à la figure 1.2, un graphe étiqueté (a) ainsi que trois de ses sous-graphes induits (b), (c) et (d). Les sommets de ces graphes ont l'étiquette X ou Y, alors que les arêtes ont l'étiquette s ou t. Bien qu'ils proviennent de sous-parties différentes du graphe (a), on remarque que les trois sous-graphes sont topologiquement identiques. Dans les termes de la théorie des graphes, ont dit que ces graphes sont *isomorphes* :

Définition 5 (Isomorphisme). *Soit deux graphes étiquetés $G = (V, E, l_v, l_e)$ et $G' = (V', E', l'_v, l'_e)$, un* ***isomorphisme*** *de G à G' est une bijection $\varphi : V \longrightarrow V'$ telle que*

1. $\forall u \in V,\ l_v(u) = l'_v(\varphi(u))$.

2. $\forall u, v \in V,\ (u,v) \in E \Leftrightarrow (\varphi(u), \varphi(v)) \in E'$ *et* $l_e(u,v) = l'_e(\varphi(u), \varphi(v))$.

On dit que G et G' sont ***isomorphes****, noté $G \simeq G'$, s'il existe un isomorphisme φ de G à G'. On remarque que cette relation est symétrique, i.e. $G \simeq G' \Leftrightarrow G' \simeq G$ et φ^{-1} est un isomorphisme de G' à G. Par ailleurs, on dit qu'un graphe G contient un graphe G', noté $G' \subseteq G$, s'il existe un isomorphisme de G' à un sous-graphe de G. Enfin, un* ***automorphisme*** *est un isomorphisme allant d'un graphe à lui-même.*

La tâche de déterminer s'il existe un isomorphisme entre deux graphes est un problème célèbre, mieux connu sous le nom du problème d'isomorphisme de graphe (Fortin, 1996; McKay, 1981). Également, le problème d'isomorphisme de sous-graphe consiste à déterminer si un graphe est isomorphe à un sous-graphe d'un autre graphe. Alors que la complexité du problème d'isomorphisme de graphe n'est pas connue à ce jour, il existe plusieurs algorithmes efficaces pour résoudre ce problème, dont l'algorithme NAUTY développé par McKay (McKay, 1981). En revanche, le problème d'isomorphisme de sous-graphe a été démontré *NP*-complet (Garey et Johnson, 1990). Ce problème étant au coeur d'applications de différents domaines, dont celles traitées dans cette thèse, plusieurs méthodes exactes et heuristiques ont été proposées pour le résoudre, voir e.g. (Ullmann, 1976; Schmidt et Druffel, 1976; Cordella et al., 2004).

Considérons à nouveau la figure 1.2, où les graphes (*b*), (*c*) et (*d*) sont des sous-graphes du graphe (*a*), et cherchons des isomorphismes allant de (*b*) à un sous-graphe de (*a*). Ainsi, l'isomorphisme trivial, associant chaque sommet de (*b*) à lui-même est un tel isomorphisme. De plus, le graphe (*b*) est isomorphe aux graphes (*c*) et (*d*), par les isomorphismes φ_1 et φ_2 donnés par la table suivante :

i	$\varphi_1(v_i)$	$\varphi_2(v_i)$
1	v_5	v_5
2	v_2	v_4
3	v_3	v_3

Les bijections φ_1 et φ_2 sont donc aussi des isomorphismes de (*b*) vers un sous-graphe de

(a).

L'*exploration des sous-graphes fréquents* est une spécialisation de la découverte des patrons fréquents où l'on possède une base de données contenant des graphes étiquetés et où le but est de trouver tous les graphes qui sont isomorphes à un sous-graphe d'un nombre suffisant de graphes de la base de données. Ce problème, sur lequel porte cette thèse, joue un rôle clé dans des applications provenant de divers domaines, en particulier, dans la synthèse de nouveaux médicaments (Borgelt et al., 2005; Sternberg et al., 1995), la classification de composés chimiques (Deshpande et al., 2002), l'analyse et la prédiction de structures protéiques (Huan et al., 2004a; Zaki et al., 2004), l'indexation de données sous forme de graphes (Yan et al., 2005), l'exploration de documents et de requêtes XML (Yang et al., 2003; Chen et al., 2004; Termier et al., 2002; Punin et al., 2001), ainsi que l'exploration de réseaux biologiques (Koyuturk et al., 2004; Hu et al., 2005) et sociaux (Wasserman et Faust, 1994; Carrington et al., 2005).

1.2 La génération de conjectures en théorie des graphes

Le processus de découverte en sciences comporte essentiellement trois étapes : 1) l'observation d'un nouveau phénomène, 2) l'émission d'hypothèses expliquant ce phénomène, et 3) la vérification des hypothèses par expérimentation. En mathématiques, la découverte d'un nouveau théorème opère de manière similaire. On observe tout d'abord une relation mathématique paraissant nouvelle et utile. On tente ensuite de trouver un cas particulier du problème pour lequel la relation n'est plus vérifiée, i.e. un contre-exemple. Si un tel exemple ne peut être trouvé, on émet alors l'hypothèse que la relation découverte est vraie pour tous les cas. Une telle hypothèse porte le nom de conjecture. La dernière étape consiste à démontrer ou réfuter la conjecture. Si on parvient à lui trouver une preuve, la conjecture devient alors un théorème.

Depuis son invention, l'ordinateur a joué un rôle grandissant dans la découverte de nouveaux théorèmes en mathématiques, particulièrement, dans la génération automatisée de nouveaux concepts (Lenat, 1984; Colton, 1999; Larson, 2002; Bailey, 2000) et la démonstration automatisée de théorèmes (McCune, 1997; Wos, 1996). Un domaine pour lequel l'emploi de systèmes informatiques a été spécialement fécond est la théorie des graphes. Une illustration célèbre de ceci est la conjecture des quatre couleurs, datant de 1852, dont la preuve n'a été obtenue que récemment à l'aide d'ordinateurs (Appel et al., 1977; Robertson et al., 1997). Aujourd'hui, l'ordinateur est devenu un outil indispensable à la génération et la démonstration de conjectures dans ce domaine, comme le montre le document en ligne Written-on-the-wall (Fajtlowicz, 2008; Fajtlowicz et DeLaVina, 2008) contenant plus d'un millier de conjectures générées avec l'aide de l'ordinateur.

De façon générale, une conjecture est une relation entre plusieurs propriétés qui paraît vraie pour tous les éléments d'un ensemble donné. Notons P et Q deux prédicats représentant des propriétés portant sur les éléments d'un ensemble $\mathcal{X}$, tel que $P(x)$ et $Q(x)$ sont vraies si $x \in \mathcal{X}$ satisfait P et Q. Ces propriétés peuvent prendre diverses formes, telles que la conjonctions ou la disjonction d'autres propriétés, ou peuvent même vérifier l'appartenance d'un élément à une certaine classe. Avec ces propriétés, on peut exprimer trois types de relations :

1. Conditions suffisantes : $\forall x \in \mathcal{X}$, $P(x) \Rightarrow Q(X)$.
2. Conditions nécessaires : $\forall x \in \mathcal{X}$, $P(x) \Leftarrow Q(X)$.
3. Conditions nécessaires et suffisantes : $\forall x \in \mathcal{X}$, $P(x) \Leftrightarrow Q(X)$.

En théorie des graphes, on rencontre souvent des conjectures ayant la forme de relations algébriques portant sur les invariants de graphe. Un invariant i est une fonction qui associe à un graphe G une quantité $i(G)$, la plupart du temps réelle, qui est insensible à la numérotation des sommets. Par exemple, le diamètre d'un graphe G, que l'on note souvent $D(G)$, est un invariant correspondant à la plus grande distance séparant deux

sommets de G[1]. Soit $\mathcal{C}$ une classe graphes et $\mathcal{I} = \{i_1, i_2, \ldots, i_p\}$ un ensemble d'invariants définis sur les graphes de $\mathcal{C}$, une relation algébrique sur les invariants de $\mathcal{I}$ est de la forme :

$$\forall G \in \mathcal{C}, \quad \text{vrai} \Rightarrow f\left(i_1(G), i_2(G), \ldots i_p(G)\right) \begin{pmatrix} \geq \\ = \end{pmatrix} 0,$$

où f est une fonction linéaire ou non-linéaire sur l'espace des invariants. Soit un graphe G ayant m arêtes, notons $n_1(G)$ le nombre de sommets de G adjacents à un seul autre sommet, $\alpha(G)$ la taille du plus grand ensemble de sommets non-adjacents de G, appelé nombre de stabilité, et $r(G)$ la plus petite distance entre un sommet v de G et le sommet de G le plus éloigné de v, appelé le rayon. Un exemple de conjecture ayant la forme d'une relation algébrique linéaire, obtenue par Caporossi et Hansen avec l'aide de leur système AGX (Caporossi et Hansen, 2004), est la suivante :

pour tout arbre T d'index minimum[2], $2\alpha(T) - m - n_1(T) - D(T) = 0$.

Cette conjecture est, à ce jour, encore ouverte. De même, soit $\rho(G)$ le plus petit nombre de chaînes disjointes nécessaires pour recouvrir les sommets de G, appelé le nombre de recouvrement par chaînes, la conjecture suivante, obtenue par DeLaVina, Fajtlowicz et Waller à l'aide du système GRAFFITI (DeLaVina et al., 2005), est toujours ouverte :

pour tout graphe G, $\alpha(G) - r(G) - \ln(\rho(G)) \geq 0$.

Alors que les relations algébriques sur les invariants de graphe constituent le type de conjectures le plus étudié, tel que soulevé dans (Hansen et al., 2005), il existe bien d'autres sortes de résultats intéressants en théorie des graphes. Parmi ceux-ci se trouvent les résultats portant sur la caractérisation par sous-graphes interdits, qui décrit une classe de graphes en termes des sous-graphes que les graphes de cette classe ne peuvent pas avoir.

Définition 6 (Caractérisation par sous-graphes interdits (CSI)). *Soit $\mathcal{G}$ l'ensemble conte-*

[1]La distance entre deux sommets u, v est le plus petit nombre d'arêtes que l'on doit emprunter pour aller de u à v

[2]L'index (*ou rayon spectral*) d'un graphe G est la plus petite valeur propre de la matrice d'adjacence de G.

nant tous les graphes. Une ***classe*** *ou* ***famille*** *de graphes $\mathcal{C} \subseteq \mathcal{G}$ est une ensemble possiblement infini de graphes ayant une propriété commune. Étant donné un ensemble de graphes $\mathcal{H}$, on dit qu'un graphe G est sans $\mathcal{H}$, s'il n'y a aucun graphe de $\mathcal{H}$ isomorphe à un de ses sous-graphes induits, et écrivons $\mathcal{G}_{\mathcal{H}}$ l'ensemble de tous les graphes sans $\mathcal{H}$. Une* ***caractérisation par sous-graphes interdits*** *(CSI) d'une classe $\mathcal{C}$ est un ensemble de graphes $\mathcal{H}$ tel qu'un graphe G appartient à la classe $\mathcal{C}$ si et seulement si G est sans $\mathcal{H}$, i.e. $\mathcal{G}_{\mathcal{H}} = \mathcal{C}$.*

Les CSI jouent un rôle clé en théorie des graphes, permettant notamment le développement d'algorithmes polynomiaux pour reconnaître les graphes d'une classe donnée (Faudree et al., 1997), ou de trouver des liens hiérarchiques entre différentes classes de graphes (Brandstädt et al., 2003). De plus, les CSI sont à la base de résultats célèbres en théorie des graphes, comme la caractérisation des graphes parfaits (Berge, 1963), trouvée par Chudnovsky et al. (Chudnovsky et al., 2006), voulant qu'un graphe est parfait s'il ne possède pas comme sous-graphe induit un cycle impair contenant cinq sommets ou plus, ou son complément. Une autre CSI importante, venant de Beineke (Beineke, 1970), caractérise les graphes de ligne (Brandstädt et al., 1999) avec neuf sous-graphes interdits, montrés à la figure 1.3.

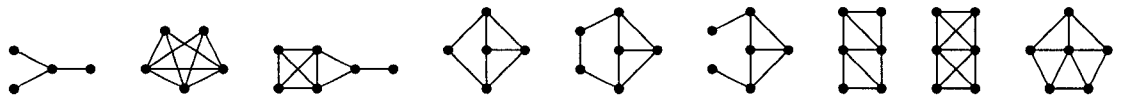


Figure 1.3 Une caractérisation par sous-graphes interdits des graphes de ligne.

Soit P le prédicat vérifiant qu'un graphe G est sans $\mathcal{H}$, et Q le prédicat vérifiant que G appartient à la classe de graphes $\mathcal{C}$. Une CSI est une relation ayant la forme d'une condition nécessaire et suffisante : $\forall G \in \mathcal{G},\ P(G) \Leftrightarrow Q(G)$. Cependant, la plupart des classes de graphes n'ont pas de CSI, et il est souvent intéressant de trouver des conditions suffisantes ou des conditions nécessaires pour qu'un ensemble de sous-graphes interdits

caractérise $\mathcal{C}$. Une condition suffisante est de la forme : si un graphe G est sans $\mathcal{H}$, alors il appartient à $\mathcal{C}$. Selon la notation présentée précédemment, une condition suffisante est donc un ensemble de graphes $\mathcal{H}$ tel que $\mathcal{G}_{\mathcal{H}} \subseteq \mathcal{C}$. On remarque qu'une condition suffisante ne permet pas de décrire entièrement la classe $\mathcal{C}$. En effet, si G n'est pas sans $\mathcal{H}$, on ne peut pas déterminer si G est dans $\mathcal{C}$ ou non. D'autre part, une condition nécessaire s'exprime de la manière suivante : si un graphe G est dans $\mathcal{C}$ alors il est sans $\mathcal{H}$. Ceci implique que $\mathcal{G}_{\mathcal{H}} \supseteq \mathcal{C}$. Une fois de plus, une condition nécessaire n'offre qu'une description partielle de $\mathcal{C}$: si G n'est pas dans $\mathcal{C}$ alors il peut être sans $\mathcal{H}$ ou non.

1.3 Problématique et objectifs de recherche

De manière générale, le problème de trouver les patrons fréquents dans une base de données se résume à deux tâches : 1) énumérer de manière unique chaque patron, et 2) calculer le support de ces patrons dans la base de données. Lorsque les patrons possèdent une structure complexe, il est souvent possible de représenter un même patron de différentes façons. Ainsi, un graphe possède un grand nombre de représentations topologiquement identiques, i.e. graphes isomorphes, obtenues en renumérotant ses sommets. On risque donc, en faisant l'énumération des patrons, d'explorer plusieurs patrons équivalents. Une façon de contourner ce problème consiste à trouver une représentation canonique d'un patron, i.e. une représentation optimisant un critère donné, et d'explorer un patron seulement si sa représentation est canonique. Cependant, dans bien des cas, trouver la représentation canonique d'un patron est un problème complexe. Par exemple, trouver la représentation canonique d'une paire de graphes est équivalent à résoudre le problème d'isomorphisme de graphe. De même, calculer le support d'un patron dans la base de données est, de façon générale, une opération coûteuse. Ainsi, calculer le support d'un graphe dans un ensemble de graphes revient à résoudre un grand nombre de fois le problème d'isomorphisme de sous-graphe qui est un problème *NP*-complet.

Dans le cas de la découverte des sous-graphes fréquents, la difficulté de trouver la représentation canonique d'un graphe et de calculer son support dans la base de données est grandement réduite lorsque les graphes de la base de données possèdent un grand nombre d'étiquettes de sommets et d'arêtes différentes. On peut ainsi, à l'aide de ces étiquettes, établir un ordre partiel des sommets d'un graphe, et, dans le calcul de la représentation canonique de ce graphe, se limiter aux permutations des sommets respectant cet ordre. Dans le cas extrême où tous les sommets ont des étiquettes différentes, on peut simplement ordonner les sommets en ordre croissant d'étiquette, et prendre la représentation canonique provenant de cet ordre. De même, une quantité importante d'étiquettes simplifie la tâche de déterminer si un graphe H de la base de données contient un certain graphe G. Ainsi, lorsqu'on cherche un sommet de H pouvant correspondre à un sommet v de G, on peut se limiter aux sommets qui ont la même étiquette que v et des arêtes incidentes ayant les mêmes étiquettes que les arêtes incidentes à v. En somme, la complexité de la découverte des sous-graphes fréquents augmente lorsqu'on travaille avec des graphes n'ayant pas ou ayant peu d'étiquettes différentes. Or, à ce jour, très peu d'attention a été portée à ce type de données, et les algorithmes actuels pour la découverte des sous-graphes fréquents sont d'une efficacité limitée pour de telles données. Cependant, il existe plusieurs applications utilisant ce genre de données, entre autres, dans le domaine de la vision par ordinateur où l'information est représentée par des maillages 2D ou 3D (Wong, 1992; Kim et al., 2006; Nakatsuji et al., 2005; Page et al., 2003), et dans les réseaux de transport et de communication, où l'information est essentiellement topologique (Zhu et al., 2003; Reittu et Norros, 2007). De plus, explorer des graphes sans étiquette permettrait de trouver des relations de la base de données qui sont à la fois plus générales et fréquentes. Le premier objectif de cette thèse est d'améliorer les méthodes existantes pour la découverte des sous-graphes fréquents, dans le cas où les graphes de la base de données ont peu d'étiquettes.

Lorsque la base de données comporte un grand nombre de structures, ce qui est sou-

vent le cas, le calcul du support des patrons est de loin l'opération la plus coûteuse de la découverte des patrons fréquents. Pour réduire le coût associé à cette tâche, les méthodes actuelles emploient essentiellement deux approches. La première approche consiste à calculer de manière incrémentale les incorporations d'un patron dans la base de données, e.g. isomorphismes de sous-graphe dans la découverte des sous-graphes fréquents. Cette technique est généralement employée lors d'une exploration en profondeur de l'espace de recherche. Supposons que l'on génère un patron Y en prolongeant un patron X dont les incorporations dans la base de données ont déjà été calculées. Comme toute structure contenant Y contient également X, on peut obtenir les incorporations de Y simplement en prolongeant celles de X. Alors que cette technique fonctionne bien pour les bases de données de taille réduite, elle est inefficace pour des bases de données plus grosses ou lorsque les données présentent une importante symétrie, e.g. graphes non-étiquetés. Dans ces cas, le calcul incrémental du support demande une quantité énorme de mémoire, et la mise-à-jour des incorporations est très longue. La seconde approche pour réduire les coûts engendrés par la calcul du support se base sur le principe *Apriori*. Puisque le support d'un patron n'est jamais supérieur à celui de ses sous-patrons, un patron est fréquent seulement si tous ses sous-patrons le sont. Si on connait le support de tous les sous-patrons d'un patron X, on peut éviter de calculer le support de X si un de ses sous-patrons n'est pas fréquent. Le problème de cette approche est qu'elle demande de visiter tous les sous-patrons d'un patron X avant de visiter X. Les deux techniques généralement employées pour ce problème est d'explorer l'espace de recherche par niveau, i.e. explorer les patrons de taille k avant ceux de taille k+1, ou de coder les patrons de sorte que l'exploration de l'espace des codes ne visite un patron qu'après ses sous-patrons. Or, ces techniques possèdent également des désavantages. La première souffre, en général, de la génération d'un grand nombre de patrons équivalents, et demande beaucoup de mémoire. De même, il peut être difficile de trouver un code pouvant être utilisé pour la seconde technique. La seconde technique demande, par ailleurs, de calculer la représentation canonique de tous les sous-patrons d'un patron, ce qui peut

être très coûteux. Le second objectif de cette recherche consiste à développer de nouvelles techniques pour réduire le nombre de calculs de support, sans avoir recours à des structures de données nécessitant beaucoup de mémoire.

Comme mentionné précédemment, les conjectures les plus étudiées en théorie des graphes ont la forme de relations algébriques entre les invariants d'un graphe. En conséquence, la vaste majorité des systèmes pour la génération automatisée de conjectures dans ce domaine produisent des conjectures de ce type. Cependant, tel que soulevé dans (Hansen et al., 2005), il existe plusieurs autres formes de résultats en théorie des graphes pour lesquels il serait intéressant d'automatiser la génération, dont la caractérisation par sous-graphes interdits (CSI). Or, tandis que certaines méthodes ont été proposées pour trouver des CSI particuliers, e.g. la CSI de l'intersection de plusieurs classes de graphes (Barrus, 2004), aucune de ces méthodes ne permet la génération automatisée de conjectures portant sur la CSI. Le troisième objectif de cette thèse est de développer un tel système. En plus de pouvoir générer des conjectures sur une CSI, ce système devra également permettre de générer des conditions suffisantes et des conditions nécessaires pour avoir une CSI. Enfin, le quatrième et dernier objectif de cette thèse est d'utiliser ce système pour générer de nouvelles conjectures, et de valider ces conjectures en les démontrant.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

Ce chapitre fait un survol des principaux travaux et résultats de la littérature portant sur la découverte des patrons fréquents et la génération automatisée de conjectures en théorie des graphes.

2.1 La découverte des concepts intéressants

Le problème de la découverte des patrons fréquents, sur lequel porte une partie de cette thèse, s'inscrit à l'intérieur d'un problème plus général formulé par Mannila et Toivonen (Mannila et Toivonen, 1997). Ce problème, connu sous le nom de la *découverte des concepts intéressants*, peut être formulé comme suit.

Définition 7 (Découverte de concepts intéressants). *Soit une base de données $\mathcal{D}$, un langage $\mathcal{L}$ exprimant des concepts ou hypothèses quelconques, et un prédicat de sélection P, trouver une théorie de $\mathcal{D}$ selon $\mathcal{L}$ et P, i.e. l'ensemble*

$$Th(\mathcal{D}, \mathcal{L}, P) = \{\varphi \in \mathcal{L} \mid P(\mathcal{D}, \varphi) = \text{vrai}\}.$$

Ainsi, $Th(\mathcal{D}, \mathcal{L}, P)$ renferme tous les concepts ou hypothèses du langage $\mathcal{L}$ qui sont intéressants selon le prédicat P. En pratique, la richesse du langage $\mathcal{L}$ peut induire un espace de concepts énorme ou même infini, rendant compliquée la recherche des concepts intéressants. Il est souvent nécessaire de limiter la richesse de $\mathcal{L}$ à l'aide d' un ensemble de contraintes, que l'on nomme *biais déclaratif*. Ce biais spécifie les propriétés que doivent avoir les concepts intéressants. Cependant, même dans un espace restreint, la re-

cherche ne peut se faire efficacement sans la présence d'un ordre utile. Ainsi, la définition de la relation de spécialisation des concepts est à la base des algorithmes efficaces pour la recherche des concepts intéressants.

Définition 8 (Généralisation et spécialisation des concepts). *Une relation de* ***spécialisation*** *est un ordre partiel $\preceq$ sur les concepts de $\mathcal{L}$. On dit qu'un concept φ est aussi général qu'un autre concept θ si $\varphi \preceq \theta$. De façon équivalente, on dit alors que θ est aussi spécifique que φ. On note $\varphi \prec \theta$ la relation de spécialisation telle que $\varphi \preceq \theta$ et non $\theta \preceq \varphi$.*

La relation $\preceq$ est une relation ***monotone*** *de spécialisation selon le prédicat P si pour chaque $\mathcal{D}$ et φ on a : si $\varphi \preceq \theta$ et $P(\mathcal{D}, \varphi) = \text{vrai}$ alors $P(\mathcal{D}, \theta) = \text{vrai}$. À l'inverse, $\preceq$ est une relation* ***anti-monotone*** *de spécialisation selon P si pour chaque $\mathcal{D}$ et φ on a : si $\varphi \preceq \theta$ et $P(\mathcal{D}, \theta) = \text{vrai}$ alors $P(\mathcal{D}, \varphi) = \text{vrai}$. Une relation monotone selon P est anti-monotone selon $\neg P$.*

Définition 9 (Concept minimal et maximal). *Soit une relation de spécialisation $\preceq$ sur $\mathcal{L}$, et soit $\Pi \subseteq \mathcal{L}$ un ensemble de concepts, on définit $\min(\Pi)$ comme l'ensemble des concepts les plus généraux de Π, i.e.*

$$\min(\Pi) = \{\varphi \in \Pi \mid \nexists \theta \in \Pi \text{ t.q. } \theta \prec \varphi\},$$

et nous appelons ***minimal*** *tout concept de cet ensemble. De même, notons $\max(\Pi)$ l'ensemble des concepts les plus spécifiques de Π, i.e.*

$$\max(\Pi) = \{\varphi \in \Pi \mid \nexists \theta \in \Pi \text{ t.q. } \varphi \prec \theta\},$$

et nommons ***maximaux*** *les concepts de cet ensemble.*

La relation de spécialisation induit ainsi une structure dans l'espace de recherche, appelée *graphe de spécialisation* ou *diagramme de Hasse* . Cette structure, prenant souvent

la forme d'un treillis[1], peut se représenter comme un graphe dirigé acyclique dont les noeuds sont les concepts de $\mathcal{L}$, les arcs correspondent à des raffinements (i.e. spécialisations) de concepts, et la racine est le concept le plus général, noté $\perp$. Ce graphe peut être parcouru à l'aide d'opérateurs permettant, à partir d'un noeud, d'explorer les noeuds correspondant à des concepts plus spécifiques ou plus génériques.

Définition 10 (Opérateurs de spécialisation et généralisation). *Étant donné un langage de concepts $\mathcal{L}$, un* ***opérateur de spécialisation*** ρ_s *associe à chaque concept φ un ensemble $\rho_s(\varphi)$ contenant les concepts spécialisant φ : $\rho_s(\varphi) = \{\theta \in \mathcal{L} \mid \varphi \prec \theta\}$. Un opérateur ρ_s est dit* ***direct*** *ou* ***immédiat*** *s'il n'associe à φ que les concepts plus spécifiques les plus généraux, i.e. $\rho_s(\varphi) = \min(\rho_s(\varphi))$. De même, un* ***opérateur de généralisation*** ρ_g *associe à chaque concept φ l'ensemble $\rho_g(\varphi) = \{\theta \in \mathcal{L} \mid \theta \prec \varphi\}$ qui contient les concepts généralisant φ. Un opérateur ρ_g est* ***direct*** *ou* ***immédiat*** *s'il associe à φ que les concepts plus généraux les plus spécifiques, i.e. $\rho_g(\varphi) = \max(\rho_g(\varphi))$.*

Pour illustrer ces notions, retournons au problème de trouver les ensembles fréquents d'une base de données contenant des ensembles d'items. Dans ce contexte, le langage des concepts correspond à tous les ensembles possibles d'items, i.e. $\mathcal{L} = \{T \subseteq \mathcal{I}\}$. De plus, la relation de spécialisation sur deux ensembles d'items est la relation d'inclusion. Soit $\varphi = X$ et $\theta = Y$ deux ensembles d'items, on a $\varphi \preceq \theta$ si et seulement si $X \subseteq Y$. Cette relation engendre un graphe de spécialisation de la forme d'un treillis booléen contenant tous les sous-ensembles (*power set*) de $\mathcal{I}$. Enfin, un ensemble d'items est intéressant s'il est fréquent. La figure 2.1 montre le graphe de spécialisation pour $\mathcal{I} = \{A, B, C, D\}$. Au bas se trouve l'ensemble vide. La spécialisation se fait vers le haut, en ajoutant à chaque niveau un item de plus, jusqu'à l'ensemble $\mathcal{I}$. Ainsi, l'opérateur direct de spécialisation ρ_s sur un ensemble d'items X consiste à ajouter à X un item qu'il ne

[1] On parle de semi-treillis si on ne peut garantir l'existence que d'un seul élément parmi le *suprémum* et l'*infimum* d'une paire de concepts.

possède pas, i.e.

$$\rho_s(X) = \{Y \subseteq \mathcal{I} \mid X \subset Y \text{ et } |Y| = |X| + 1\},$$

alors que l'opérateur direct de généralisation enlève un élément à X

$$\rho_g(X) = \{Y \subseteq \mathcal{I} \mid Y \subset X \text{ et } |Y| = |X| - 1\}.$$

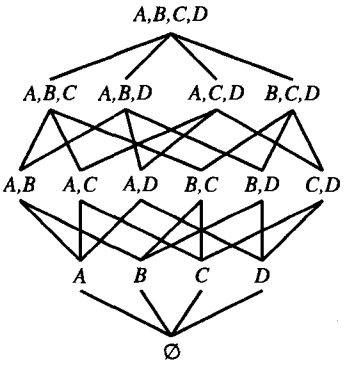


Figure 2.1 Le graphe de spécialisation des sous-ensembles d'items de l'ensemble $\mathcal{I} = \{A, B, C, D\}$.

Les méthodes de résolution pour la découverte des concepts intéressants peuvent être classées selon leur façon de parcourir le graphe de spécialisation. Les méthodes *ascendantes* débutent la recherche avec les concepts les plus généraux du langage, i.e. $\varphi_g \in \min(\mathcal{L})$, et utilisent un opérateur de spécialisation pour ensuite explorer des concepts de plus en plus spécifiques. À l'inverse, les approches *descendantes* débutent avec les concepts le plus spécifiques du langage, i.e. $\varphi_s \in \max(\mathcal{L})$, et explorent ensuite des concepts de plus en plus généraux avec un opérateur de généralisation. Finalement, un troisième type de méthodes, basé sur la *recherche d'espaces de versions* (Mitchell, 1982), combine les stratégies de recherche ascendante et descendante. Ces méthodes définissent l'espace des concepts valides, appelé *espace de versions*, à l'aide de deux

ensembles, $\mathcal{G}$ et $\mathcal{S}$, contenant respectivement les concepts valides les plus généraux et les plus spécifiques. Cet espace est ensuite raffiné en spécialisant les concepts de $\mathcal{G}$ et, en parallèle, généralisant les concepts de $\mathcal{S}$, jusqu'à la convergence de ces deux ensembles. Par ailleurs, les méthodes de résolution peuvent être subdivisées plus finement selon l'ordre dans lequel sont explorés les concepts du graphe de spécialisation. Ainsi, dans la recherche *en largeur*, tous les concepts ayant le même niveau de généralité sont explorés avant les concepts plus spécifiques, dans le cas des approches ascendantes, ou avant les concepts plus généraux, dans le cas des approches descendantes. Comme la recherche en largeur procède par niveau de généralité, on lui donne souvent le nom de *recherche par niveau.* Contrairement à la recherche par niveau, la recherche *en profondeur* explore toutes les spécialisations, dans le cas des approches ascendantes, ou toutes les généralisations, dans le cas des approches descendantes, d'un concept avant d'explorer un autre concept du même niveau de généralité. Les méthodes de résolution sont également divisées en méthodes *complètes* et *heuristiques*. Les méthodes complètes sont celles qui garantissent de trouver tous les concepts intéressants. En pratique, l'espace de recherche peut être trop vaste pour une exploration complète, et seulement une fraction de cet espace, composée typiquement des concepts qui paraissent les plus intéressants, est explorée. Ces méthodes font généralement appel à des heuristiques pour choisir les concepts à explorer.

Étant donné le nombre exponentiel de noeuds, typiquement observé dans les graphes de spécialisation, il est nécessaire de fournir aux méthodes de recherche un moyen d'éliminer de leur exploration les noeuds qui n'ont aucune chance d'être intéressants, un processus appelé *élagage*. Une des principales techniques pour élaguer l'espace de recherche, consiste à vérifier la fermeture descendante d'un concept par rapport au prédicat P.

Définition 11 (Fermeture descendante). *Soit un espace de recherche défini par la relation de spécialisation $\preceq$ et soit θ un concept de cet espace. On dit que θ est fermé vers le*

bas, par rapport à P, si tout concept $\varphi \prec \theta$ est intéressant selon P.

Propriété 1. *Soit $\mathcal{D}$ une base de données et P un prédicat de sélection anti-monotone selon la relation de spécialisation $\preceq$. De plus, soient φ et θ deux concepts tels que $\varphi \preceq \theta$. Si φ n'est pas intéressant, alors θ ne peut pas être intéressant. Plus formellement, on a*

$$\forall\, \varphi, \theta \in \mathcal{L}\ ,\ \varphi \preceq \theta \wedge \neg P(\mathcal{D}, \varphi) \Rightarrow \neg P(\mathcal{D}, \theta).$$

En d'autres termes, un concept θ est intéressant seulement si tous les concepts plus généraux $\varphi \preceq \theta$ sont intéressants. Ce principe, connu sous le nom du principe *Apriori* permet donc de filtrer certains concepts inintéressants sans avoir à évaluer directement le prédicat P, et d'élaguer tous les concepts plus spécifiques qu'un concept inintéressant. Illustrons une fois de plus ce principe dans le contexte de la découverte d'ensembles fréquents d'items. Soit deux ensembles d'items X et Y tel que $X \subset Y$. Si X n'est pas fréquent, alors Y n'est pas fréquent, puisque pour chaque occurrence de Y dans $\mathcal{D}$, il y a nécessairement une occurrence de X, i.e. $\text{freq}(X, \mathcal{D}) \geq \text{freq}(Y, \mathcal{D})$. Il n'est donc pas nécessaire de spécialiser X lors de la recherche des ensembles fréquents. De même, un ensemble Y est infréquent si au moins un de ses sous-ensembles $X \subset Y$ est infréquent. On peut alors détecter facilement les ensembles infréquents, et éviter de calculer leur support dans la base de données. Ces idées sont à la base de l'algorithme APRIORI, proposé par Agrawal et al. (Agrawal et al., 1993). Cet algorithme, dont une version de base est montrée à la figure 2.2, prend en entrée une base de données $\mathcal{D}$ ainsi qu'un seuil minimum de support s_{min}, et obtient les sous-ensembles fréquents d'items dans $\mathcal{D}$ à l'aide d'une recherche par niveau. En débutant au premier niveau avec un ensemble candidat $\mathcal{C}_1$ contenant tous les items de $\mathcal{I}$, le support des ensembles candidats est obtenu en traversant la base de données et en calculant le nombre de transactions les contenant. Les ensembles candidats qui sont fréquents, i.e. dont le support est au moins s_{min}, sont alors ajoutés à $\mathcal{F}_k$. Les ensembles candidats de k+1 items, i.e. $\mathcal{C}_{k+1}$, sont ensuite générés

Algorithme APRIORI

Entrées: Une base de données $\mathcal{D}$ et un seuil de support minimum s_{min}.
Sorties: Les sous-ensembles fréquents d'items $\mathcal{F}$.

$\mathcal{F} := \emptyset$;
$\mathcal{C}_1$:= l'ensemble des items $\mathcal{I}$;
$k := 1$;
tant que $\mathcal{C}_k \neq \emptyset$ **faire**
  $\mathcal{F}_k := \emptyset$;
  pour chaque $X \in \mathcal{C}_k$ **faire**
    si $|\{Y \in \mathcal{D} \mid X \subseteq Y\}| \geq s_{min}$ **alors** $\mathcal{F}_k := \mathcal{F}_k \cup \{X\}$;
  $\mathcal{F} := \mathcal{F} \cup \mathcal{F}_k$;
  $\mathcal{C}_{k+1}$:= APRIORI-gen ($\mathcal{F}_k$) ;
  $k := k + 1$;
retourner $\mathcal{F}$;

Procedure APRIORI-gen ($\mathcal{F}_k$)

Entrées: Les sous-ensembles fréquents de k items $\mathcal{F}_k$
Sorties: Les sous-ensembles candidats de k+1 items $\mathcal{C}_{k+1}$

$\mathcal{C}_{k+1} := \emptyset$;
pour chaque $X, Y \in \mathcal{F}_k$, *t.q.* $|X \cap Y| = |X| - 1$ **faire**
  Créer l'ensemble de k+1 items $Z = X \cup Y$;
  si *chaque sous-ensemble de i items de Z est dans* $\mathcal{F}_k$ **alors**
    $\mathcal{C}_{k+1} := \mathcal{C}_{k+1} \cup \{Z\}$;
retourner $\mathcal{C}_{k+1}$;

Figure 2.2 L'algorithme APRIORI pour la découverte d'ensembles fréquents, et sa procédure APRIORI-gen pour la génération d'ensembles candidats.

par la procédure APRIORI-gen en prenant l'union deux ensembles fréquents X et Y de k items, qui ne diffèrent que d'un seul item. L'ensemble généré $Z = X \cup Y$ possède alors k+1 items. La procédure élimine ensuite tous les candidats contenant un sous-ensemble de k items qui n'est pas fréquent, i.e. qui n'est pas dans $\mathcal{F}_k$. Enfin, l'algorithme parcourt la base de données pour calculer le support des candidats de $\mathcal{C}_{k+1}$, et ne conserve que ceux dont le support est au moins s_{min}. Ce processus se termine lorsqu'il n'y a plus de candidats, i.e. lorsque $\mathcal{C}_k = \emptyset$.

2.2 La découverte des requêtes fréquentes

Nous avons vu, à la section précédente, un exemple de la découverte des concepts intéressants qui consiste à trouver les ensembles fréquents d'items d'une base de données. Une autre application importante de ce problème, également basée sur la notion de fréquence, est la *découverte des requêtes fréquentes* dans une base de données multi-relationnelles (Džeroski, 2003). Cette application, ayant des liens étroits avec la *programmation logique inductive* (PLI) (Dzeroski et Lavrac, 1994; Muggleton, 1992), trouve, dans une base de données ayant typiquement la forme d'un *programme logique* (Bratko, 2001), des expressions en logique du premier ordre, appelées requêtes.

Définition 1 (Requête en logique du premier ordre). *Soit une clause C, une* ***substitution*** $\theta = \{X_1/a_1, \ldots, X_m/a_m\}$ *est un ensemble de paires variable/terme, telle que l'application de θ à C, notée $C\theta$, correspond à la clause C dans laquelle chaque variable X_i prend la valeur a_i. Soit un programme logique $\mathcal{D}$, une* ***requête*** *Q est une expression logique de la forme*

$$Q = ? - A_1, A_2, \ldots, A_n$$

correspondant à la conjonction des atomes A_i. La ***réponse*** *à la requête Q, notée* $\text{reponse}(Q, \mathcal{D})$, *est l'ensemble des substitutions qui vérifient Q dans $\mathcal{D}$, i.e.*

$$\text{reponse}(Q, \mathcal{D}) = \{\theta \mid \mathcal{D} \vDash Q\theta\},$$

*où $\vDash$ est l'****implication logique*** *qui est satisfaite si et seulement si toutes les substitutions qui satisfont l'expression de gauche satisfont également l'expression de droite.*

Comme vérifier l'implication logique est un problème *NP*-complet, trouver les réponses à une requête dans un programme logique, un processus basé sur l'*inférence logique* (Bratko, 2001), est *NP*-difficile.

Définition 2 (Découverte de requêtes fréquentes). *Soit $\mathcal{D}$ une base de données sous la forme d'un programme logique et $\mathcal{L}$ un langage d'hypothèses définissant les requêtes pouvant être formulées. Soit $Q \in \mathcal{L}$ une requête contenant un atome K appelé* clé, *la fréquence de Q dans $\mathcal{D}$ selon K est*

$$\text{freq}(Q, K) = \frac{|\text{reponse}(K, \mathcal{D}) \cap \text{reponse}(Q, \mathcal{D})|}{|\text{reponse}(K, \mathcal{D})|},$$

i.e. la fraction des substitutions vérifiant K dans $\mathcal{D}$ qui vérifient également Q. Étant donné un seuil minimum de fréquence f_{min} et une clé K, la découverte des requêtes fréquentes *consiste à trouver l'ensemble $\mathcal{F}$ des requêtes qui sont fréquentes dans $\mathcal{D}$ selon K, soit l'ensemble*

$$\mathcal{F} = \{Q \in \mathcal{L} \mid \text{freq}(Q, K) \geq f_{min}\}.$$

Afin d'illustrer ces concepts, considérons une base de données renfermant la définition des prédicats *étudiant*, *cours* et *inscrit*, fournissant de l'information sur des étudiants, des cours disponibles, et l'inscription des étudiants à ces cours. Supposons que l'on choisisse *étudiant*(X) pour être la clé, la requête

$$? - etudiant(X),\ inscrit(X, \text{algebre}),\ inscrit(X, \text{physique}).$$

est fréquente si la proportion d'étudiants inscrits aux cours d'algèbre et de physique est supérieure ou égale à f_{min}. Par contre, si on choisit *cours*(Y) comme clé, la requête

$$? - cours(Y),\ etudiant(X),\ inscrit(X, Y),\ inscrit(X, \text{algebre}).$$

est fréquente si la proportion de cours ayant des étudiants également inscrits au cours d'algèbre est au moins f_{min}.

Comme nous avons vu, l'espace de recherche de la découverte des concepts intéressants est défini à l'aide d'un ordre sur les concepts. Dans le contexte de la découverte des requêtes fréquentes, un ordre naturel, appelé *généralité sémantique* (Buntine, 1988), repose sur l'implication logique. Cependant, comme vérifier l'implication logique est difficile, on utilise plutôt la *généralité syntaxique*, basée sur la notion de *subsomption* (Plotkin, 1969).

Définition 3 (Subsomption). *Soit C et C' deux clauses, on dit que C θ-*subsume *C' si et seulement s'il existe une substitution θ tel que $C\theta \subseteq C'$. Par ailleurs, si C θ-subsume C' alors $C \models C'$, tandis que l'inverse n'est pas nécessairement vrai.*

La subsomption permet ainsi de définir la relation de spécialisation telle que $C \preceq C'$ si et seulement si C *θ-subsume* C'. Cette relation induit un espace de recherche qui peut être représenté sous la forme d'un graphe de spécialisation qui a pour base la clé, et pouvant être parcouru à l'aide de l'opérateur de spécialisation

$$\rho_s(C) = \{C' \mid \exists\theta \text{ t.q. } C\theta \subset C'\}.$$

Comme c'est le cas pour la plupart des concepts, on utilise en pratique un opérateur direct de spécialisation, i.e. qui n'obtient que les concepts plus spécifiques les plus généraux. Dans le cas qui nous intéresse, cet opérateur spécialise une requête en lui ajoutant un nouvel atome. Cependant, deux requêtes différentes peuvent être équivalentes sous la subsomption. Par exemple, la requête

$$? - p(X, a),\ q(X),\ q(X),\ p(Y, a),\ q(Y),\ q(Y)$$

est équivalente à la requête plus simple $? - p(X, a),\ q(X)$ sous la substitution $\theta = \{X/Y\}$. Ainsi, on exige normalement un opérateur qui n'obtient que les spécialisations directes qui sont des *représentants minimaux* des classes d'équivalence définies sous la subsomption.

L'algorithme WARMR, proposé par Dehaspe et Toivonen (Dehaspe et Toivonen, 1999), est une extension de l'algorithme APRIORI pour la découverte des requêtes fréquentes, exploitant la structure de l'espace de recherche défini par la généralisation syntaxique. Cet algorithme, montré à la figure 2.3, prend en paramètres une base de données $\mathcal{D}$ sous la forme d'un programme logique, un seuil minimum de support s_{min}, ainsi qu'un langage d'hypothèses $\mathcal{L}$, appelé WRMODE, limitant le type de requêtes pouvant être explorées. Étant donné un atome K servant de *clé*, WARMR retourne l'ensemble $\mathcal{F}$ des requêtes fréquentes. Tout comme l'algorithme APRIORI, WARMR alterne entre une phase de génération de candidats à l'aide d'un opérateur de spécialisation, et une phase d'évaluation des candidats selon leur fréquence. L'algorithme commence avec des ensembles de requêtes fréquentes $\mathcal{F}$ et infréquentes $\mathcal{I}$ vides, et un ensemble de candidats ne contenant que la clé. À chaque niveau k, WARMR obtient ensuite les candidats fréquents $\mathcal{F}_k$, en comptant, pour chaque candidat, le nombre de substitutions vérifiant la clé qui vérifient également le candidat. Les candidats fréquents sont alors ajoutés à $\mathcal{F}$ et les infréquents à $\mathcal{I}$. L'ensemble de candidats du niveau suivant $\mathcal{C}_{k+1}$ est ensuite obtenu à l'aide de la fonction WARMR-gen. Pour chaque requête $Q \in \mathcal{F}_k$, on ajoute à $\mathcal{C}_{k+1}$ la requête Q' spécialisant Q, i.e. obtenue en lui ajoutant un atome, sauf si :

1. Q' est plus spécifique qu'une requête infréquente de $\mathcal{I}$, ou
2. Q' est équivalente à une requête fréquente de $\mathcal{F}$.

Comme pour l'algorithme APRIORI, WARMR se termine lorsqu'il n'y a plus de requêtes candidates, i.e. $\mathcal{C}_k = \emptyset$. Cependant, contrairement à l'algorithme APRIORI, WARMR ne vérifie pas directement la fermeture descendante d'une requête générée Q', car les requêtes généralisant Q' ne sont pas forcément valides selon le langage d'hypothèses. Au lieu de cela, WARMR conserve la liste des requêtes infréquentes, et vérifie que Q' ne généralise aucune de ces requêtes.

Comme vérifier de la θ-subsomption est un problème *NP*-complet, l'algorithme WARMR est limité par la taille de la base de données et par le seuil minimum de support. Dans

Algorithme WARMR

Entrées: Un base de données $\mathcal{D}$, un langage d'hypothèses $\mathcal{L}$, une clé K, et un seuil minimum de support s_{min}.

Sorties: Les requêtes fréquentes $\mathcal{F}$.

$\mathcal{C}_1 := \{? - K\}$;
$\mathcal{F} := \mathcal{I} := \emptyset$;
$k := 1$;
tant que $\mathcal{C}_k \neq \emptyset$ **faire**
  $\mathcal{F}_k := \{Q \in \mathcal{C}_k \text{ t.q. } |reponse(K, \mathcal{D}) \cap reponse(Q, \mathcal{D})| \geq s_{min}\}$;
  $\mathcal{F} := \mathcal{F} \cup \mathcal{F}_k$;
  $\mathcal{I} := \mathcal{I} \cup (\mathcal{C}_k \setminus \mathcal{F}_k)$;
  $\mathcal{C}_{k+1} :=$ `WARMR-gen` $(\mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{F}_k)$;
  $k := i + 1$;
retourner $\mathcal{F}$;

Procedure `WARMR-gen` $(\mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{F}_k)$

Entrées: Le langage d'hypothèses $\mathcal{L}$, les requêtes fréquentes $\mathcal{F}$ et infréquentes $\mathcal{I}$, et les requêtes fréquentes du niveau courant $\mathcal{F}_k$.

Sorties: Les requêtes candidates $\mathcal{C}_{k+1}$ du niveau suivant.

$\mathcal{C}_{k+1} := \emptyset$;
pour chaque *requête* $Q \in \mathcal{F}_k$ **faire**
  pour chaque *requête* $Q' \in \rho_s(Q)$ **faire**
    si $\forall Q_I \in \mathcal{I},\ Q' \preceq Q_I$ *et* $\forall Q_F \in \mathcal{F},\ Q'$ *n'est pas équivalent à* Q_F **alors**
      $\mathcal{C}_{k+1} := \mathcal{C}_{k+1} \cup \{Q'\}$;
retourner $\mathcal{C}_{k+1}$;

Figure 2.3 L'algorithme WARMR pour la découverte des requêtes fréquentes, et sa procédure WARMR-gen.

le but de réduire la complexité de cette tâche, Nijssen et Kok ont proposé un algorithme appelé FARMER (Nijssen et Kok, 2001), qui évite la θ-subsomption à l'aide de structures de données avancées, au coût de générer des requêtes équivalentes. Nijssen et Kok ont illustré, à l'aide d'expériences, l'avantage de leur algorithme sur plusieurs jeux de tests, et l'équivalence des requêtes obtenues avec leur algorithme et WARMR pour un type particulier de biais déclaratif.

2.3 La découverte des sous-graphes fréquents

Une autre application de la découverte des concepts intéressants, basée sur la fréquence, est la découverte des sous-graphes fréquents. Comme nous l'avons vu dans l'introduction, ce problème joue un rôle très important dans plusieurs domaines, tels que la bioinformatique et la chimie computationnelle. Cette section donne une vue d'ensemble des méthodes pour résoudre ce problème.

De façon générale, les méthodes pour trouver les sous-graphes fréquents d'un ensemble de graphes peuvent être catégorisées selon trois éléments : 1) l'espace de recherche, 2) la stratégie d'exploration de cet espace, et 3) la façon dont ces méthodes gèrent les problèmes d'isomorphisme de graphe et de sous-graphe. Comme vu précédemment, l'espace de recherche se définit à l'aide d'un opérateur de spécialisation ρ_s. Dans le cas de la découverte des sous-graphes fréquents, l'ensemble des graphes spécialisant un graphe G selon ρ_s contient, de façon générale, tous les graphes ayant un sous-graphe isomorphe à G, i.e. $\rho_s(G) = \{G' \mid G \subset G'\}$. Il est coutume d'appeler ces graphes les *extensions* ou les *prolongements* de G. La figure 2.4 montre l'espace de recherche contenant tous les sous-graphes connexes d'au plus trois sommets dont les étiquettes peuvent être X ou Y. On remarque qu'une extension d'un graphe G s'obtient en ajoutant une nouvelle arête reliant deux sommets de G, ou un nouveau sommet et une arête le reliant à un sommet de G.

L'espace de recherche dépend également du type de graphes recherchés. Ainsi, on peut ne s'intéresser qu'à trouver les chaînes fréquentes, comme les algorithmes MOLFEA développé par Kramer et al. (Kramer et al., 2001) et PREFIXSPAN proposé par Pei et al. (Pei et al., 2001), ou les arbres fréquents, tels que les algorithmes TREEMINER de Zaki (Zaki, 2002), FREQT de Asai et al. (Asai et al., 2002), et CHOPPER développé par Wang et al. (Wang et al., 2004a). Ces deux problèmes sont beaucoup moins complexes que la tâche plus générale de trouver les sous-graphes fréquents pouvant contenir des cycles, du fait qu'il existe des techniques efficaces pour les problèmes d'isomorphismes de graphe et de sous-graphe pour ce type de données. Dans le cas général autorisant les cycles, on s'intéresse typiquement à n'obtenir que des sous-graphes connexes. Cependant, on peut restreindre la recherche à certains types de sous-graphes. Par exemple, on peut se limiter aux sous-graphes induits, comme le font les algorithmes AGM d'Inokuchi et al. (Inokuchi et al., 2000) et ADI-MINE de Wang et al. (Wang et al., 2004b). De même, on peut ne s'intéresser qu'aux sous-graphes fréquents maximaux, comme les algorithmes SPIN de Huan et al. (Huan et al., 2004b), CMTREEMINER de Xia et Yang (Xia et Yang, 2005), et MARGIN de Thomas et al. Puisque tout sous-graphe d'un graphe fréquent maximal est lui-même fréquent, les graphes fréquents maximaux offrent une représentation compacte de la solution. Par ailleurs, on peut chercher les sous-graphes qui sont fréquents selon une mesure différente du support, comme le nombre d'isomorphismes de sous-graphe d'un petit graphe à un plus gros. Des exemples d'algorithmes pour ce problème sont SUBDUE de Cook et al. (Cook et Holder, 1994), DRYADE de Termier et al. (Termier et al., 2004), et SIGRAM développé par Kuramochi et Karypis (Kuramochi et Karypis, 2005). Notons, enfin, que des systèmes permettant d'imposer dynamiquement des contraintes sur le type de sous-graphes recherchés ont été proposés récemment, e.g. (Wang et al., 2005a; Zhu et al., 2007).

La stratégie d'exploration de l'espace de recherche est sûrement l'élément le plus important d'un algorithme pour la découverte des sous-graphes fréquents. Tel que mentionné

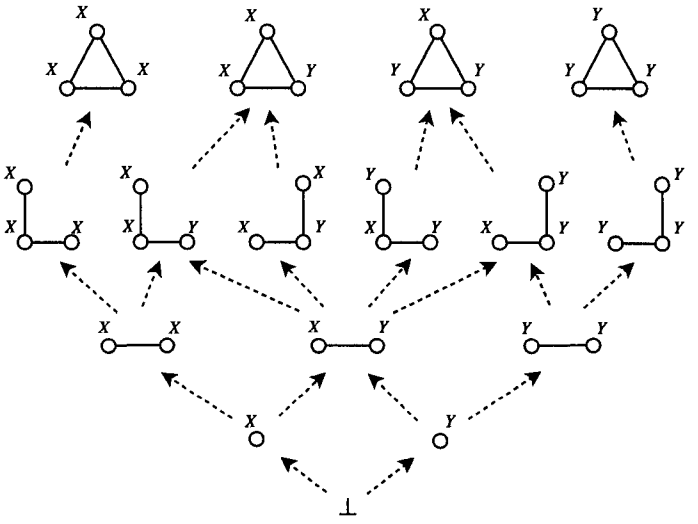


Figure 2.4 L'espace de recherche contenant tous les sous-graphes connexes d'au plus trois sommets et deux étiquettes de sommet.

précédemment, on peut séparer les stratégies d'exploration en trois classes : les stratégies de recherche en largeur, les stratégies de recherche en profondeur, et les stratégies heuristiques. Contrairement aux deux premières classes, les stratégies heuristiques se distinguent du fait qu'elles sacrifient l'exploration complète de l'espace de recherche au profit de la rapidité. Ainsi, les stratégies heuristiques tentent d'obtenir seulement les concepts les plus intéressants. Une telle approche heuristique est l'algorithme CLIP (Yoshida et Motoda, 1995). L'algorithme SUBDUE, qui sera vu plus en détail, est une autre approche heuristique qui utilise une mesure de la théorie de l'information, la *longueur de description minimale*, pour trouver des sous-graphes intéressants.

Les algorithmes utilisant la recherche en largeur, par ailleurs, explorent l'espace de recherche niveau par niveau, où chaque niveau renferme typiquement des sous-graphes ayant un sommet ou une arête de plus que les sous-graphes du niveau précédent. Comme le fait l'algorithme APRIORI pour les ensembles fréquents d'items, les sous-graphes fréquents sont obtenus en trois étapes : 1) on génère l'ensemble des sous-graphes can-

didats du niveau suivant en joignant des paires de sous-graphes fréquents du niveau courant, 2) on vérifie la fermeture descendante des graphes candidats, i.e. on vérifie que tous leur sous-graphes du niveau précédent sont fréquents, et 3) on calcule le support des candidats vérifiant la fermeture descendante, et on ne conserve que les candidats fréquents. Ces étapes sont répétées, jusqu'à ce qu'un niveau ne contienne aucun sous-graphe fréquent. L'algorithme AGM et l'algorithme FSG, proposé par Kuramochi et Karypis (Kuramochi et Karypis, 2001), sont des exemples d'algorithmes utilisant la recherche en largeur. Cependant, les algorithmes de ce type souffrent typiquement de trois problèmes. Premièrement, la jonction de deux sous-graphes fréquents pour générer les graphes candidats exige normalement de calculer le plus grand sous-graphe commun de ces deux graphes, un problème difficile. Ensuite, la génération des candidats produit un grand nombre de graphes isomorphes. Enfin, cette technique demande de stocker les sous-graphes fréquents à chacun des niveaux, ce qui peut demander une quantité importante de mémoire.

Le dernier type de stratégie d'exploration utilise la recherche en profondeur. Contrairement à la recherche en largeur, la recherche en profondeur travaille avec un seul sous-graphe à la fois, dont tous les prolongements sont explorés avant l'exploration d'un autre sous-graphe de la même taille. Cette stratégie de recherche a trois avantages par rapport à la recherche en largeur. Tout d'abord, elle nécessite beaucoup moins de mémoire que la recherche en largeur qui doit conserver tous les sous-graphes fréquents du niveau courant. Ensuite, elle permet d'élaguer implicitement un grand nombre de sous-graphes non-fréquents, selon l'idée suivante : comme tout prolongement d'un sous-graphe non-fréquent ne peut être fréquent, on peut éviter d'explorer les prolongements de sous-graphes non-fréquents sans risquer de manquer des fréquents. Enfin, ce type de recherche permet l'emploi de techniques incrémentales pour le calcul du support. Des exemples d'algorithmes utilisant une stratégie de recherche en profondeur sont les algorithmes GSPAN développé par Yan et Han (Yan et Han, 2002), FFSM de Huan et al. (Huan et al.,

2003), et GASTON proposé par Nijssen et Kok (Nijssen et Kok, 2004).

Dans les prochaines sections, nous présentons plus en détail certains algorithmes caractérisant bien les méthodes de résolution pour la découverte des sous-graphes fréquents.

2.3.1 L'algorithme AGM

L'algorithme AGM (*Apriori Graph Mining*), proposé par Inokuchi et al. (Inokuchi et al., 2000), est un des premiers algorithmes développés pour la découverte de sous-graphes fréquents. Cet algorithme emploie une stratégie de recherche en largeur pour obtenir les sous-graphes fréquents dans la base de données, qui sont induits mais pas nécessairement connexes. Tout comme l'algorithme APRIORI, l'algorithme AGM, détaillé à la figure 2.5 procède niveau par niveau, en commençant avec les sous-graphes contenant un seul sommet, et génère à chacun des niveaux suivants un ensemble de sous-graphes candidats contenant un sommet de plus qu'au niveau précédent. Également comme l'algorithme APRIORI, AGM vérifie la fermeture descendante de ces candidats.

Génération des candidats

La génération des candidats dans AGM est semblable à celle de l'algorithme APRIORI pour les ensembles fréquents d'items. Dans le cas d'AGM, un sous-graphe candidat de k+1 sommets est créé à partir de deux sous-graphes fréquents de k sommets, qui ne diffèrent que d'un seul sommet et ses arêtes incidentes. Soit deux graphes dirigés de k sommets, G_k et H_k, dont les matrices d'adjacence sont :

$$A_k = \begin{pmatrix} A_{k-1} & a_1 \\ {a_2}^{\mathsf{T}} & a_{kk} \end{pmatrix} \quad \text{et} \quad B_k = \begin{pmatrix} B_{k-1} & b_1 \\ {b_2}^{\mathsf{T}} & b_{kk} \end{pmatrix}.$$

Algorithme AGM

Entrées: Un base de données graphiques $\mathcal{D}$ et un seuil minimum de support s_{min}.
Sorties: Les sous-graphes fréquents $\mathcal{F}$ dans $\mathcal{D}$.

$\mathcal{F} := \emptyset$;
$\mathcal{C}_1 :=$ les sommets étiquetés de $\mathcal{D}$;
$k := 1$;
tant que $\mathcal{C}_k \neq \emptyset$ **faire**
  pour chaque *graphe* $G \in \mathcal{C}_k$ **faire**
    si $|\{G' \in \mathcal{D} \mid G \subseteq G'\}| \geq s_{min}$ **alors** $\mathcal{F}_k := \mathcal{F}_k \cup \{G\}$;
  $\mathcal{F} := \mathcal{F} \cup \mathcal{F}_k$;
  $\mathcal{C}_{k+1} :=$ `AGM-gen` $(\mathcal{F}_k)$;
  $k := k + 1$;
retourner $\mathcal{F}$;

Procedure `AGM-gen` $(\mathcal{F}_k)$

Entrées: Les sous-graphes fréquents du niveau courant $\mathcal{F}_k$.
Sorties: Les sous-graphes candidats du niveau suivant $\mathcal{C}_{k+1}$.

$\mathcal{C}_{k+1} := \emptyset$;
pour chaque *paire de graphes* $G_k, H_k \in \mathcal{F}_k$ **faire**
  Soit $A_k = \begin{pmatrix} A_{k-1} & a_1 \\ {a_2}^{\mathsf{T}} & a_{kk} \end{pmatrix}$ et $B_k = \begin{pmatrix} B_{k-1} & b_1 \\ {b_2}^{\mathsf{T}} & b_{kk} \end{pmatrix}$,
    les matrices d'adjacences de G_k et H_k ;
  si $A_k \leq B_k$ *et* $A_{k-1} = B_{k-1}$ **alors**
    pour chaque *paire d'étiquettes* $l_1, l_2 \in L$ **faire**
      Créer le graphe G_{k+1} ayant pour matrice d'adjacences

$$M_{k+1} = \begin{pmatrix} A_{k-1} & a_1 & b_1 \\ {a_2}^{\mathsf{T}} & a_{kk} & l_1 \\ {b_2}^{\mathsf{T}} & l_2 & b_{kk} \end{pmatrix} ;$$

      ajoute := *vrai* ;
      pour *chaque sommet* v *de* G_{k+1} **faire**
        Soit S_k le graphe obtenu en retirant v de G_{k+1} et ses arêtes incidentes;
        si $\nexists\, H_k \in \mathcal{F}_k$ *t.q.* $\mathrm{canon}(G_k) = \mathrm{canon}(H_k)$ **alors** ajoute := *faux* ;
      si ajoute = *vrai* **alors** $\mathcal{C}_{k+1} := \mathcal{C}_{k+1} \cup \{G_{k+1}\}$;
retourner $\mathcal{C}_{k+1}$;

Figure 2.5 L'algorithme AGM pour la découverte des sous-graphes fréquents, et sa procédure AGM-gen pour la génération des sous-graphes candidats.

Si les matrices d'adjacence A_k et B_k sont identiques à l'exception des éléments de la k-ème rangée et colonne, i.e. $A_{k-1} = B_{k-1}$, elles sont jointes pour former la matrice d'adjacence suivante :

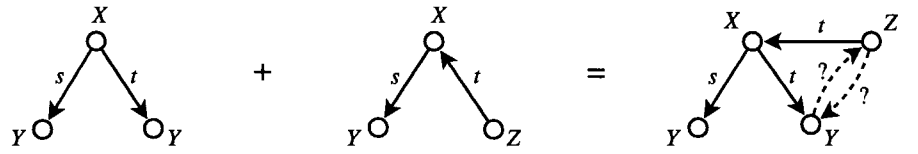
$$M_{k+1} = \begin{pmatrix} A_{k-1} & a_1 & b_1 \\ {a_2}^\mathsf{T} & a_{kk} & l_1 \\ {b_2}^\mathsf{T} & l_2 & b_{kk} \end{pmatrix},$$

où les éléments l_1 et l_2 sont indéterminés. En somme, M_{k+1} correspond à un graphe de k+1 sommets contenant G_k et H_k, et dont les deux sommets qui diffèrent dans ces deux graphes, sont reliées ou non par une arête d'étiquette quelconque. Soit $L(G_k)$ et $L(H_k)$ les ensembles contenant les étiquettes des arêtes de G_k et H_k, il y a $|L(G_k) \cup L(H_k)| + 1$ valeurs possibles pour les éléments l_1 et l_2 (la valeur supplémentaire est pour le cas où les deux sommets ne sont pas reliés). Par conséquent, l'union de G_k et H_k entraînera la création de $(|L(G_k) \cup L(H_k)| + 1)^2$ sous-graphes candidats différents. Par ailleurs, si G_k et H_k sont des graphes non-dirigés, l'union de leur matrices d'adjacence donnera la matrice symétrique suivante :

$$M_{k+1} = \begin{pmatrix} A_{k-1} & a & b \\ a^\mathsf{T} & a_{kk} & l \\ b^\mathsf{T} & l & b_{kk} \end{pmatrix}.$$

Comme il n'y a plus qu'un seul élément indéterminé l, le nombre de sous-graphes candidats générés, dans ce cas, ne sera que $|L(G_k) \cup L(H_k)| + 1$. La figure 2.6 montre la génération d'un sous-graphe candidat à 4 sommets, à partir de deux sous-graphes dirigés de 3 sommets, ayant X, Y, Z comme étiquettes de sommets, et s, t comme étiquettes d'arêtes. Les lignes pointillées signifient que l'arête correspondante peut être absente ou présente, et dans le dernier cas, que son étiquette est indéterminée. Selon la valeur que prennent ces arêtes, on peut ainsi créer 9 sous-graphes candidats différents.

On remarque, par ailleurs, que la matrice d'adjacence, obtenue lors de l'union de G_k et



(a)

	v_1	v_2	v_3
v_1	X	s	t
v_2		Y	
v_3			Y

+

	v'_1	v'_2	v'_3
v'_1	X	s	
v'_2		Y	
v'_3	t		Z

=

	u_1	u_2	u_3	u_4
u_1	X	s	t	
u_2		Y		
u_3			Y	?
u_4	t		?	Z

(b)

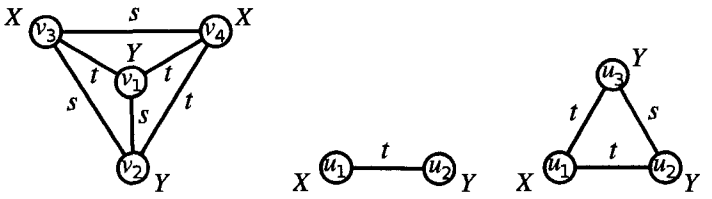
Figure 2.6 (*a*) La jonction de deux graphes dirigés de 3 sommets par l'algorithme AGM, et (*b*) la jonction au niveau des matrices d'adjacence.

H_k, dépend de l'ordre dans lequel on considère ces graphes, i.e. $A_k + B_k \neq B_k + A_k$. Cependant, les matrices obtenues selon un ordre ou son inverse représentent deux automorphismes du même graphe. Pour éviter d'engendrer deux fois le même sous-graphe candidat, AGM impose ainsi que G_k et H_k soient joints seulement si la chaîne obtenue en concaténant les éléments de A_k est lexicographiquement inférieure à celle obtenue pour B_k, noté $A_k \leq B_k$. Enfin, pour vérifier qu'un sous-graphe S_k du graphe G_{k+1} est bel et bien fréquent, on calcule son code canonique, noté $\mathrm{canon}(S_k)$, en trouvant la permutation des sommets pour laquelle la chaîne obtenue de la matrice d'adjacences permutée est minimale. On vérifie ensuite qu'il existe un sous-graphe fréquent de $\mathcal{F}_k$ ayant le même code canonique.

Calcul du support

Après avoir généré les sous-graphes candidats, on doit déterminer lesquels de ces candidats sont fréquents, en calculant leur support dans la base de données. La stratégie utilisée par AGM pour accélérer le test d'isomorphisme de sous-graphe consiste à réutiliser certains des calculs faits aux étapes précédentes. Considérons, par exemple, la tâche

de déterminer si le graphe G_2 de la figure 2.7(b) est isomorphe à un sous-graphe du graphe G_1 montré en (a). L'arbre de recherche associée à cette tâche, où chaque niveau représente l'association d'une paire de sommets, est présenté à la figure 2.8. En suivant la numérotation, le premier isomorphisme que l'on trouve associe u_1 à v_3 et u_2 à v_1. Supposons maintenant que l'on veuille déterminer si le graphe G_3, montré en (c), est isomorphe à un sous-graphe de G_1. Comme $G_2 \subset G_3$, on peut réutiliser le travail fait pour G_2. Ainsi, puisque la portion de l'arbre de recherche située à gauche du chemin v_3-v_1 ne mène à aucun isomorphisme pour G_2, elle ne mènera à aucune solution pour G_3. On peut alors débuter la recherche d'un isomorphisme pour G_3 à partir de ce chemin. Utilisant cette stratégie, on trouve un isomorphisme de G_3 à un sous-graphe G_1 qui associe u_1 à v_4, u_2 à v_1, et u_3 à v_2.



(a) G_1 (b) G_2 (c) G_3

Figure 2.7 Trois graphes étiquetés G_1, G_2 et G_3, tels que $G_2 \subset G_3 \subset G_1$.

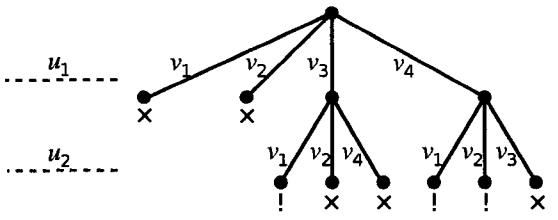


Figure 2.8 L'arbre de recherche correspondant la recherche d'un sous-graphe du graphe G_1 de la figure 2.7 isomorphe à G_2.

2.3.2 L'algorithme FSG

Développé par Kuramochi et Karypis (Kuramochi et Karypis, 2001), l'algorithme FSG (*Frequent Sub-Graph mining*), tout comme AGM, utilise la recherche en largeur et le principe *Apriori* pour trouver les sous-graphes fréquents de la base de données. Cependant, alors qu'AGM obtient des sous-graphes induits qui sont connexes ou non, FSG obtient les sous-graphes connexes fréquents pouvant être induits ou non. Au plus haut niveau, l'algorithme FSG est similaire à AGM. À partir d'un ensemble de sous-graphes candidats initial, FSG alterne entre une phase d'évaluation des candidats, où la base de données est traversée pour calculer le support des candidats, et une phase de génération de candidats. À la différence d'AGM qui spécialise les graphes en leur ajoutant un sommet, l'algorithme FSG génère à chaque niveau des sous-graphes candidats ayant une arête de plus.

Génération des candidats

Comme dans AGM, la génération des sous-graphes candidats dans FSG se fait en combinant deux sous-graphes fréquents partageant une structure commune. Soit G_k et H_k deux graphes de k arêtes, un *noyau* pour G_k et H_k est un sous-graphe N_{k-1} de k-1 arêtes qui est inclus dans ces deux graphes, i.e. $N_{k-1} \subset G_k$ et $N_{k-1} \subset H_k$. Un sous-graphe candidat G_{k+1} de k+1 arêtes est généré de deux graphes G_k et H_k si ces graphes partagent un noyau N_{k-1}. Tout comme AGM, la jonction de deux sous-graphes G_k et H_k dans FSG produit plusieurs sous-graphes candidats différents. La différence entre ces candidats peut se produire pour trois raisons, illustrées à la figure 2.9. Premièrement, la différence entre le noyau et les deux graphes peut être une paire de sommets u et v ayant la même étiquette dans les deux graphes, mais rattachés au noyau à des endroits différents. Ce cas, montré en (a), entraîne la création de deux candidats, selon si on considère les sommets u et v comme distincts ou non. Ensuite, le noyau peut avoir plusieurs automorphismes,

tel que montré en (b). Il existe, dans ce cas, plusieurs façons d'associer les sommets des sous-graphes correspondants au noyau de G_k et H_k, chacune donnant un candidat différent. Dans le pire cas, si le noyau est une clique ayant n sommets de même étiquette, le nombre de candidats différents générés sera $n!$. Finalement, il peut y avoir plus d'un noyau commun entre G_k et H_k, tel que montré en (c). Dans ce cas, on doit traiter chaque noyau séparément. Comme chaque noyau a une arête de moins que G_k et H_k, il y aura au plus k-1 noyaux partagés par ces deux graphes.

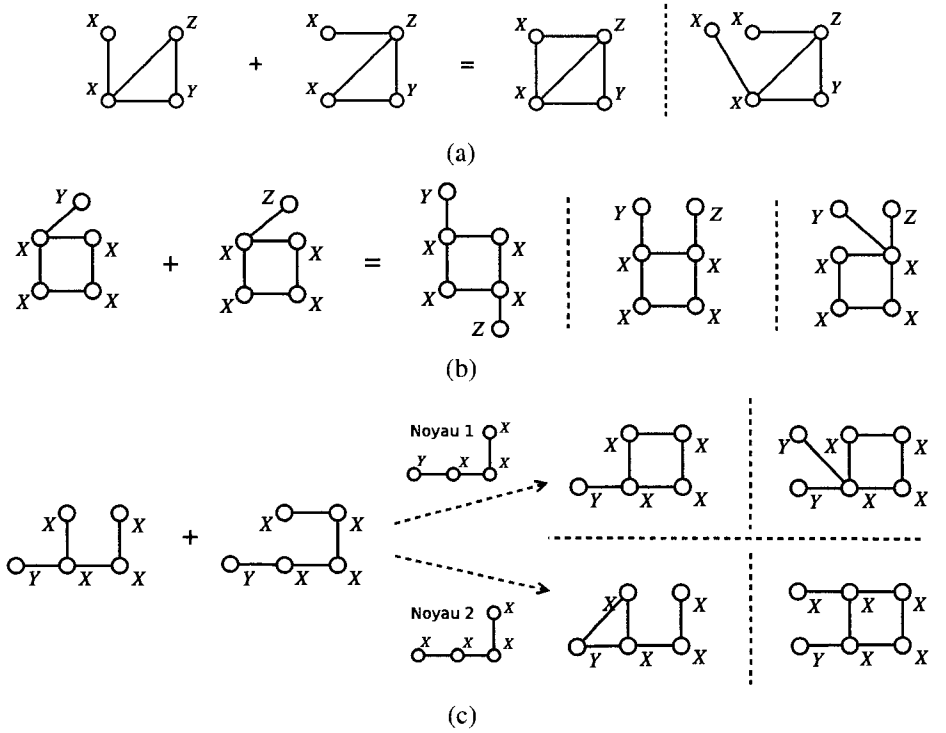


Figure 2.9 Trois raisons expliquant la création de candidats différents dans FSG : (a) les graphes ne diffèrent que par une paire de sommets de même étiquette, rattachés au noyau à des endroits différents, (b) le noyau possède plusieurs automorphismes, et (c) il existe plusieurs noyaux communs aux deux graphes.

La figure 2.10 montre, avec plus de détails, la procédure de l'algorithme FSG pour la génération des sous-graphes candidats de k+1 arêtes, à partir des sous-graphes fréquents de k arêtes. Pour chaque paire de sous-graphes fréquents G_k et H_k telle que $\mathrm{code}(\mathrm{G_k}) \leq$

code(H_k), où le code est une fois de plus obtenu en concaténant les éléments de la matrice d'adjacences, et pour chaque noyau N_{k-1} commun à G_k et H_k, la procédure génère l'ensemble des automorphismes de N_{k-1}. Ensuite, pour chacun de ces automorphismes, un candidat G_{k+1} ayant k+1 arêtes est généré tel qu'expliqué précédemment. La procédure vérifie ensuite que G_{k+1} n'est pas isomorphe à un candidat déjà généré en calculant son code canonique, i.e. le code minimal pour toute permutation des sommets, et en le comparant à celui des candidats présents dans $\mathcal{C}_{k+1}$. S'il s'agit bien d'un nouveau candidat, la procédure s'assure que G_{k+1} satisfait la fermeture descendante. Cette étape est faite en calculant le code canonique de chaque sous-graphe S_k, produit en retirant une arête de G_{k+1}, et en le comparant avec ceux des graphes fréquents de $\mathcal{F}_k$. Finalement, si G_{k+1} satisfait la fermeture descendante, il est ajouté à l'ensemble des candidats $\mathcal{C}_{k+1}$.

Par ailleurs, l'algorithme FSG emploie une technique particulière pour identifier les noyaux communs aux graphes G_k et H_k. Cette technique exploite le fait qu'on peut conserver le code canonique de chacun des k sous-graphes de G_k et H_k créés en retirant une arête. On peut alors comparer un à un les codes canoniques des sous-graphes de G_k et H_k, et conserver les paires de représentations identiques. Ces paires de représentations identiques correspondent aux noyaux. En supposant que les codes canoniques des sous-graphes de G_k et H_k ont déjà été calculés à une étape précédente, cette technique nécessite en tout k^2 comparaisons.

Calcul du support

La stratégie employée par FSG pour le calcul du support consiste à conserver, pour chaque sous-graphe fréquent $G_k \in \mathcal{F}_k$, l'ensemble des graphes de $\mathcal{D}$ le supportant, i.e. $\{G' \in \mathcal{D} \mid G_k \subseteq G'\}$. Lorsqu'on doit calculer le support d'un graphe G_{k+1}, on identifie d'abord l'intersection des ensembles de graphes supportant chacun des k+1 sous-graphes S_k de G_{k+1}, créés en lui retirant une arête. Puisque le nombre de graphes

Procedure `FSG-gen` ($\mathcal{F}_k$)

Entrées: Les sous-graphes fréquents du niveau courant $\mathcal{F}_k$.
Sorties: Les sous-graphes candidats du niveau suivant $\mathcal{C}_{k+1}$.

$\mathcal{C}_{k+1} := \emptyset$;
pour chaque *paire de graphes* $G_k, H_k \in \mathcal{F}_k$ **faire**
  si $\text{code}(G_k) \leq \text{code}(H_k)$ **alors**
    $\mathcal{N} := \{N_{k-1}$ de $k-1$ arêtes $\mid N_{k-1} \subset G_k$ et $N_{k-1} \subset H_k\}$;
    pour chaque *noyau* $N_{k-1} \in \mathcal{N}$ **faire**
      pour chaque *automorphisme* ϕ *de* N_{k-1} **faire**
        Créer un candidat G_{k+1} à partir de G_k, H_k et ϕ ;
        si $\nexists\, G'_{k+1} \in \mathcal{C}_{k+1}$ *t.q.* $\text{canon}(G_{k+1}) = \text{canon}(G'_{k+1})$ **alors**
          ajoute := *vrai* ;
          pour chaque *arête* e *de* G_{k+1} **faire**
            Soit S_k le graphe généré en retirant e de G_{k+1} ;
            si $\nexists\, H_k \in \mathcal{F}_k$ *t.q.* $\text{canon}(S_k) = \text{canon}(H_k)$ **alors**
              ajoute := *faux* ;
          si ajoute = *vrai* **alors** $\mathcal{C}_{k+1} := \mathcal{C}_{k+1} \cup \{G_{k+1}\}$;
retourner $\mathcal{C}_{k+1}$;

Figure 2.10 La procédure de génération de sous-graphes candidats de l'algorithme FSG.

dans l'intersection est une borne supérieure sur le support de G_{k+1}, on peut éviter de calculer le support de G_{k+1} si ce nombre est en dessous du seuil minimum de support s_{min}.

2.3.3 L'algorithme GSPAN

Les algorithmes AGM et FSG, basés sur une recherche en largeur utilisant le principe *Apriori*, souffrent d'importants problèmes liés à cette approche. Ainsi, comme ces algorithmes doivent générer et maintenir un très grand nombre de candidats, pour la plupart infréquents, l'espace nécessaire pour conserver ces candidats, et l'effort requis pour tester si ces candidats sont fréquents limitent la taille des bases de données pouvant être traitées. L'algorithme GSPAN (*Graph-based Substructure PAtterN mining*), proposé par Yan et Han (Yan et Han, 2002), élimine la génération de candidats à l'aide d'une recherche en profondeur d'un espace hiérarchique, où chaque noeud correspond au code d'un graphe. En associant à chaque graphe un code sous la forme d'une séquence dont la nature sera précisée plus loin, la tâche de trouver les sous-graphes fréquents d'une base de données équivaut alors à une exploration dans l'espace des séquences. De plus, si cette exploration est faite selon l'ordre lexicographique des séquences, on peut élaguer tous les noeuds qui n'ont pas un code canonique, puisque le noeud de code canonique correspond à un graphe isomorphe déjà exploré.

La figure 2.11 illustre l'approche utilisée par GSPAN. Cette figure montre l'espace hiérarchique formé d'un arbre où chaque noeud correspond au code d'un graphe, et chaque branche au prolongement, avec une arête, du graphe correspondant. Ainsi, chaque niveau de cet arbre contient le code de graphes ayant une arête de plus qu'au précédent, et la racine '$\perp$' correspond au code d'un graphe sans sommet ni arête. Cet arbre contient les

codes de deux graphes isomorphes G_1 et G_2, tels que

$$\text{canon}(G_1) = \text{code}(G_1) < \text{code}(G_2).$$

Comme le code de G_1 est inférieur à celui G_2, le noeud correspondant à G_1 sera nécessairement exploré avant celui de G_2. De plus, puisque G_2 n'a pas un code canonique, le noeud correspondant à ce graphe sera élagué durant la recherche.

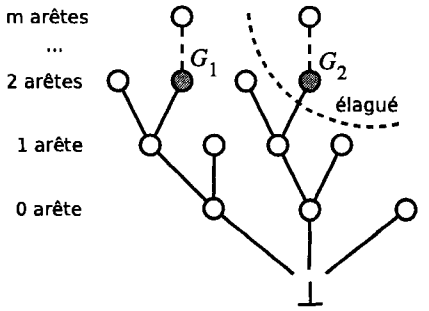


Figure 2.11 Un exemple d'un espace hiérarchique et d'élagage de noeuds non canoniques.

L'algorithme GSPAN est montré plus en détails à la figure 2.12. Cet algorithme, qui reçoit en paramètres une base de données $\mathcal{D}$ ainsi qu'un seuil minimum de support s_{min}, et retourne l'ensemble $\mathcal{F}$ des sous-graphes fréquents dans $\mathcal{D}$, commence par identifier les ensembles d'arêtes fréquentes $E_{\mathcal{F}}$ et infréquentes $E_{\mathcal{I}}$ dans $\mathcal{D}$. Selon le principe *Apriori*, un graphe contenant une arête infréquente est nécessairement infréquent. Ainsi, GSPAN peut retirer les arêtes infréquentes $E_{\mathcal{I}}$ de tous les graphes de $\mathcal{D}$, sans risque d'éliminer un sous-graphe fréquent. Ensuite, suivant un ordre croissant de code, chaque arête $e \in E_{\mathcal{F}}$ est explorée avec la procédure *explore*. Cette procédure retourne l'ensemble des sous-graphes fréquents qui sont les *descendants* de ce noeud, qui sont situés dans la branche de l'arbre de recherche commençant à ce noeud. Ces sous-graphes fréquents sont alors ajoutés à $\mathcal{F}$. Puisque tous les sous-graphes fréquents contenant l'arête e ont été trouvés, GSPAN peut retirer cette arête de tous les graphes de $\mathcal{D}$. L'algorithme termine

en retournant $\mathcal{F}$.

La procédure récursive *explore*, qui est responsable de l'exploration en profondeur de l'arbre de recherche, reçoit en paramètres $\mathcal{D}$, s_{min}, ainsi qu'un graphe G_k de k arêtes, et retourne les sous-graphes fréquents $\mathcal{F}_k$ descendants de G_k. Cette procédure vérifie tout d'abord si le code de G_k est canonique. Si ce n'est pas le cas, le graphe isomorphe à G_k de code canonique a déjà été exploré, et il est inutile d'explorer G_k. Par contre, si le code de G_k est bien canonique, la procédure ajoute ce graphe à $\mathcal{F}_k$, puis génère l'ensemble $\mathcal{C}_{k+1}$ des sous-graphes obtenus en prolongeant d'une arête G_k avec la procédure *prolonge*. En suivant l'ordre lexicographique croissant de leur code, le support de chaque sous-graphe $G_{k+1} \in \mathcal{C}_{k+1}$ dans $\mathcal{D}$ est calculé. Si ce support est au moins s_{min}, le noeud de ce graphe est exploré avec un appel récursif à la procédure *explore*, et les sous-graphes fréquents retournés par cette procédure sont ajoutés à $\mathcal{F}_k$. Sinon, le noeud est élagué.

2.3.4 Le code DFS

La caractéristique la plus importante de l'algorithme GSPAN est la manière dont les graphes sont codés. Ce code repose sur le concept d'arbre de recherche en profondeur, appelé *arbre DFS* :

Définition 4 (Arbre DFS). *Soit $G = (V, E, l_v, l_e)$ un graphe étiqueté simple ayant un ordre sur ses sommets, i.e. $v_i < v_j$, $1 \leq i < j \leq |V|$, un* ***arbre DFS*** *pour* ***Depth-First Search****, est un arbre produit par l'exploration en profondeur de G en suivant l'ordre des sommets. Soit T un arbre DFS pour G, on nomme* ***racine*** *de T le premier sommet exploré, i.e. le sommet v_1, et* ***sommet extrêmal*** *le sommet exploré en dernier, i.e. le sommet $v_{|V|}$. De plus, on appelle* ***chemin extrêmal*** *la chaîne de sommets et d'arêtes allant de la racine au sommet extrêmal.*

La figure 2.13 montre un graphe étiqueté (*a*), ainsi que trois arbres DFS, (*b*), (*c*) et

Algorithme GSPAN

Entrées: Un base de données graphiques $\mathcal{D}$ et un seuil minimum de support s_{min}.
Sorties: Les sous-graphes fréquents $\mathcal{F}$ dans $\mathcal{D}$.

Soit $E_{\mathcal{F}}$ et $E_{\mathcal{I}}$ l'ensemble des arêtes fréquentes et infréquentes dans $\mathcal{D}$;

pour chaque *graphe* $G \in \mathcal{D}$ **faire**
    $E(G) := E(G) \setminus E_{\mathcal{I}}$;
$\mathcal{F} := \emptyset$;
pour chaque *arête* $e \in E_{\mathcal{F}}$, *en ordre croissant de code* **faire**
    Soit G_1 le graphe contenant que l'arête e ;
    $\mathcal{F} := \mathcal{F} \cup$ `explore` $(\mathcal{D}, s_{min}, G_1)$;
    pour chaque *graphe* $G \in \mathcal{D}$ **faire**
        $E(G) := E(G) \setminus \{e\}$;
retourner $\mathcal{F}$;

Procedure `explore` $(\mathcal{D}, s_{min}, G)$

Entrées: Une base de données graphiques $\mathcal{D}$, un seuil minimum de support s_{min}, et un graphe G_k
Sorties: Les sous-graphes fréquents $\mathcal{F}_k$ descendants de G_k

$\mathcal{F}_k := \emptyset$;
si $\mathrm{code}(G_k) = \mathrm{canon}(\mathrm{G_k})$ **alors**
    $\mathcal{F}_k := \mathcal{F}_k \cup \{G_k\}$;
    $\mathcal{C}_{k+1} :=$ `prolonge` $(\mathcal{D}, G_k)$;
    pour chaque *enfant* $G_{k+1} \in \mathcal{C}_{k+1}$, *selon l'ordre croissant de leur code* **faire**
        si $|\{G \in \mathcal{D} \mid G_{k+1} \subseteq G\}| \geq s_{min}$ **alors**
            $\mathcal{F}_k := \mathcal{F}_k \cup$ `explore` $(\mathcal{D}, s_{min}, G_{k+1})$;
retourner $\mathcal{F}_k$;

Figure 2.12 L'algorithme GSPAN pour trouver les sous-graphes fréquents d'un ensemble de graphes étiquetés

(d), produit par une recherche en profondeur selon un ordre différent des sommets de (a). Les lignes pleines en (b), (c) et (d) correspondent aux arêtes des arbres DFS. On constate qu'un graphe peut avoir un nombre exponentiel d'arbres DFS. Par ailleurs, un arbre DFS permet de définir un ordre sur les arêtes d'un graphe.

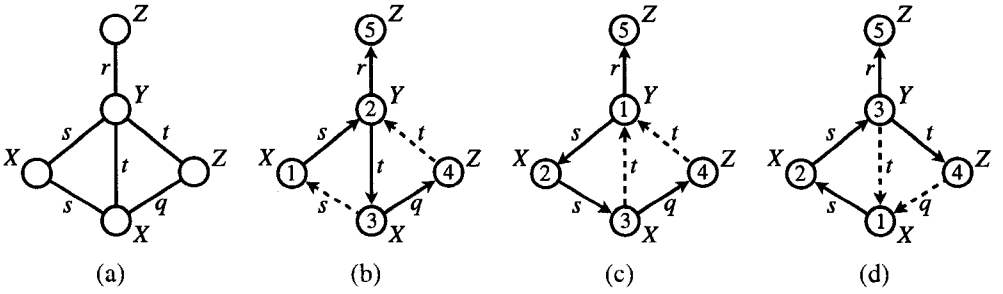


Figure 2.13 (a) Un graphe étiqueté et (b)-(d) plusieurs arbres DFS. Les arêtes arrière sont en pointillé.

Définition 5 (Ordre sur les arêtes). *Soit $G = (V, E, l_v, l_e)$ un graphe étiqueté simple et T un arbre DFS pour G, l'ensemble des* ***arêtes avant****, noté $E_{f,T}$, contient les arêtes de G qui sont dans T, i.e. $E \cap T$. De même, l'ensemble des* ***arêtes arrière****, noté $E_{b,T}$, contient les arêtes de G qui ne sont pas dans T, i.e. $E \setminus T$. Suivant la définition de T, on a*

$$E_{f,T} = \{(v_i, v_j) \in E \mid i < j\}$$
$$E_{b,T} = \{(v_i, v_j) \in E \mid i > j\}$$

Dans les définitions qui suivent, on suppose que $e_1 = (v_{i_1}, v_{j_1})$ et $e_2 = (v_{i_2}, v_{j_2})$. On peut définir, à l'aide de T, un ordre linéaire $\prec_T$ sur les arêtes de E selon trois cas :

1. $e_1 \in E_{f,T}$ *et* $e_2 \in E_{f,T}$ *:*

$$e_1 \prec_T e_2 \Leftrightarrow j_1 < j_2,$$

i.e. e_1 est inférieure à e_2 ssi son sommet terminal est inférieur à celui de e_2.

2. $e_1 \in E_{b,T}$ *et* $e_2 \in E_{b,T}$ *:*

$$e_1 \prec_T e_2 \Leftrightarrow i_1 < i_2 \ \textit{ou} \ i_1 = i_2 \ \textit{et} \ j_1 < j_2.$$

i.e. e_1 *est inférieur à* e_2 *ssi son sommet initial est inférieur à celui de* e_2 *ou, en cas d'égalité, le sommet terminal de* e_1 *est inférieur à celui de* e_2*.*

3. $e_1 \in E_{f,T}$*, et* $e_2 \in E_{b,T}$ *:*

$$e_1 \prec_T e_2 \Leftrightarrow j_1 \leq i_2,$$

i.e. e_1 *est inférieur à* e_2 *ssi le sommet terminal de* e_1 *n'est pas supérieur au sommet initial de* e_2*.*

En somme, l'ordre des arêtes de G se détermine de la manière suivante. En commençant avec la racine v_1 de T comme sommet courant, on ajoute une arête avant du sommet courant vers le prochain sommet dans la numérotation. On ajoute ensuite toutes les arêtes arrière connectant ce nouveau sommet aux sommets déjà visités, et on répète avec le nouveau sommet comme sommet courant. Lorsqu'il n'y a plus de nouveau sommet, on fait un retour-arrière vers le sommet précédent de l'ordre et on essaie d'ajouter une nouvelle arête avant. Le processus se termine lorsqu'on a ajouté toutes les arêtes de G. Illustrons ceci à l'aide d'un exemple. Soit G un graphe étiqueté et T son arbre DFS, tels que montrés à la figure 2.13(b). Les arêtes *avant* de G selon T sont (v_1, v_2), (v_2, v_3), (v_2, v_5), (v_3, v_4), alors que les arêtes *arrière* sont (v_3, v_1) et (v_4, v_2). Selon la définition précédente, l'ordre des arêtes de G est :

$$(v_1, v_2) \prec_T (v_2, v_3) \prec_T (v_3, v_1) \prec_T (v_3, v_4) \prec_T (v_4, v_2) \prec_T (v_2, v_5).$$

De même, l'arbre de la figure 2.13(c) induit l'ordre suivant :

$$(v_1, v_2) \prec_T (v_2, v_3) \prec_T (v_3, v_1) \prec_T (v_3, v_4) \prec_T (v_4, v_1) \prec_T (v_1, v_5).$$

Par ailleurs, l'ordre des arêtes permet de définir un code pour un graphe étiqueté.

Définition 6 (Code DFS). *Soit $G = (V, E, l_v, l_e)$ un graphe et T un arbre DFS pour G, on définit un* ***code de recherche en profondeur*** *ou* ***code DFS*** *pour G et T, noté* $\mathrm{code}(G, T)$, *comme une séquence*

$$\mathrm{code}(e_1)\mathrm{code}(e_2)\ldots\mathrm{code}(e_{|E|-1})\mathrm{code}(e_{|E|}),$$

telle que $e_i \prec_T e_{i+1}$, et où le code d'une arête (v_i, v_j) est

$$(i, j, l_v(v_i), l_e(v_i, v_j), l_v(v_j)),$$

i.e. le quintuplet contenant les numéros des sommets initial et terminal, l'étiquette du sommet initial, l'étiquette de l'arête, et finalement, l'étiquette du sommet terminal. L'ordre linéaire $\prec_T$ permet la définition d'un ***ordre lexicographique*** *sur les codes. Soit deux codes α et β tels que*

$$\begin{aligned} \alpha = \mathrm{code}(G_\alpha, T_\alpha) &= a_1 a_2 \ldots a_{n-1} a_n \\ \beta = \mathrm{code}(G_\beta, T_\beta) &= b_1 b_2 \ldots b_{m-1} b_m, \end{aligned}$$

on a $\alpha < \beta$ ssi un des deux cas suivants est vrai :

1. $a_k = b_k, 1 \leq k \leq n$ *et* $n < m$ *;*
2. $\exists t, 1 \leq t \leq \min\{m, n\}$ *t.q.* $a_k = b_k, 1 \leq k < t$, *et* $a_t < b_t$.

Finalement, le ***code canonique*** *d'un graphe G est le plus petit code d'un graphe G, selon l'ordre lexicographique, pour tout arbre DFS de G.*

La figure 2.14 montre les codes du graphe de la figure 2.13(*a*), générés en utilisant les arbres DFS montrés en (*b*), (*c*) et (*d*). Selon l'ordre lexicographique qui vient d'être défini, le code (*d*) est plus petit que le code (*b*), qui est plus petit que le code (*c*). En fait,

Arête	code (b)	code (c)	code (d)
e_1	(1,2,X,s,Y)	(1,2,Y,s,X)	(1,2,X,s,X)
e_2	(2,3,Y,t,X)	(2,3,X,s,X)	(2,3,X,s,Y)
e_3	(3,1,X,s,X)	(3,1,X,t,Y)	(3,1,Y,t,X)
e_4	(3,4,X,q,Z)	(3,4,X,q,Z)	(3,4,Y,t,Z)
e_5	(4,2,Z,t,Y)	(4,1,Z,t,Y)	(4,1,Z,q,X)
e_6	(2,5,Y,r,Z)	(1,5,Y,r,Z)	(3,5,Y,r,Z)

Figure 2.14 Les codes provenant des arbres DFS montrés à figure 2.13.

on peut vérifier que le code (d) est le code canonique de ce graphe.

Lors de l'exploration de l'espace de recherche, un graphe est prolongé en lui ajoutant une nouvelle arête reliant deux sommets de ce graphe, ou bien un nouveau sommet ainsi qu'une arête le reliant à un sommet du graphe. Cependant, comme la séquence des arêtes ajoutées au graphe doit correspondre à un code DFS, une nouvelle arête ne peut pas être ajoutée n'importe où. Ainsi, les arêtes *avant* ne peuvent être ajoutées que sur le chemin extrêmal de l'arbre DFS. De même, les arêtes *arrière* ne peuvent relier que le sommet extrêmal à un autre sommet du chemin extrêmal. Considérons, par exemple, le graphe de la figure 2.15(a), dont les sommets du chemin extrêmal correspondent aux cercles pleins, et le sommet extrêmal est identifié par "s.e.". Les prolongements valides à l'aide d'une arête *arrière* partant du sommet extrêmal sont montrés en (b) et (c). De même, les figures (d) à (g) montrent les prolongements valides à l'aide d'une arête *avant* allant d'un sommet sur le chemin extrêmal vers un nouveau sommet.

2.3.5 Extensions de GSPAN

Le succès important qu'a connu l'algorithme GSPAN et sa stratégie de recherche basée sur le code DFS a encouragé le développement d'autres méthodes basées sur cette stratégie. Une de ces méthodes, l'algorithme CLOSEGRAPH proposé par Yan et Han (Yan et Han,

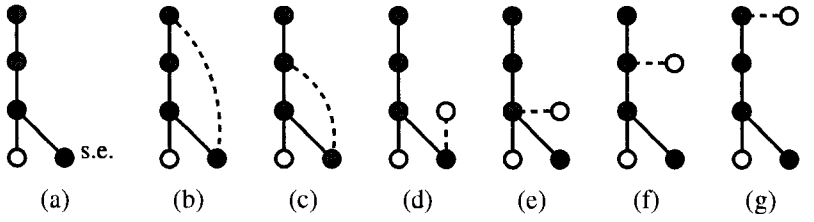


Figure 2.15 (*a*) Un graphe et (*b*)-(*g*) quelques uns des prolongements de ce graphe par GSPAN. Les sommets noircis forment le chemin extrêmal se terminant par le sommet extrêmal (s.e.). Les arêtes en pointillé représentent les prolongements du graphe.

2003), se limite à obtenir les sous-graphes connexes fréquents qui sont *fermés*. Dans le contexte de cet algorithme, un sous-graphe G est dit fermé s'il n'existe aucun super-graphe $G' \supset G$ tel que $\sup(G', \mathcal{D}) = \sup(G, \mathcal{D})$. L'idée est qu'un sous-graphe G qui n'est pas fermé est peu intéressant, puisqu'il existe un sous-graphe fermé $G' \supset G$ contenant chacune des instances de G dans la base de données. En pratique, seulement une fraction des sous-graphes sont fermés, ce qui permet de réduire la quantité de données retournées à l'usager par l'algorithme. De plus, se limiter aux sous-graphes fermés permet, dans certains cas, d'accélérer la recherche en évitant d'explorer des branches de l'espace de recherche ne menant à aucun sous-graphe fermé.

Un autre algorithme basé sur GSPAN est l'algorithme GASTON (*GrAph Sequence Tree extractiON*), proposé par Nijssen et Kok (Nijssen et Kok, 2004). Comme nous l'avons déjà mentionné, les problèmes d'isomorphisme de graphe et de sous-graphe sont moins complexes pour les graphes ne contenant pas de cycle, tels que les chaînes et les arbres, du fait qu'il existe des méthodes de résolution efficaces pour ce type de données. Or, dans bien des cas, la plupart des sous-graphes fréquents de la base de données sont justement des chaînes ou des arbres. GASTON s'inspire de ce principe pour accélérer l'exploration de l'espace de recherche. Ainsi, les chaînes fréquentes sont tout d'abord obtenues de manière très efficace. Ensuite, ces chaînes sont prolongées pour trouver les arbres fréquents. Enfin, les arbres fréquents sont à leur tour prolongés pour trouver les

graphes cycliques fréquents.

Un dernier algorithme inspiré de GSPAN est l'algorithme DSPM (*Diagonally Subgraphs Pattern Mining*), développé par Cohen et Gudes (Cohen et Gudes, 2004). Cet algorithme explore l'espace de recherche à l'aide d'une variante de la stratégie d'exploration de GSPAN, appelée *recherche à préfixe inverse*. Alors que la recherche à préfixe inverse utilise également le code DFS, cette stratégie diffère de celle employée par GSPAN par le fait que les prolongements d'un graphe sont explorés selon l'ordre inverse, i.e. décroissant, de leur code. Ceci garantit que tous les sous-graphes d'un graphe seront explorés avant ce graphe, ce qui permet de vérifier la fermeture descendante de ce graphe. Lors de l'exploration, les sous-graphes fréquents rencontrés sont conservés par DSPM. Ensuite, lorsqu'un graphe est exploré, cet algorithme calcule le code canonique de chacun de ses sous-graphes connexes, selon la technique employée par GSPAN, et vérifie qu'il existe, pour chacun de ces sous-graphes, un graphe fréquent ayant le même code canonique. En plus de cette stratégie de recherche, l'algorithme DSPM emploie également la technique de génération de candidats de la recherche par niveau. Selon les auteurs, cette stratégie hybride de recherche permet d'élaguer un nombre supérieur de sous-graphes non-fréquents.

2.3.6 L'algorithme SUBDUE

Les méthodes présentées jusqu'à maintenant utilisent la fréquence comme mesure pour déterminer si un sous-graphe est intéressant ou non. L'algorithme SUBDUE, présenté par Cook et Holder (Cook et Holder, 1994), emploie une mesure différente, basée sur l'idée que la meilleure théorie pour décrire un ensemble de données est la théorie minimisant le nombre de bits requis pour décrire ces données, un principe connu sous le nom de *longueur de description minimale* :

Définition 7 (Longueur de description minimale). *Soit G un graphe étiqueté, trouver un*

sous-graphe $G_{\min}$ de G tel que

$$G_{\min} = \underset{G' \subseteq G}{\operatorname{argmin}}\{I(G') + I(G|G')\},$$

où $I(G')$ est le nombre de bits requis pour encoder G', et $I(G|G')$ est le nombre de bits nécessaires pour encoder le graphe G dans lequel chaque instance de G' est remplacé par un sommet, ainsi que les liens permettant de reconstruire G à partir de ce nouveau sommet.

Une façon naïve de calculer le nombre de bits $I(G)$ pour encoder un graphe étiqueté $G = (V, E, L, l_{v,e})$ est la suivante. En supposant que chaque étiquette de L est représentée par un entier, il faut $O(|V| \log |L|)$ bits pour encoder les étiquettes des sommets. La matrice d'adjacence peut ensuite être utilisée pour encoder les arêtes. Si le graphe est dirigé, il faut $O(|V|^2 \log |L|)$ bits pour encoder les arêtes et leur étiquette, sinon il faut $O(\frac{|V|(|V|-1)}{2} \log |L|)$ bits pour encoder celles-ci. Alors que la longueur de description minimale présente des avantages sur la mesure de fréquence, cette mesure a le désavantage de ne pas être monotone ou anti-monotone selon la relation de spécialisation. Ainsi, plus $G_{\min}$ a de sommets, plus le nombre de sommets réduits dans G, pour chaque instance de $G_{\min}$ sera important, mais moins il sera fréquent donc moins d'instances seront remplacées dans G. Par ailleurs, puisque cette mesure n'est pas monotone, il n'existe pas de technique efficace pour élaguer l'espace de recherche, et on doit souvent avoir recours à une approche heuristique. Pour surmonter ce problème, SUBDUE explore l'espace de recherche à l'aide d'une stratégie de recherche heuristique *par faisceau*, un type de recherche arborescente dans laquelle on explore qu'un nombre réduit de branches à chaque niveau, donné par la *largeur du faisceau*. De même, SUBDUE utilise une méthode heuristique de résolution pour le problème d'isomorphisme de sous-graphe, permettant de trouver en temps polynomial un sous-graphe maximisant le critère de longueur de description minimale.

La figure 2.16 donne une description de haut niveau de l'algorithme SUBDUE. Au lieu d'avoir en paramètre une base de données renfermant plusieurs graphes, SUBDUE ne reçoit qu'un seul graphe étiqueté G ainsi qu'une largeur de faisceau M, et retourne un sous-graphe $G_{\min} \subseteq G$ qui minimise la longueur de description de G. En commençant avec un ensemble $\mathcal{C}_1$ contenant tous le sommets étiquetés de G, et à chaque niveau k, SUBDUE détermine les sous-graphes candidats $G_j \in \mathcal{C}_k$, $1 \leq j \leq M$ dont la valeur heuristique, donnée par

$$f(G_j) = I(G_j) + I(G|G_j),$$

est parmi les M plus faible. Pour chacun de ces sous-graphes G_j, SUBDUE vérifie ensuite si la valeur heuristique de G_j est inférieure à celle de $G_{\min}$. Si c'est le cas, G_j devient le meilleur candidat. Par la suite, SUBDUE ajoute à l'ensemble initialement vide des candidats du prochain niveau, $\mathcal{C}_{k+1}$, tous les sous-graphes engendrés par le prolongement d'une instance de G_j dans G. Le prolongement consiste à ajouter une nouvelle arête allant d'un sommet de G_j vers un autre sommet de G_j, ou vers un nouveau sommet de G. Après avoir généré tous les candidats du prochain niveau, SUBDUE partitionne ces candidats en classes à l'aide d'une méthode heuristique pour le problème d'isomorphisme de sous-graphe. Ainsi, les sous-graphes dont la similarité, obtenue par cette méthode, est au dessus d'un seuil de tolérance sont mis dans une même classe. Au niveau suivant, i.e. k+1, les sous-graphes d'une classe sont considérés comme des instances du même graphe. Le processus se termine lorsqu'il n'y a plus de nouveaux candidats.

L'algorithme SUBDUE permet également de biaiser la recherche afin d'obtenir un sous-graphe $G_{\min}$ ayant des caractéristiques particulières. Ainsi, on peut utiliser une fonction de coût modifiée

$$f_b(G') = f(G') \times \prod_{r=1}^{|\mathcal{R}|} \text{regle}_r(G')^{e_r},$$

où $\mathcal{R}$ est un ensemble de règles retournant une valeur supérieure ou égale à 1, et e_r est un exposant déterminant l'importance de la règle r. Ces règles permettent, entre autres,

Algorithme SUBDUE

Entrées: Un graphe étiqueté G et une largeur de faisceau M.
Sorties: Un sous-graphe $G_{\min}$ minimisant la longueur de description de G.

$f(G_{\min}) := \infty$;
$\mathcal{C}_1 :=$ les sommets étiquetés de G;
$k := 1$;

tant que $\mathcal{C}_k \neq \emptyset$ **faire**
  $\mathcal{C}_{k+1} := \emptyset$;
  Trier les graphes $G_j \in \mathcal{C}_k$ tel que $f(G_j) \leq f(G_{j+1})$;
  pour j *allant de* 1 *à* $\min\{M, |\mathcal{C}_k|\}$ **faire**
    Évaluer $f(G_j) = I(G_j) + I(G|G_j)$;
    si $f(G_j) < f(G_{\min})$ **alors** $G_{\min} := G_j$;
    pour chaque *instance* G' *de* G_j *dans* G **faire**
      $\mathcal{C}_{k+1} := \mathcal{C}_{k+1} \cup$ l'ensemble des prolongements de G' dans G ;
  Partitionner les sous-graphes de $\mathcal{C}_{k+1}$ en classes à l'aide d'une méthode heuristique ;
  $k := k + 1$;
retourner $G_{\min}$;

Figure 2.16 L'algorithme SUBDUE pour trouver un sous-graphe $G_{\min}$ minimisant la longueur de description d'un graphe G.

de biaiser la recherche vers la découverte d'une structure compacte, ou d'une structure isolée, i.e. dont les instances sont le plus disjointes possibles.

2.4 La génération de conjectures en théorie des graphes

Un des premiers programmes créés dans le but d'automatiser la génération de conjectures en mathématiques est le programme *Automated Mathematician* (AM), écrit par Douglas Lenat au milieu des années 70 (Lenat, 1979). Ce programme, qui génère des relations sur les nombres entiers, a permis, entre autres, de reproduire deux résultats célèbres : la conjecture de Goldbach voulant que tout entier supérieur à 2 puisse s'écrire comme la somme de deux nombre premiers, ainsi que le théorème fondamental de l'arithmétique voulant que tout entier supérieur à 1 s'écrive de manière unique comme le produit de nombres premiers. Un autre programme permettant de générer des concepts en mathématiques pures est le système HR (pour *Hardy-Ramanujan*), développé par l'équipe de Simon Colton à la fin des années 90 (Colton et al., 1999; Colton, 1999). Ce puissant programme, qui automatise également la démonstration des conjectures générées, a réussi l'exploit de générer plusieurs nouveaux théorèmes dans les domaines de la théorie des groupes et la théorie des nombres.

Dans le domaine de la théorie des graphes, un programme précurseur est le système GRAPH développé par Cvetkovic et al., au début des années 80 (Cvetkovic et al., 1981; Cvetkovic et Pevac, 1983a; Cvetkovic et Pevac, 1983b). Alors que GRAPH ne génère pas directement de conjectures, il permet de manipuler et visualiser interactivement des graphes, et d'obtenir la valeur de différents invariants pour ces graphes. Un autre système permettant de calculer des relations sur les invariants d'un graphe est le programme INGRID, développé par Brigham et ses collègues durant les années 80 (Brigham et Dutton, 1983). Ce programme calcule des bornes sur la valeur d'un invariant en dérivant des relations sur cet invariant à partir de relations déjà connues. Pour un sommaire des

résultats obtenus par INGRID, voir (Brigham et Dutton, 1985; Brigham et al., 1989; Brigham et Dutton, 1991). Le système *Graph Theorist*, développé par Epstein à la fin des années 80 (Epstein, 1988; Epstein et Sridharan, 1991), est un autre système permettant de générer de nouveaux concepts à partir d'une base de concepts spécifiques à la théorie des graphes. Ce système est parvenu à déduire des propriétés élémentaires de certaines classes de graphe, e.g. la propriété que tous les arbres sont acycliques. Bien qu'intéressants, ces systèmes n'ont pas permis, à eux seuls, de découvrir des résultats qui n'étaient pas connus en théorie des graphes. Les systèmes présentés dans les trois prochaines sections ont permis, en revanche, la découverte d'un grand nombre de nouvelles relations sur les invariants de graphe.

2.4.1 Le système GRAFFITI

Le système GRAFFITI, introduit par Fajtlowicz en 1986 (Fajtlowicz, 1987; Fajtlowicz, 1988a; Fajtlowicz, 1988b; Fajtlowicz, 1990; Fajtlowicz, 1995), a permis, à ce jour, de générer plus d'un millier de conjectures sous la forme de relations algébriques entre des invariants de graphe, dont une bonne portion est disponible publiquement dans un document appelé *Written-on-the-wall* (Fajtlowicz, 2008; Fajtlowicz et DeLaVina, 2008). Pour chacune des conjectures s'y trouvant, le document donne également l'état de cette conjecture : ouverte, réfutée ou démontrée.

GRAFFITI emploie deux bases de données. La première contient un ensemble de graphes particuliers, proposés par différents chercheurs ou ayant servi à réfuter une conjecture produite par le système, et la seconde un ensemble de conjectures générées par le système, qui n'ont pu être réfutées. Le processus de génération de conjectures de GRAFFITI comporte, en somme, six étapes :

1. L'usager choisit un invariant pour lequel il aimerait trouver des bornes.
2. Le système génère un grand nombre de conjectures sous la forme d'inégalités

dont le côté gauche est l'invariant choisi à la première étape, et le côté droit est une relation algébrique (linéaire ou non) sur d'autres invariants. Il évalue ensuite les deux côtés de chaque inégalité sur tous les graphes de sa base de données.

3. La conjecture est rejetée si elle n'est pas satisfaite pour un graphe de la base de données.
4. Sinon, le système évalue si la relation est intéressante : une relation est intéressante seulement si elle donne, pour au moins un graphe de la base de données, de strictement meilleures bornes que les relations déjà dans la base de relations.
5. Si la relation est jugée intéressante, elle est ajoutée à la base de relations, sinon elle est mise de côté jusqu'à ce qu'on ajoute à la base de données un nouveau graphe pour lequel cette relation est plus forte que toutes les autres.
6. Le système vérifie ensuite que, pour chaque graphe de la base de données, il existe une relation de sa base de relations portant sur le même invariant, et qui est satisfaite à égalité pour ce graphe. Si c'est le cas, les nouvelles relations générées par le système sont ajoutées à la liste des conjectures à démontrer. Sinon, le système retourne à l'étape 2.

2.4.2 Le système AUTOGRAPHIX

Un système plus récent mais également important est le système AUTOGRAPHIX (AGX), développé par Caporossi et Hansen (Caporossi et Hansen, 2000) à la toute fin des années 90. Alors qu'AGX génère également des conjectures ayant la forme de relations algébriques sur des invariants de graphe, le problème est abordé de façon différente : la génération d'une conjecture est transformée en problème d'optimisation dont la fonction objectif est une relation sur les invariants, et l'espace des solutions est une famille de graphes. Soit $\mathcal{C}$ une classe de graphes, $i_1(G), i_2(G), \ldots i_l(G)$ un ensemble d'invariants d'un graphe $G \in \mathcal{C}$, et $p_1, p_2, \ldots p_l$ les valeurs ciblées pour ces invariants. La

tâche de trouver un graphe ayant ces valeurs d'invariants correspond ainsi au problème de trouver

$$G^* \in \underset{G \in \mathcal{C}}{\operatorname{argmin}} f(G) = \sum_{k=1}^{l} |i_k(G) - p_k|.$$

De même, soit une conjecture $h(G) \leq g(G)$, où $h(G)$ et $g(G)$ sont des fonctions sur les invariants de G, et soit le problème suivant :

$$\min_{G \in \mathcal{C}} f(G) = g(G) - h(G).$$

Si on trouve un graphe G tel que $f(G) < 0$, la conjecture est réfutée. La même idée peut être utilisée pour suggérer des conjectures. Ainsi, soit $i_1(G)$ et $i_2(G)$ deux invariants de G, et soit le problème suivant :

$$\min_{G \in \mathcal{C}} f(G) = i_1(G) - i_2(G).$$

Si on ne trouve aucun graphe G tel que $f(G) < 0$, cela suggère que $i_1(G) \geq i_2(G)$ pour tout graphe dans $\mathcal{C}$. Par ailleurs, si tous les graphes minimisant f font partie d'une même classe, on peut utiliser cette information pour trouver une démonstration à la conjecture. AGX permet également d'imposer certaines contraintes sur les graphes recherchés, e.g. les graphes recherchés doivent faire partie d'une classe donnée, en incorporant ces contraintes dans la fonction objectif. Ainsi, supposons que l'on cherche un arbre minimisant un certain invariant $i(G)$, cette tâche peut être formulée comme le problème de trouver

$$G^* \in \underset{G}{\operatorname{argmin}}\ i(G) + M\left(|n - m - 1| + \max\{0, D(G) - m\}\right),$$

où n et m sont le nombre de sommets et d'arêtes de G, $D(G)$ représente le diamètre du graphe G, et M est une très grande constante. Ainsi, $\max\{0, D(G) - m\}$ impose que G soit connexe, car dans le cas contaire $D(G)$ est infini. De même, si G est connexe,

$|n - m - 1|$ interdit à G d'avoir un cycle. Par conséquent, ces deux contraintes font en sorte que tout graphe minimisant la fonction objectif est un arbre.

Pour résoudre ces problèmes d'optimisation, AGX emploie une métaheuristique à voisinage variable appelée *Variable Neighborhood Search* (VNS) (Mladenovic et Hansen, 1997). Cette métaheuristique emploie une méthode de descente à plusieurs voisinages qui sont considérés tour à tour jusqu'à ce qu'une solution améliorant la solution courante soit trouvée. Les voisinages utilisés pour les graphes proviennent de combinaisons de mouvements simples, tels qu'ajouter un sommet ou une arête à un graphe, ou bien échanger les arêtes de quatre sommets de ce graphe (mouvement *2-opt*). Pour sortir des optimums locaux, VNS perturbe la solution courante avant d'appliquer la méthode de descente, en lui ajoutant ou retirant un certain nombre d'arêtes. Cette perturbation est augmentée jusqu'à ce que la solution courante soit améliorée, ou on atteigne une limite fixée d'avance.

Alors que cette approche requiert à l'usager de fournir au système un problème à optimiser, AGX possède également un mode de génération de conjectures totalement automatisé, appelé *approche numérique* dans (Caporossi et Hansen, 2004). Soit un ensemble de m graphes pour lesquels on possède la valeur de n invariants. On note $X = [x_{ij}]_{m \times n}$ la matrice des observations dont chaque rangée renferme les valeurs d'invariants d'un graphe. Le but est de trouver une base contenant toutes les relations affines de la forme $a_1 x_{i1} + a_2 x_{i2} + \ldots + a_n x_{in} = b$, satisfaites pour $1 \leq i \leq m$. Soit μ_j la valeur moyenne des éléments de la j-ème colonne de X, i.e. $\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_{ij}$, notons $\bar{X}$ la matrice des observations centrées obtenue en enlevant aux valeurs d'une colonne de X la moyenne des valeurs de cette colonne, i.e. $\bar{x}_{ij} = x_{ij} - \mu_j$. Toute relation affine entre les colonnes de X est une relation linéaire entre les colonnes de $\bar{X}$: soit $x_{ij} = \sum_{\substack{k=1 \\ k \neq j}}^{n} c_k x_{ik} + d$ une

relation entre les colonnes de X alors

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{\substack{k=1\\k\neq j}}^{n} c_k x_{ik} + d\right) = \sum_{\substack{k=1\\k\neq j}}^{n} c_k \cdot \frac{1}{m}\sum_{i=1}^{m} x_{ik} + \frac{1}{m}\sum_{i=1}^{m} d = \sum_{\substack{k=1\\k\neq j}}^{n} c_k \mu_k + d,$$

et ainsi

$$\bar{x}_{ij} = x_{ij} - \mu_j = \sum_{\substack{k=1\\k\neq j}}^{n} c_k x_{ik} - c_k \mu_k = \sum_{\substack{k=1\\k\neq j}}^{n} c_k \left(x_{ik} - \mu_k\right) = \sum_{\substack{k=1\\k\neq j}}^{n} c_k \bar{x}_{ik}$$

Il existe donc une relation linéaire entre les variables centrées, ayant les mêmes coefficients. De plus, soit la matrice de covariance $V = [v_{ij}]_{n\times n} = \bar{X}^{\mathsf{T}}\bar{X}$, on a

$$v_{jk} = \sum_{i=1}^{m} \bar{x}_{ij}\bar{x}_{ik} = \sum_{i=1}^{m}\left(\sum_{\substack{l=1\\l\neq j}}^{n} c_l \bar{x}_{il}\right)\bar{x}_{ik} = \sum_{\substack{l=1\\l\neq j}}^{n} c_l \sum_{i=1}^{m} \bar{x}_{ik}\bar{x}_{il} = \sum_{\substack{l=1\\l\neq j}}^{n} c_l v_{lk}.$$

Si une relation linéaire existe entre les colonnes de $\bar{X}$, cette relation s'applique également aux colonnes de V. Enfin, puisque V est une matrice carrée, toute relation entre ses colonnes correspond à un vecteur de son noyau, i.e. un vecteur U tel que $VU = 0$. Une base du noyau de V est donc obtenue en résolvant ce système à l'aide de la méthode d'élimination de Gauss. Chaque vecteur de cette base donne les coefficients d'une relation affine entre les variables x_i, $1 \leq i \leq n$. Pour calculer le terme constant b de cette relation, on utilise les données initiales. Enfin, pour obtenir d'autres types de relation que des formes affines, on peut calculer, pour chaque graphe, de nouveaux invariants obtenus à l'aide d'opérations non-linéaires sur les n invariants déjà connus. De nouvelles relations peuvent alors être obtenues en appliquant le procédé venant d'être décrit sur la matrice augmentée de colonnes refermant les valeurs de ces nouveaux invariants, e.g. $x_{i,n+j} = x_{ij}^2$, $x_{i,2n+j} = log(x_{ij})$, etc.

Une autre approche automatisant la génération de conjectures, appelée *approche algébri-*

que dans (Caporossi et Hansen, 2004), ressemble à la technique employée par INGRID :

1. Trouver les graphes extrémaux ou quasi-extrémaux pour une formule donnée sur des invariants, i.e. des graphes pour lesquels la valeur de la formule est maximum ou minimum.
2. Reconnaître les classes des graphes extrémaux obtenus (cycles, étoiles, cliques, chemins, arbres, etc.).
3. Obtenir d'une base de relations connues, les relations entre les invariants pour cette classe de graphes, e.g. relations simples sur le nombre de sommets et d'arêtes d'un graphe.
4. Combiner ces relations pour obtenir de nouvelles conjectures plus fortes que celles déjà connues.

En tout, AGX a permis de générer quelques centaines de conjectures sur les invariants de graphe. Par exemple, ce système a été employé pour obtenir des résultats portant sur la plus grande valeur propre d'arbres à coloration contrainte (Cvetkovic et al., 2001). De plus, dans le domaine de la chimie computationnelle, AGX a été utilisé pour obtenir des bornes sur la connectivité d'arbres chimiques ainsi que sur l'énergie d'un graphe (Caporossi et al., 1999a; Caporossi et al., 1999b).

2.4.3 Le système GRAPHEDRON

Un dernier système implémenté récemment par Mélot et al., appelé GRAPHEDRON (*Graph Polyhedron*) (Mélot, 2007; Christophe et al., 2008), transforme la tâche de trouver les relations affines entre des invariants d'un graphe en celle d'obtenir les facettes d'un polyèdre dans l'espace des invariants. Cette approche, initialement présentée dans (Caporossi et Hansen, 2004) comme l'*approche géométrique*, procède de la manière suivante. Soit un ensemble de k observations, chacune correspondant à un graphe différent d'une classe donnée de graphes, et pour lesquelles on connaît la valeur de p invariants.

On note x_{ij} la valeur du j-ème invariant calculée sur le i-ème graphe. En considérant, chaque $(x_{i1}, x_{i2}, \ldots, x_{ip})$ comme un point dans l'espace des invariants à p dimensions, toute relation affine entre les p invariants, valide pour chaque observation, est un hyperplan dans l'espace des invariants pour lequel tous les points sont d'un même côté. De plus, étant donné $a_1 x_{i1} + a_2 x_{i2} + \ldots + a_n x_{ip} \geq b$ une inégalité satisfaite pour $1 \leq i \leq k$, on peut toujours trouver une autre relation de force égale ou supérieure, satisfaite *à égalité* pour au moins une observation. En supposant que l'on possède un point pour chaque graphe de la classe observée, les relations de forces maximales sont donc les facettes du polyèdre consituant l'enveloppe convexe des observations dans l'espace des invariants. Une illustration de cette approche, dans le cas de deux invariants i_1 et i_2, est montrée à la figure 2.17.

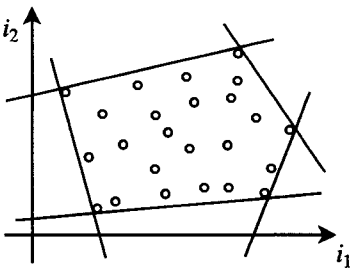


Figure 2.17 Une illustration de l'approche géométrique de GRAPHEDRON.

Pour que cette méthode produise des relations valides, il n'est pas nécessaire d'avoir une observation pour chaque graphe de la classe observée. On doit cependant avoir une observation correspondant à chaque sommet du polyèdre, ce qui peut être fait de deux façons. Premièrement, comme les sommets du polyèdre correspondent à des graphes extrémaux, on peut se limiter à chercher ce type de graphe. L'autre solution, employée par GRAPHEDRON, consiste à produire une observation pour tous les graphes de la classe ne dépassant pas une certaine taille, e.g. 10 sommets ou moins.

CHAPITRE 3

DÉMARCHE ET ORGANISATION DU DOCUMENT

Ce chapitre présente la démarche employée pour mener à bien les objectifs de cette thèse, ainsi que la structure générale du document.

Comme mentionné dans l'introduction, cette thèse comporte quatre buts de recherche. Premièrement, elle vise à développer de nouvelles techniques permettant d'améliorer la découverte des sous-graphes fréquents dans les cas où les graphes possèdent peu d'étiquettes différentes. Le second but de la thèse, relié au premier, est d'améliorer la découverte des patrons fréquents en réduisant le nombre de calculs de support dans la base de données, sans avoir recours à des structures complexes exigeant beaucoup de mémoire. Les troisième et quatrième buts portent sur la génération automatisée de conjectures en théorie des graphes. Il s'agit, dans un premier temps, d'implémenter des méthodes permettant d'automatiser la génération de conjectures portant sur la caractérisation par sous-graphes interdits, et, ensuite, d'utiliser ces méthodes pour trouver de nouveaux théorèmes portant sur ce sujet. Les approches proposées pour atteindre ces buts sont présentées sous la forme de trois articles, correspondant respectivement aux chapitres 3, 4 et 5 de cette thèse.

Le premier article, portant le titre "Improving frequent subgraph mining in the presence of symmetry" (voir (Desrosiers et al., 2007b)), a été soumis pour publication à la revue *Journal of Machine Learning Research*. Cet article propose un nouvel algorithme, appelé SYGMA, permettant de trouver de manière efficace les sous-graphes fréquents dans un ensemble de graphes possédant un nombre réduit d'étiquettes. L'algorithme SYGMA se distingue des autres algorithmes pour le même problème par sa technique d'énumération de graphes. En somme, SYGMA transforme l'espace de recherche en

un arbre orienté, à l'aide d'une fonction p associant à chaque graphe G un graphe parent unique $p(G)$. L'avantage de cette technique est que l'on possède une grande liberté quant au choix de p, nous permettant ainsi de développer des stratégies efficaces. Par exemple, SYGMA emploie dans cette fonction de puissants invariants de graphe, permettant de briser la symétrie associée au nombre réduit d'étiquettes. De même, un choix judicieux de la fonction p nous permet de définir des tests rapides pour détecter les graphes redondants et non-redondants durant l'exploration de l'arbre de recherche. Une autre caractéristique de SYGMA est qu'il utilise l'information sur la symétrie d'un graphe, sous la forme d'équivalences topologiques, i.e. orbites, entre les sommets et les paires de sommets du graphe. Cette information est employée pour éliminer certains calculs redondants, entre autres, pour éliminer les extensions équivalentes menant à des graphes non-fréquents. Dans le but d'évaluer l'efficacité de SYGMA, l'article présente une section expérimentale où l'on compare cet algorithme avec un des algorithmes les plus populaires pour la découverte des sous-graphes fréquents, l'algorithme GSPAN.

Les travaux présentés dans le second article, intitulé "Using background knowledge to improve structured data mining" (voir (Desrosiers et al., 2008)), généralisent les techniques décrites dans le premier article. Également soumis à la revue *Journal of Machine Learning Research*, cet article propose une nouvelle approche permettant d'améliorer la découverte des patrons fréquents, en réduisant le nombre de calculs de support dans la base de données. Cette approche utilise des connaissances de fond sur les données, sous la forme d'une heuristique, pour définir la topologie de l'espace de recherche, de sorte à minimiser le nombre de calculs de support pour des patrons non-fréquents. L'idée est de définir le parent $p(X)$ d'un patron X comme le sous-patron ayant le moins de chance d'être fréquent, selon l'heuristique. Puisqu'un patron dont le parent est non-fréquent ne peut être fréquent, cette technique permet de minimiser le nombre de patrons non-fréquents explorés. Pour évaluer son efficacité, cette approche est testée sur le problème de la découverte des sous-graphes fréquents, dont les résultats sont présentés dans la

section expérimentale de l'article.

Enfin, le troisième et dernier article, dont le titre est "Automated generation of conjectures on forbidden subgraph characterization" (voir (Desrosiers et al., 2007a)) porte sur la génération de conjectures sur la caractérisation par sous-graphes interdits. Cet article, soumis à la revue *Discrete Mathematics*, présente cinq nouvelles méthodes dédiées aux tâches suivantes :

1. Trouver des conditions *suffisantes* pour caractériser une classe de graphes, sous la forme d'*un seul* sous-graphe interdit.
2. Trouver des conditions *suffisantes* pour caractériser une classe de graphes, sous la forme d'un ensemble contenant *plusieurs* sous-graphes interdits.
3. Trouver des conditions *nécessaires* pour caractériser une classe de graphes, sous la forme d'*un seul* sous-graphe interdit.
4. Trouver des conditions *nécessaires* pour caractériser une classe de graphes, sous la forme d'un ensemble contenant *plusieurs* sous-graphes interdits.
5. Trouver des caractérisations par sous-graphes interdits pour une classe de graphes (conditions nécessaires et suffisantes).

La présentation de ces méthodes est faite en deux étapes. On donne, en premier lieu, une description théorique de ces méthodes, et on démontre quelques unes de leur propriétés. Puisque ces méthodes nécessitent l'exploration d'un espace infini de graphes, on donne, dans un deuxième temps, des techniques pratiques pour implémenter ces méthodes : une première technique employant une métaheuristique, telle que la métaheuristique VNS dans AGX, et une seconde technique faisant l'énumération complète des graphes ayant un nombre limité de sommets. Enfin, pour répondre au dernier objectif de cette thèse, l'article comporte une importante section expérimentale dans laquelle on emploie ces méthodes pour générer de nouvelles conjectures sous la forme de conditions suffisantes et/ou nécessaires pour avoir une caractérisation par sous-graphes interdits. Les classes de graphes que l'on cherche à caractériser dans l'article renferment des graphes satisfaisant

à égalité des relations sur les invariants d'*irredondance*, de *domination* et de *stabilité* (ou *indépendance*), formant la chaîne de domination (Haynes et al., 1998).

CHAPTER 4

IMPROVING FREQUENT SUBGRAPH MINING IN THE PRESENCE OF SYMMETRY

4.1 Introduction

Graph mining is a recent discipline which aims to extract useful knowledge from a large amount of structured data modeled as graphs. Already, this discipline plays a key role in important fields like chemoinformatics and bioinformatics, especially in the process of drug discovery. In the next decade, its importance will undoubtedly increase with the emergence of new technologies dealing with a greater amount of structured information, particularly in the Web domain. The discovery of frequent subgraphs is a fundamental task of graph mining which consists in finding statistically significant sub-structures in a database of graphs. Different variants of this task exist, depending on the type of sub-structures we want to obtain. In this paper, we will consider the task of finding the frequent connected edge induced subgraphs of a database. This problem, known as the frequent subgraph mining problem, can be formulated as follows:

Definition 1 (Frequent subgraph mining). *Given a graph database $\mathcal{D}$, the support of a graph G in $\mathcal{D}$, written $sup(G, \mathcal{D})$, is the number of graphs in $\mathcal{D}$ containing G as an edge induced subgraph. Given a minimum support threshold s_{min}, the frequent subgraph mining problem consists in finding the connected graphs frequent in $\mathcal{D}$, i.e. the connected graphs G for which $sup(G, \mathcal{D}) \geq s_{min}$.*

Several approaches have been proposed for this problem, which can be separated in two groups: levelwise and depth-first approaches. As their name implies, levelwise mining techniques, such as AGM developed by Inokuchi et al. (Inokuchi et al., 2000) and FSG

proposed by Kuramochi and Karypis (Kuramochi et Karypis, 2001), explore the graph search space level by level, where each level contains graphs that have one more vertex or edge than the previous one. The frequent graphs of the next level are found by first generating candidate graphs with pairs of graphs of the current level, and then filtering out infrequent ones. The main advantage of such techniques comes from the *Apriori* principle by which a graph is frequent only if all its subgraphs are. Since a graph is explored after its subgraphs, it is possible to eliminate infrequent graphs without having to compute their support, by testing if their immediate subgraphs are frequent. However, levelwise approaches suffer from two problems: the generation of many redundant candidate graphs, and the requirement to store the frequent graphs at each level. Depth-first mining approaches, such as GSPAN proposed by Han and Yan (Yan et Han, 2002), FFSM by Huan et al. (Huan et al., 2003), and GASTON by Nijssen and Kok (Nijssen et Kok, 2004), overcome these problems by exploring the graph search space depth-first. Starting with a graph containing a single frequent vertex or edge, these techniques recursively extend a graph by adding a new edge between two existing vertices, or a new vertex connected to an existing vertex. Since a graph is no more frequent than its subgraphs, there is no need to extend infrequent graphs. Infrequent graphs can thus be pruned implicitly, without the risk of pruning frequent ones. Various experimental studies, see (Yan et Han, 2002) for example, have shown depth-first mining approaches to be superior, in most cases, to levelwise ones, both in terms of computation times and memory requirements.

The difficulty of the frequent subgraph mining problem arises from two tasks: enumerating all the possible subgraphs of database graphs, and calculating the support of these subgraphs in the database. Since the vertices of a graph can be ordered in many ways, a graph can have a great number of topologically equivalent copies, called isomorphic graphs. To enumerate all subgraphs without redundancy, one must compute the canonical representation of a graph, which amounts to solving the graph isomorphism problem.

Furthermore, testing if a graph is contained in a database graph is a well known *NP*-hard problem called subgraph isomorphism problem. In nearly all cases, support computation is the most costly operation of finding the frequent subgraphs of a database. Yet, the complexity of these tasks is somewhat reduced when the database graphs have added information in the form of vertex or edge labels. For instance, one can use labels to limit the vertices that can be paired while testing for subgraph isomorphism. However, if the database graphs are unlabeled or only have a few labels, then the complexity of these problems greatly reduces the size of manageable datasets. Thus far, little attention has been given to such datasets, and current algorithms tend to do very poorly on them. Still, there are many applications which deal with this type of data, mainly in the fields of computer vision, where the information is represented as 2D or 3D meshes, and communication/transportation networks, where the information is mostly topological. Moreover, mining unlabeled subgraphs could yield relations in the database that are both more general and frequent.

In this paper, we present a novel algorithm called SYGMA (Symmetry-free Graph Mining Algorithm) that improves the task of finding the frequent edge induced connected subgraphs of a database containing graphs that have a few or no labels. This algorithm uses various strategies that reduce the impact of symmetry, caused by the limited number of labels, on the tasks of enumerating subgraphs and computing their support in the database. Unlike most algorithms for the same task, ours does not rely on memory-expensive structures that store graph embeddings, since such a strategy is highly inefficient in these cases. To illustrate this, consider the embeddings of an unlabeled complete graph H (i.e., a graph for which all pairs of vertices are connected by an edge) of m vertices into an unlabeled complete graph G of n vertices. For $m = 6$ and $n = 12$, which are realistic values for this problem, there are as much as $\binom{n}{m}m! = 665280$ embeddings of H in G. Also, for the purpose of simplicity, we have limited our algorithm to deal only with vertex labels. Yet, the techniques presented in this paper could easily

be extended to mine other types of subgraphs, such as subgraphs with edge labels, or vertex induced subgraphs.

The rest of this paper is structured as follows. In section 2, we present the details of our algorithm. In section 3, we give some experimental results that compare, on various instances, SYGMA to one of the most popular frequent subgraph mining algorithms, GSPAN. Finally, we conclude this paper with a brief summary of contributions and results.

4.2 The SYGMA Algorithm

4.2.1 Preliminary concepts

A labeled graph is a tuple $G = (V, E, L, l)$, where V is a set of vertices, $E \subset V^2$ a set of edges, L a set of labels, and $l : V \rightarrow L$ is a function that gives a unique label to each vertex of G. Given two labeled graphs $G = (V, E, L, l)$ and $G' = (V', E', L', l')$, we say that G is isomorphic to G', written $G \simeq G'$, iff there exists a bijection $\varphi : V \rightarrow V'$, called isomorphism, such that

1. $(u, v) \in E \Leftrightarrow (\varphi(u), \varphi(v)) \in E'$,
2. $\forall v \in V,\ l(v) = l(\varphi(v))$.

An automorphism is an isomorphism from a graph to itself. Furthermore, a subgraph isomorphism from G to G' is an isomorphism from G to a subgraph of G'. If such an isomorphism exists, we say that G' contains G and write $G \subseteq G'$.

Let Γ be the set of all permutations of V, and let φ be a permutation of Γ. We write G^{φ} the graph with vertex set $V^{\varphi} = V$ and edge set $E^{\varphi} = \{(u, v) \mid \exists (x, y) \in E \text{ s.t. } u =$

$\varphi(x)$ and $v = \varphi(y)\}$. The automorphism group of G is the set containing the automorphisms of G, i.e. the set $Aut(G) = \{\varphi \in \Gamma \mid G^{\varphi} = G\}$. The orbits of a vertex $v \in V$, written $Orb(v)$ is the set of vertices u such that there exists an automorphism mapping v to u, i.e., $Orb(v) = \{u \in V \mid \exists \varphi \in Aut(G) \text{ s.t. } u = \varphi(v)\}$. Similarly, the orbit of a pair of vertices (u, v), written $Orb(u, v)$ is the set of vertex pairs (x, y) such that there exists an automorphism mapping u to x and v to y, i.e., the set $Orb(u, v) = \{(x, y) \in V^2 \mid \exists \varphi \in Aut(G) \text{ s.t. } \varphi(u) = x \text{ and } \varphi(v) = y\}$. A vertex partition of G is an ordered sequence of pairwise disjoint non-empty sets called cells, the union of which is V. Given two vertex partitions π_1 and π_2, we say that π_1 is finer than π_2 if each cell of π_1 is a subset of a cell in π_2. Furthermore, let π be a vertex partition of G and u, v be two vertices of G. We denote $\pi(u)$ and $\pi(v)$ the unique cells containing vertices u and v, and write $\pi(u) < \pi(v)$ if the cell containing u comes befores the cell containing v in π. Likewise, we write $\varphi(u) < \varphi(v)$ if u precedes v in a permutation φ.

Automorphisms can be used to solve the graph isomorphism problem. Indeed, we can determine if two graphs G and G' are isomorphic by finding the canonical representation of these graphs and verifying if these representations are identical.

Definition 2 (Canonical representation). *Let G be a graph such that $|V| = n$ and A be the symmetrical adjacency matrix of G. We define a function code that uniquely maps G to the string produced by concatenating the elements of the upper half of A:*

$$code(G) = (a_{1,2}\ a_{1,3}\ a_{2,3}\ \ldots\ a_{i,j}\ a_{i,j+1}\ \ldots\ a_{n-1,n})\,.$$

The canonical representation of G is thus the lexicographically smallest code produced by any permutation of G, i.e., $\min_{\varphi}\ code(G^{\varphi})$, and we call canonical permutation of G any permutation leading to this representation.

4.2.2 Subgraph Enumeration

The subgraph enumeration strategy used by SYGMA is similar to the one proposed by Kuramochi and Karypis for their algorithm VSIGRAM (Kuramochi et Karypis, 2005), although their algorithm is not made for the frequent subgraph mining problem. Like VSIGRAM, our algorithm uses a partial edge ordering that orders the edges of a graph G following the rank of their vertices in a canonical permutation of G:

Definition 3 (Canonical edge ordering). *Let G be a graph, φ be a canonical permutation of G and $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$ be two edges of G such that $\varphi(u_1) \leq \varphi(v_1)$ and $\varphi(u_2) \leq \varphi(v_2)$. The canonical edge ordering, defined by precedence operator $\prec_E$, is such that $e_1 \prec_E e_2$ iff $Orb(e_1) \neq Orb(e_2)$ and either one of the following is true*

1. $\varphi(u_1) < \varphi(u_2)$
2. $u_1 = u_2$ *and* $\varphi(v_1) < \varphi(v_2)$.

This ordering allows us to transform the search space into a rooted tree by mapping to each graph G a parent graph $p(G)$ produced by removing from G a non-disconnecting edge[1] that is minimum according to $\prec_E$. If a vertex incident to this edge becomes isolated, i.e. adjacent to no vertex of G, this vertex is also removed from G. Thus, $p(G)$ is a connected graph that has exactly one less edge, and possibly one less vertex, than G. The tree resulting from p is then explored depth-first, as shown in Figure 4.1. Starting with a graph G containing a single edge, G is recursively extended until it becomes infrequent. Let e be the last edge added to G, a canonical permutation φ of G is first found using MacKay's NAUTY algorithm (McKay, 1981). In the process, the vertex and vertex pair orbits of G are also obtained, with little added cost. Then, using φ, we find a minimum non-disconnecting edge e^*. If e is not topologically equivalent to e^*, i.e. if

[1] An edge is disconnecting if its removal produces a graph that is not connected.

$Orb(e) \neq Orb(e^*)$, then we can prune G since another graph isomorphic to G will be explored at a different point of the traversal. Note that this is different from the strategy used by VSIGRAM, where G is pruned if $G - \{e\} \simeq G - \{e^*\}$, thus requiring one more graph isomorphism test. Otherwise, we compute the support of G and extend this graph if it is frequent. For the vertex extensions, we consider for each vertex orbit O_V of G a single vertex v, and a possible label λ. We then extend G into a graph G' obtained by adding to G a new vertex of label λ and connect this vertex to v. Similarly, for edge extensions, we consider all orbits of non-connected vertices O_E and a single vertex pair (u, v) in this orbit. We then create a graph G' by adding to G an edge connecting u and v.

Proposition 1. *By traversing depth-first the rooted tree defined by function p, we can explore every graph without redundancy.*

Proof. To prove that every connected graph G is explored by the traversal, we must show that there exists a path in the tree from G to the root of this tree. Since all possible vertex and edge extensions are considered in the traversal, G will be explored if its parent is explored. Furthermore, since the parent of a connected graph is also connected, by recursion, G has for ancestor the root of the tree, and is therefore reachable. Next, consider two isomorphic graphs G and G'. Since the canonical edge ordering is insensitive to vertex permutations, a minimum non-disconnecting edge in G is topologically equivalent to one in G'. Thus $p(G) \simeq p(G')$, and since we only consider one extension per vertex or vertex pair orbit, only one of G or G' will be explored. Therefore, the exploration is not redundant. □

Algorithm SYGMA

Input: A graph database $\mathcal{D}$ and a support threshold s_{min}.
Output: The frequent subgraphs $\mathcal{F}$ of $\mathcal{D}$.
$\mathcal{F} := \emptyset$;

foreach *vertex label* λ_1 *in* $\mathcal{D}$ **do**
  foreach *vertex label* λ_2 *in* $\mathcal{D}$, $\lambda_1 \leq \lambda_2$ **do**
    Let G be the graph with two vertices v_1 and v_2 of label λ_1 and λ_2, and edge (v_1, v_2) ;
    $\mathcal{F} := \mathcal{F} \cup$ `explore` $(\mathcal{D}, s_{min}, G)$;

return $\mathcal{F}$;

Procedure `explore` $(\mathcal{D}, s_{min}, G)$

Input: A graph database $\mathcal{D}$, a support threshold s_{min} and a graph G.
Output: The frequent extensions $\mathcal{F}$ of G.

Let e be the last edge added to G ;
Compute the orbits of G and a canonical permutation φ of G ;
Using φ, find a minimum non-disconnecting edge e^* of G ;

if $Orb(e) \neq Orb(e^*)$ *or* $sup(G, \mathcal{D}) < s_{min}$ **then return** $\emptyset$;

$\mathcal{F} := \{G\}$;

% Vertex extensions
foreach *vertex orbit* O_V *and label* λ *in* $\mathcal{D}$ **do**
  Let v be a vertex in O_V ;
  Let G' be the graph obtained by connecting a new vertex of label λ to v ;
  $\mathcal{F} := \mathcal{F} \cup$ `explore` $(\mathcal{D}, s_{min}, G')$;

% Edge extensions
foreach *non-connected vertex pair orbit* O_E **do**
  Let (u, v) be a vertex pair in O_E ;
  Let G' be the graph obtained by connecting vertices u and v ;
  $\mathcal{F} := \mathcal{F} \cup$ `explore` $(\mathcal{D}, s_{min}, G')$;

return $\mathcal{F}$;

Figure 4.1 The SYGMA algorithm and its recursive procedure *explore*.

Redundant graph detection

While the main lines of our subgraph enumeration strategy are similar to those used in VSIGRAM, our algorithm stands out with its efficient technique to prune redundant graphs. This pruning technique uses a procedure that partitions the vertices of a graph G, as shown in Figure 4.2. The vertices are first ordered by increasing label values and grouped into cells of equal values, forming a partition π_0. Then, at each iteration t, the current partition π_t is refined by considering pairs of cells $V_i, V_j \in \pi$ and by splitting V_j using V_i. Denote $\delta(v, V_i)$ and $\hat{\delta}(v, V_i)$, respectively, the number of non-disconnecting and disconnecting edges incident to a vertex v and any vertex in V_i. The vertices v of V_j are first ordered by decreasing values of $\delta(v, V_i)$ and then by decreasing values of $\hat{\delta}(v, V_i)$. These vertices are then split into groups of equal values, forming a subpartition π'. If π' refines V_j, i.e. if $|\pi'| > 1$, V_j is replaced by π' in the partition. This process is repeated until no further refinement is possible. Finally, the partition π_T, returned by this refinement procedure, is used to obtain a canonical permutation of G: when looking for a canonical permutation of G, we only consider the permutation of vertices within the cells of π_T. A direct consequence of this is the following proposition:

Proposition 2. *Let π_t be the partition of the vertices of a graph G, at any step t of the refinement procedure, and let φ be a canonical permutation of G. For any two vertices $u, v \in V$, if $\pi_t(u) < \pi_t(v)$ then $\varphi(u) < \varphi(v)$.*

The pruning technique used by SYGMA detects non-minimum extensions while refining the vertex partition, as described in the following proposition.

Proposition 3. *Let G be a graph, let π_t be the partition of the vertices of G at any step t of the refinement procedure, and consider any edge $e_1 = (u_1, v_1)$ of G, such that $\pi_t(u_1) < \pi_t(v_1)$. Edge e_1 is non-minimum in G, following $\prec_E$, if there exists a non-disconnecting edge $e_2 = (u_2, v_2)$ such that $\pi_t(u_2) \leq \pi_t(v_2)$, and if either one of the following applies*

Refinement procedure

Input: A graph G.
Output: A refined partition of V.

Let π_0 be the initial partition s.t. $\forall u, v \in V,\ \pi(u) < \pi(v)$ iff $l(u) < l(v)$;
$t := 0$;
repeat
  $\pi_{t+1} := \pi_t$;
  $t := t + 1$;
  foreach *cell* $V_i \in \pi_t$ **do**
    foreach *cell* $V_j \in \pi_t$ *s.t.* $|V_j| > 1$ **do**
      Let π' be the subpartition of V_j s.t. $\forall u, v \in V_j, \pi'(u) < \pi'(v)$ iff
      $\delta(u, V_i) > \delta(v, V_i)$ or $\big(\delta(u, V_i) = \delta(v, V_i)$ and $\hat{\delta}(u, V_i) > \hat{\delta}(v, V_i)\big)$;
      if $|\pi'| > 1$ **then** replace cell V_j by π' ;
until $\pi_t = \pi_{t-1}$ *or* $|\pi_t| = |V|$;
return π_t ;

Figure 4.2 A procedure to find a refined vertex partition.

1. $\pi_t(u_2) < \pi_t(u_1)$
2. $\pi_t(u_2) = \pi_t(u_1)$ *and* $\pi_t(v_2) < \pi_t(v_1)$.

Proof. We prove cases (1) and (2) separately.

1. Following Proposition 2, we have that $\varphi(u_2) < \varphi(u_1)$. Moreover, following Definition 3, we have $e_2 \prec_E e_1$ and thus e_1 is not minimum.

2. (*a*) If there is a vertex $w \in V$ such that (u_1, w) is a non-disconnecting edge and $\pi_t(w) < \pi_t(v_1)$, then, following Proposition 2, we have that $\varphi(w) < \varphi(v_1)$. Moreover, following Definition 3, we have $(u_1, w) \prec_E e_1$ and thus e_1 is not minimum. (*b*) Else, assume there is no vertex $x \in V$ such that (u_2, x) is a non-disconnecting edge and such that $\pi_t(x) < \pi_t(v_2)$, otherwise use x instead of v_2 in what follows. Let V_i, $i = \pi_t(v_2)$, be the cell containing v_2, and let V_j, $j = \pi_t(u_1) = \pi_t(u_2)$, be the cell containing vertices u_1 and u_2. We have that

$$\begin{aligned} \delta(u_2, V_k) = \delta(u_1, V_k) = 0 \quad &, \text{ for } k < i \\ \delta(u_2, V_k) \geq 1 > 0 = \delta(u_1, V_k) \quad &, \text{ for } k = i \end{aligned}$$

   Thus, the partition π' produced by splitting V_j with V_i will be such that $\pi'(u_2) < \pi'(u_1)$ and, following Proposition 2, we have that $\varphi(u_2) < \varphi(u_1)$. Moreover, following Definition 3, we have $e_2 \prec_E e_1$ and thus e_1 is not minimum.

□

Non-redundant graph detection

Like in most graph mining algorithms, the techniques used by SYGMA to detect redundant graphs help avoiding many costly isomorphism tests. These techniques, however,

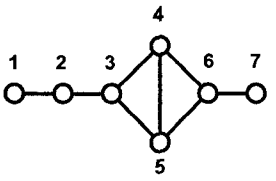
are of no help when dealing with graphs that are not redundant. Unlike other graph mining algorithms, SYGMA can also detect non-redundant graphs without any isomorphism test, as described in the next proposition. The proof of this proposition, related to the fact that all the vertices within two separate cells are either connected or not, can be found in (McKay, 1981).

Proposition 4. *Let π_T be the vertex partition returned by the refinement procedure for a graph G, and let m be the number of cells of π_T that are trivial, i.e. that contain a single vertex. If G is not pruned by Proposition 3, then G is not redundant if the following conditions are satisfied:*

1. $|\pi_T| - m \leq 2$
2. $|V| - m \leq 5$,

i.e. π_T should have at most 2 non-trivial cells and G should have at most 5 non-trivial vertices. If these two conditions are met, then the vertex orbits of G are simply the cells of π_T. Similarly, the vertex pair orbits can also be obtained directly from π_T. Consider any two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ of G. Suppose, without loss of generality, that $\pi_T(u_1) \leq \pi_T(v_1)$ and $\pi_T(u_2) \leq \pi_T(v_2)$. The orbits of non-connected vertex pairs are such that $Orb(e_1) = Orb(e_2)$ iff $\pi_T(u_1) = \pi_T(u_2)$ and $\pi_T(v_1) = \pi_T(v_2)$.

Although it seems that the conditions of Proposition 4 only apply to very specific cases, the reality is that most graphs satisfy these conditions, especially labeled graphs. In fact, as we will see in the experimental section, no isomorphism test is needed for graphs of five or less vertices, regardless the number of vertex labels of these graphs.



(a)

t	π_t
0	(1234567)
1	(456|3|2|17)
2	(45|6|3|2|7|1)

(b)

Figure 4.3 (*a*) A graph and (*b*) its vertex partition at step t of the refinement procedure.

An illustrative example

In this section, we illustrate the subgraph enumeration strategy of SYGMA using a small example. Consider the graph shown in Figure 4.3(*a*), and its vertex partition at each step t of the refinement procedure, shown in (*b*). This graph, that we denote by G, has only one disconnecting edge: $(2, 3)$. Since G is unlabeled, the first partition π_0 groups all its vertices into a single cell. These vertices are then sorted by decreasing number of non-disconnecting edges, and the result sorted by decreasing number of disconnecting edges. Then, the vertices are grouped into cells of equal value, giving the partition $\pi_1 = (456|3|2|17)$. At step $t = 2$, cell (17) is first split using cell (456) into the subpartition $(7|1)$. Cell (456) is then split using cell (3) into subpartition $(45|6)$, yielding $\pi_2 = (45|6|3|2|7|1)$. Finally, this partition cannot be refined any further, and the refinement procedure returns π_2. Since a canonical permutation φ only permutes the vertices within the cells of this partition, the minimum non-disconnecting edge, i.e. the first non-disconnecting edge encountered while following φ, will necessarily be $(4, 5)$. Suppose that the last edge added to G is $(3, 4)$. At step $t = 1$, we have $\pi_1(4) = \pi_1(5) = 1 < 2 = \pi_1(3)$ and, following case (2) of Proposition 3, $(4, 5)$ is a smaller non-disconnecting edge than $(3, 4)$, in any canonical permutation. Therefore, G is redundant and can be pruned. However, if the last edge added to G is $(4, 5)$, the refinement procedure will then go on without G being pruned. In

this case, the partition π_2, returned by the refinement procedure, has one non-trivial cell containing two vertices, cell (45). Thus, following Proposition 4, G is not redundant. Furthermore, the first vertices of each cell of π_2 can be used as the representatives of the vertex orbits of G, i.e. the set $\{4, 6, 3, 2, 7, 1\}$. Finally, we obtain the representatives of the non-connected vertex pair orbits by taking, for each pair of cells $V_i, V_j \in \pi_2$, a pair of non-connected vertices (u, v) where $u \in V_i$ and $v \in V_j$: $\{(4,2),(4,7),(4,1),(6,3),(6,2),(6,1),(3,7),(3,1),(2,7),(7,1)\}$.

4.2.3 Support calculation

As mentioned previously, the important symmetry caused by the reduced number of labels prohibits the use of complex structures to store subgraph embeddings. Instead of relying on such structures, our algorithm solves the subgraph isomorphism problem directly, using a simple subgraph matching method. However, since finding a subgraph isomorphism is a rather complex task, and since our algorithm has to complete this task quite often, we employ some further strategies to calculate the support of a subgraph as efficiently as possible.

Matching constraints

The first strategy is used within the subgraph matching to prune the search space. Suppose we need to determine if a graph $G = (V, E, L, l)$ is a subgraph of $G' = (V', E', L', l')$ and let γ be a possibly partial mapping of V to V', called matching. Let $v \in V$ be any vertex, we define $N(v)$ (resp. $N'(v)$) as the set of vertices adjacent to v in G (resp. G'). Moreover, we define $L(\lambda)$ (resp. $L'(\lambda)$) as the vertices of G (resp. G') which have label λ. We also define $M(\gamma)$ and $\overline{M}(\gamma)$ (resp. $M'(\gamma)$ and $\overline{M}'(\gamma)$) as the vertices of G (resp. G') matched and unmatched under γ. The following proposition gives necessary

conditions for two vertices to be matched.

Proposition 5. *Let $v \in V$, $v' \in V'$ be two vertices. The pair (v, v') is a candidate to extend a matching γ if the following conditions are respected:*

1. $v \in \overline{M}(\gamma)$ *and* $v' \in \overline{M}'(\gamma)$.
2. $l(v) = l'(v')$.
3. $\forall u \in N(v) \cap M(\gamma)$, $\gamma(u) \in N'(v')$.
4. $\forall \lambda \in L$, $|N(v) \cap L(\lambda) \cap \overline{M}(\gamma)| \leq |N'(v') \cap L'(\lambda) \cap \overline{M}'(\gamma)|$.

The first two conditions are rather trivial, stating that vertices v and v' should not already be matched under γ, and that they should have the same label. The third condition imposes γ to be a subgraph isomorphism, i.e., for all vertices of G adjacent to v and matched under γ, the corresponding vertex in G' should be adjacent to v'. Finally, the last condition verifies that the matching can be extended, i.e., that for every vertex label λ, there are at least the same number of unmatched vertices of label λ adjacent to v' as there are adjacent to v.

Avoiding redundant calculations

The next strategy exploits previous calculations to limit the search of a new subgraph isomorphism, and is based on the fact that vertices are matched in a static order. Let $\gamma = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ and $\gamma' = \{(u'_1, v'_1), \ldots, (u'_n, v'_n)\}$ be two matchings such that $m \leq n$, we define a lexicographic order on matchings $\prec_M$, such that $\gamma \prec_M \gamma'$ iff either one of the following applies

1. $\exists k,\ 1 \leq k \leq m,$ s.t. $\begin{cases} u_i = u'_i \text{ and } v_i = v'_i, & i < k \\ u_i < u'_i \text{ or } (u_i = u'_i \text{ and } v_i < v'_i), & i = k \end{cases}$.

2. $u_i = u'_i$ and $v_i = v'_i$, $1 \leq i \leq m$, and $m < n$.

Proposition 6. *Let γ be the minimum subgraph matching of a graph G into a graph H according to $\prec_M$, and let G' be the extension of G with edge e. Any matching γ' of G' into H is such that $\gamma \preceq_M \gamma'$.*

Proof. We prove this by contradiction. Suppose that $\gamma' \prec_M \gamma$. If e is a vertex extension, let θ be the matching such that

$$\theta = \{(u'_1, v'_1), (u'_2, v'_2), \ldots, (u'_{n-1}, v'_{n-1})\},$$

i.e., γ' without the last pair. Otherwise, if e is an edge extension, then consider $\theta = \gamma'$. Following the definition of an isomorphism, we know that θ is also a matching of G into H. Furthermore, according to $\prec_M$, we have that $\theta \preceq_M \gamma' \prec_M \gamma$. However this contradicts the minimality of γ and, consequently, $\gamma \preceq_M \gamma'$. □

Proposition 6 is used in the following way. Let G be any subgraph visited during the exploration. We store the minimum matchings of G into all the database graphs containing G. Then, when G is extended, we only search for matchings superior or equal to the previous ones, according to $\prec_M$. Let N_V be the maximum number of vertices of a database graph, and N_E be the maximum number of edges of a database graph, the total memory requirement of this strategy is in $\mathcal{O}(|\mathcal{D}| \cdot N_V \cdot N_E)$, which is much lower than the memory required to store all the embeddings of G in the database.

Infrequent graph detection

The last strategy allows to detect extensions leading to infrequent graphs, based on the following proposition.

Proposition 7. *Let G' be the extension of a graph G with edge e, and consider any graph H such that $G \subset H$. If G' is not frequent then the extension H' of H with edge e is not frequent.*

Proof. Since $G \subset H \subset H'$ and because H' contains e, we have that $G' \subset H'$. Moreover, since the support of a graph is no greater than the support of its subgraphs, we have $sup(G', \mathcal{D}) \geq sup(H', \mathcal{D})$. Thus, if G' is not frequent, neither is H'. □

When the extension of a graph G with edge e is found infrequent, we store e and all equivalent edges, i.e. the edges with the same vertex pair orbit, as invalid extensions. Then, while exploring the descendants of G in the search tree, we do not consider these invalid extensions since, by Proposition 7, they lead to infrequent graphs.

4.3 Experimentation

To evaluate the performance and validity of our algorithm SYGMA, we have conducted two numerical experiments. In the first one, we validate the subgraph enumeration strategy of our algorithm by generating all graphs with a limited number of vertices and labels. In the second one, we benchmark our algorithm on synthetic and real-life datasets. In both experiments, we compare the results we obtained with those obtained with one of the most popular frequent subgraph mining algorithms, GSPAN, developed by Yan and Han (Yan et Han, 2002). We have selected this algorithm for two reasons. First, like our algorithm, GSPAN does not use any memory-expensive structure to store the embeddings of a graph in the database. Second, a recent investigation by Wörlein et al. (Wörlein et al., 2005), comparing the principal algorithms for this problem, has shown that algorithms storing embeddings offer no real advantage over GSPAN for large instances. All experiments were carried out on a 2.0GHz Intel Pentium IV PC with 512Kb

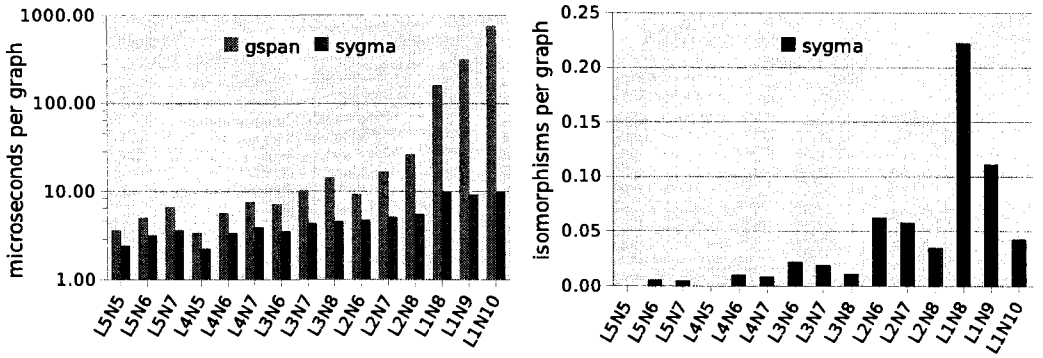
cache and 1Gb RAM, running Linux CentOS release 4.2.

4.3.1 Subgraph enumeration

In the first experiment, we consider the task of exhaustively generating a large set of graphs. More precisely, given integers N and L, we want to generate all connected graphs that have at least one edge, at most N vertices, and at most L vertex labels. This experiment serves two purposes: validating that the subgraph enumeration strategy is sound and complete, and evaluating how well this strategy deals with graph isomorphism. As reference, we compare our algorithm with the subgraph enumeration employed by GSPAN. However, since the available version of GSPAN does not allow to simply enumerate graphs, we had to implement our own version of GSPAN, optimizing as much as possible the algorithm. For the other experiment, though, we used the original version of GSPAN.

Figure 4.4 summarizes the result of this experiment: (*a*) gives the average CPU time in microseconds per non-redundant graph generated (*the Y axis has a logarithmic scale*), and (*b*) the average number of full isomorphism tests per non-redundant graph. Since GSPAN has no strategy to detect non-redundant graphs, without carrying out an isomorphism test, its average number of full isomorphism tests per non-redundant graph is 1.0, for all values of L and N. From this figure, we make the following observations. While GSPAN shows exponential scaling to the decrease of L and increase of N, our algorithm is little affected by these changes. Thus, the average CPU time per non-redundant graph found by GSPAN ranges from 3.6 μsec, for $L = 5$ and $N = 5$, to 761.3 μsec, for $L = 1$ and $N = 10$, which corresponds to a 210-fold increase. By contrast, the average CPU time of our algorithm ranges from 2.4 μsec to 9.9 μsec, for the same values of L and N, which corresponds to a 3-fold increase. Furthermore, SYGMA outperforms GSPAN for all values of L and N. In the most extreme case, for $L = 1$ and $N = 10$, the sub-

graph enumeration strategy used by SYGMA is almost more than 75 times faster than GSPAN's. Finally, we can see that only a small fraction of non-redundant nodes required SYGMA to perform an isomorphism test, and that this fraction decreases as L and N increase. For cases where $N \leq 5$, no isomorphism tests were needed, regardless of the value of L.



(a) (b)

Figure 4.4 Results on subgraph enumeration: (*a*) average CPU time (in microseconds) per non-redundant graph generated, and (*b*) the number of full isomorphism tests per non-redundant graph.

4.3.2 Frequent subgraph mining

In the second experiment, we compare the performance of our algorithm to the latest version of GSPAN, on the task of finding the frequent subgraphs of synthetic and real-life datasets.

Synthetic data

The synthetic datasets were generated with the random graph generator implemented by Karypis and Kuramochi for their work in (Kuramochi et Karypis, 2001), using combinations of values of 5 parameters, whose description and values are given in the following

table:

	Description	Values
D	Nb. of graphs in the database	1000
T	Avg. size of the database graph	$\{15, 20, 25\}$
F	Avg. nb. of frequent subgraphs	25
I	Avg. size of the frequent subgraphs	15
L	Nb. of vertex labels in the database	$\{1, 2, 3\}$

The values used for these parameters were selected to give a good range of difficulty, as well as a suitable balance between the time spent enumerating the subgraphs and the time spent calculating the support of these subgraphs in the database. Since mining graphs with a few labels requires more time, the databases we have generated for this experiment contain less graphs than those commonly reported in the literature, which usually contain around 10000 graphs. The values used for the other parameters, however, are fairly standard for benchmarking graph mining algorithms.

Figure 4.5 summarizes the results. It shows, for each dataset, the CPU time in seconds required by GSPAN and SYGMA to find the frequent subgraphs, for decreasing support thresholds (*the Y axis has a logarithmic scale*). As expected, the CPU time increases exponentially as we lower the support threshold, because of the hard subgraph isomorphism task. Furthermore, for identical values of T and support threshold, the CPU time increases as the number of vertex labels decreases, since there are more frequent subgraphs to discover. From these results, we see that SYGMA is faster than GSPAN by up to two orders of magnitude for unlabeled graphs. In the most extreme case, i.e. when $L = 1$, $T = 25$ and the support threshold is 100%, SYGMA is 110 times faster than GSPAN. Our algorithm also outperforms GSPAN for datasets with 2 and 3 labels, although this improvement is not as substantial.

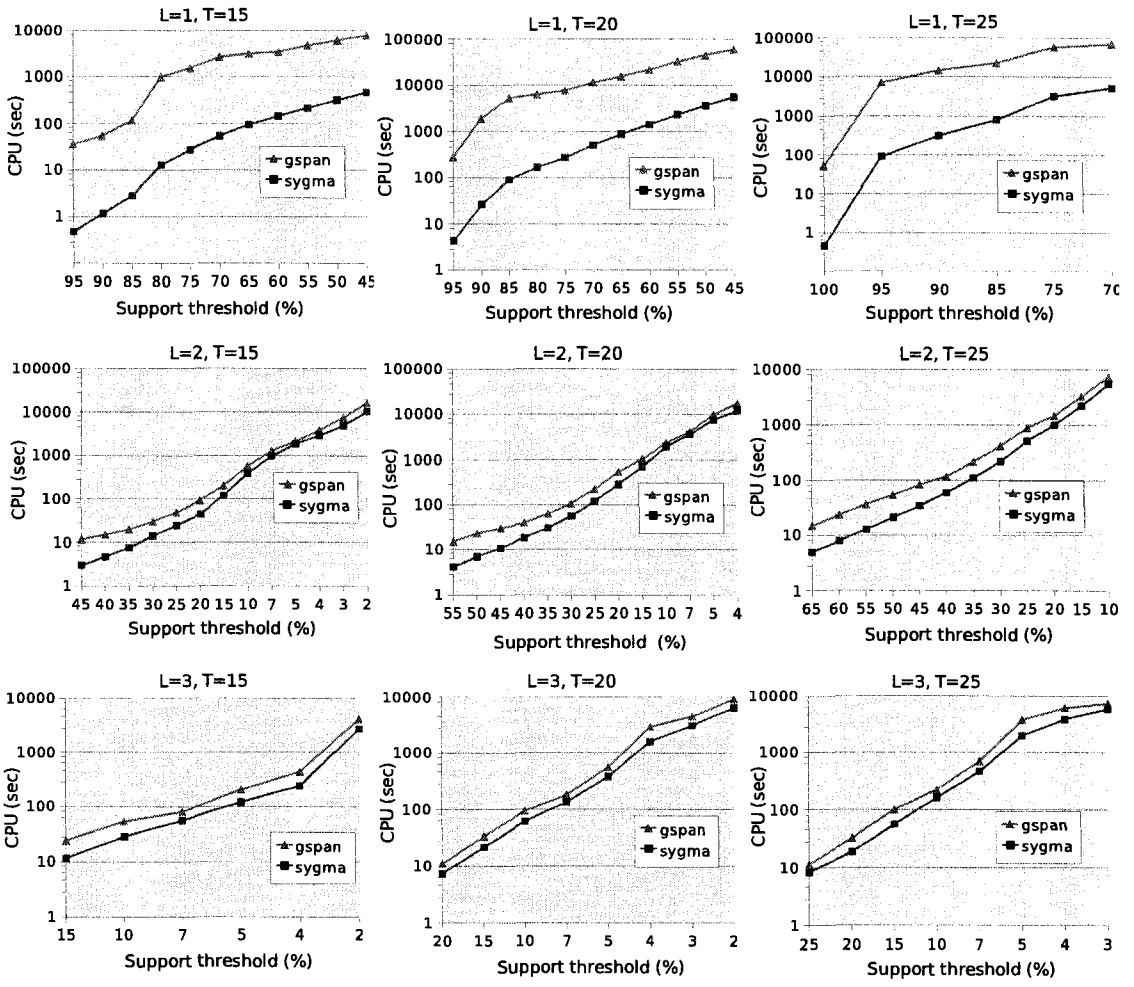
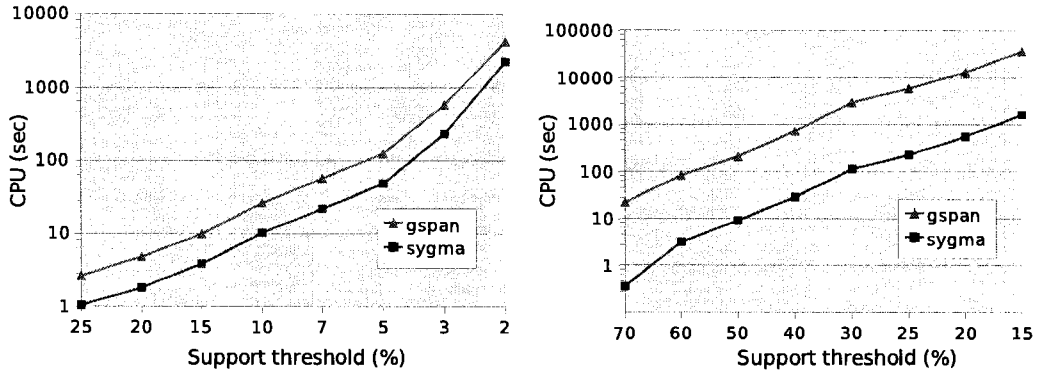


Figure 4.5 Runtimes of GSPAN and SYGMA on synthetic datasets.

Chemical compound data

The performance of SYGMA and GSPAN was also measured on a real-life dataset of chemical compounds, devised for the Predictive Toxicology Evaluation (PTE) challenge (Srinivasan et al., 1997). Two experiments were carried out: in the first experiment, the edge labels of the dataset were discarded, and in the second one, both vertex and edge labels were removed. The results of these experiments are presented in Figure 4.6. Once more, we notice that the runtimes increase exponentially as the support is reduced.

Also, for identical support values, the runtime is much greater on the dataset containing no labels. Comparing both algorithms, we see that SYGMA is 2 to 3 times faster than GSPAN on the dataset with labels, and about 25 times faster the GSPAN on the unlabeled dataset, showing once more the advantage of SYGMA for mining unlabeled graphs.



(a) (b)

Figure 4.6 Runtimes of GSPAN and SYGMA on the PTE dataset for which (*a*) edge labels were discarded, and (*b*) both vertex and edge labels were removed.

4.4 Conclusion

We have presented in this paper a novel algorithm that improves mining the frequent subgraphs of a database that has a few or no labels. This improvement is achieved through original strategies that reduce the number of costly graph and subgraph isomorphism tests, without using memory-expensive structures to store embeddings. We have shown experimentally that our algorithm significantly outperforms one of the most popular algorithm for this task, GSPAN, on various synthetic and real-life datasets.

CHAPTER 5

USING BACKGROUND KNOWLEDGE TO IMPROVE STRUCTURED DATA MINING

5.1 Introduction

Recently, applications dealing with structured information have appeared in various fields. This enhanced information has a richer content that enables a more precise representation of the environment to model. Structured data mining is a discipline that plays a key role in important fields such as bioinformatics, in particular drug design, (Wang et al., 2005b; Borgelt et al., 2005; Sternberg et al., 1995) and Web technologies (Liu, 2007; Chakrabarti, 2002), and which aims at extracting useful knowledge from a great amount of structured information. At the centre of this discipline lies the problem of finding the frequent patterns of a database, which can be defined as follows. Let $\mathcal{D}$ be a database and denote $sup(X, \mathcal{D})$ the support of the pattern X in $\mathcal{D}$, i.e. the number of patterns of $\mathcal{D}$ that have X as a subpattern. When the context is clear, we may write the support of X in $\mathcal{D}$ simply as $sup(X)$. Given an integer s_{min}, called the minimum support threshold, we say that X is frequent if $sup(X, \mathcal{D}) \geq s_{min}$. The frequent pattern mining problem, for a given database $\mathcal{D}$, consists in finding the patterns that are frequent in $\mathcal{D}$. A well-known specialization of this problem is the frequent subgraph mining problem, where the database contains graphs and the goal is to find the graphs the are isomorphic to a subgraph of at least s_{min} graphs of the database.

The frequent pattern mining problem can be decomposed in two tasks: 1) uniquely enumerate all possible patterns and 2) calculate the support of these patterns in the database. In most cases, computing the support of a pattern in the database is a complex oper-

ation, and the second task accounts for most of the time required to find the frequent patterns. Thus, in the case of frequent subgraph mining, testing if a graph is contained in another graph is known as the subgraph isomorphism problem which is *NP*-hard. Frequent pattern mining algorithms are distinguished by their enumeration strategy, which can be horizontal or vertical. Horizontal mining algorithms, such as AGM developed by Inokuchi et al. (Inokuchi et al., 2000) and FSG proposed by Kuramochi and Karypis (Kuramochi et Karypis, 2001), traverse the pattern space level-by-level, where the level k contains the patterns of size k, called k-patterns. Most often, this traversal is done in a bottom-up fashion, i.e. starting with the smallest patterns and successively enumerating patterns of increasing size. This approach allows to prune infrequent patterns that do not satisfy the downward closure property: since the support of a pattern is anti-monotone, a k-pattern is frequent only if all its (k-1)-patterns are frequent. On the other hand, vertical algorithms, like GSPAN proposed by Han and Yan (Yan et Han, 2002), FFSM by Huan et al. (Huan et al., 2003), and GASTON by Nijssen and Kok (Nijssen et Kok, 2004), explore the pattern space depth-first, recursively extending a k-pattern before visiting another pattern of the same size. This approach implicitly prunes infrequent patterns by only extending frequent ones, and has the advantage of requiring much less memory than the horizontal mining approach.

In order to have efficient frequent pattern mining algorithms, it is necessary to devise strategies to improve support computation. The strategy employed by most vertical mining algorithms consists in storing the embeddings of a pattern in the database, and updating these embeddings when the pattern is extended. Although this accelerates support computation on smaller databases, it is not efficient on large databases or when patterns can be embedded in many different ways due to pattern isomorphism. Another strategy is to check the downward closure of patterns before calculating their support. This can be done either by using horizontal mining algorithms or with a special traversal order that ensures that every pattern is explored after its subpatterns, see, e.g., (Cohen et Gudes,

2004). Again, this strategy has some disadvantages. First, horizontal mining algorithms are known to be much less efficient than vertical mining algorithms. Also, it may not be possible to find an efficient depth-first traversal order that ensures visiting a pattern after its subpatterns. Finally, checking the downward closure requires to store all the frequent patterns and to perform an isomorphism test on all the (k-1)-subpatterns of a k-pattern, which can be very time consumming.

In this paper, we propose a simple strategy that can greatly improve frequent pattern mining by avoiding some costly support computations. This strategy, which can be used alone or in combination with another one, such as storing embeddings, uses background information on the frequent patterns to avoid exploring infrequent ones. The rest of the paper is as follows. In section 2, we present our approach. Then, in section 3, we evaluate our approach on the problem of frequent subgraph mining. Finally, we close this paper with a short summary of our contributions and results.

5.2 A general approach

As it is the case for graphs, the pattern space can often be represented in the form of a lattice-like structure where the predecessors of a k-pattern are its (k-1)-subpatterns[1]. However, since a k-pattern can have many (k-1)-subpatterns, a depth-first traversal of the pattern space might visit the same pattern more than once. To visit each pattern only once, we need to transform the lattice into a rooted tree with a function p that assigns to each pattern X a unique parent $p(X)$. For frequent subgraph mining, the parent $p(G)$ of a graph G is typically a graph produced by removing a single vertex or edge from G. The only requirement for p is that it is insensitive to isomorphism: let X and Y be two isomorphic patterns, written $X \simeq Y$, we must have $p(X) \simeq p(Y)$. Once the pattern

[1]The pattern space is usually a *semilattice* since it may be closed under either *join* or *meet*

space is transformed into a rooted tree, we can then uniquely enumerate each pattern by traversing this tree, using either a depth-first or breadth-first traversal.

Proposition 1. *Let p be a parent function such that $p(X) \simeq p(Y)$ if $X \simeq Y$, and suppose that each (k+1)-extension of a k-pattern is explored once. The traversal of the rooted tree defined by function p explores every pattern exactly once.*

Proof. We prove this by recursion. Consider any pattern X and suppose that the parent $p(X)$ of X is explored exactly once, i.e. either $p(X)$ or a pattern isomorphic to $p(X)$ is explored. We will show that X is also explored once. Since every extension of $p(X)$ is explored, we know that X is explored if $p(X)$ is. Furthermore, consider a pattern Y isomorphic to X. By definition, we have that $p(X) \simeq p(Y)$. However, since either $p(X)$ or $p(Y)$ is explored and since every possible extension of $p(X)$ and $p(Y)$ is considered exactly once, either X or Y will be explored. Finally, because the root of the search tree is an ancestor of every other pattern in the tree, and since the root is explored exactly once, by recursion, every pattern is explored only once. □

The main idea of our approach is to select the parent function in a way that minimizes the number of infrequent patterns explored in the search. This is done as follows. Consider any k-pattern X and denote $S(X)$ the (k-1)-subpatterns of X. If X satisfies the *downward closure*, also known as the *Apriori principle*, then it can be either frequent or not, and we need to compute its support. Otherwise, we know that it is not frequent. Since we only need to extend frequent patterns in the traversal, X will not be visited if its parent is infrequent. Thus, to make the visit of an infrequent k-pattern X as unlikely as possible, we must select the parent of X as one of the (k-1)-subpatterns of X that are least likely to be frequent. This idea is formalized in the following propositions.

Proposition 2. *Let p and p' be two parent functions such that $sup(p(Z)) \leq sup(p'(Z))$, for all patterns Z. Every pattern X explored in the traversal of the search space defined by p is also explored in the search space defined by p'.*

Proof. Since a pattern is explored if and only if its parent is frequent, we have that $sup(p(X)) \geq s_{min}$. Furthermore, since $sup(p(Z)) \leq sup(p'(Z))$ for every pattern Z, we have, in particular, $sup(p'(X)) \geq sup(p(X)) \geq s_{min}$. Therefore, X is also explored in the search space defined by p'. □

Proposition 3. *Denote p^* the parent function that maps, for any k-pattern X, a (k-1)-subpattern of X with the lowest support, i.e. $sup(p^*(X)) = \min_{Y \in S(X)} sup(Y)$. A k-pattern is explored in the search space defined by p^* if and only if it is downward closed[2], i.e. if all its (k-1)-subpatterns are frequent.*

Proof. Let X be any k-pattern explored in the search. We know that $sup(p^*(X)) \geq s_{min}$. Furthermore, since $p^*(X)$ is a (k-1)-subpattern of X with the lowest support, we have, for every (k-1)-subpattern $Y \in S(X)$, $sup(Y) \geq sup(p^*(X)) \geq s_{min}$. Thus, all ($k$-1)-subpatterns of X are frequent and X is downward closed. □

By selecting as the parent of a k-pattern X the (k-1)-subpattern with the lowest support, we can thus minimize the number of explored patterns and, consequently, the number of support computations. Finding the optimal parent function p^*, however, is as difficult as finding the frequent patterns. Yet, in many cases, the intrinsic nature of the data makes some patterns more likely to be frequent than others. Our approach consists in using this background information to select p, as follows. Suppose we have a function h, called heuristic, that evaluates the likeliness of a pattern X to be frequent, i.e. if $h(X) > h(Y)$ then X is more likely to be frequent than Y. This function can be determined using some general knowledge on the type of data used, by extracting information from the database in a pre-processing step, or learned by any machine learning algorithm on a training dataset. The only requirements are that h should be easy to compute, and should evaluate to the same value for isomorphic patterns, i.e. $X \simeq Y \Rightarrow h(X) = h(Y)$, otherwise

[2]*Not to be confused with a common use of term* closed *in the litterature, where a pattern X is closed iff it is not contained in a pattern Y of equal support.*

isomorphic patterns could be explored redundantly from different parents. Furthermore, to avoid exploring patterns that are not downward closed, we must then choose the parent of X as a (k-1)-subpattern which minimizes h. However, since two non-isomorphic (k-1)-subpatterns can have the same value of h, this heuristic may not be powerful enough to guarantee that isomorphic patterns have the same parent. To insure this, we also need to define a total order on the pattern space, given by a precedence operator $\prec_P$. The parent of a k-pattern X can then be uniquely defined as the (k-1)-subpattern, among those with minimal value of h, that is also minimal with respect to $\prec_P$.

Figure 5.1 summarizes our approach for the depth-first traversal of the pattern space. Starting with the root pattern $\perp$, the algorithm launches the depth-first traversal by calling the recursive procedure *explore*. This procedure takes as input a database $\mathcal{D}$, a minimum support threshold s_{min} and the current k-pattern X, and returns a set $\mathcal{F}$ containing the frequent patterns of the sub-space rooted at X. The procedure first calculates the support of X. If its support is less than s_{min}, the empty set is returned. Otherwise, X is added to $\mathcal{F}$ and is extended as follows. For each possible (k+1)-extensions Y of X, the procedure computes the parent of Y as the first k-subpattern Z of Y with minimal value of h, following order $\prec_p$. If Z is isomorphic to X then X is the parent of Y. In this case, the recursive procedure *explore* is called on Y, and set of frequent patterns it returns is added to $\mathcal{F}$. When all extensions have been tested, the procedure returns $\mathcal{F}$.

5.3 Experimentation

In this section, we evaluate our approach on the frequent subgraph mining problem, using two different types of data: synthetic and real-life.

Depth-first frequent pattern mining algorithm

Input: A database $\mathcal{D}$ and a minimum support threshold s_{min}.
Output: The frequent patterns $\mathcal{F}$ of $\mathcal{D}$.
$\mathcal{F} :=$ `explore` $(\mathcal{D}, s_{min}, \perp)$;
return $\mathcal{F}$;

Procedure `explore` $(\mathcal{D}, s_{min}, X)$

Input: A database $\mathcal{D}$, a support threshold s_{min} and a k-pattern X.
Output: The frequent patterns $\mathcal{F}$ of the sub-space rooted at X.

if $sup(X, \mathcal{D}) < s_{min}$ **then return** $\mathcal{F} = \emptyset$;

$\mathcal{F} := \{X\}$;

foreach *(k+1)-extension* Y *of* X **do**
  $P := \perp$;
  foreach *k-subpattern* Z *of* Y, *following* $\prec_P$ **do**
    if $P = \perp$ *or* $h(Z) < h(P)$ **then** $P := Z$;
  if $P \simeq X$ **then** $\mathcal{F} := \mathcal{F} \cup$ `explore` $(\mathcal{D}, s_{min}, Y)$;

return $\mathcal{F}$;

Figure 5.1 Our approach to depth-first frequent pattern mining.

5.3.1 Synthetic data

In the first experiment, we considered the task of finding the frequent connected[3] subgraphs of synthetic datasets whose labels have different distributions. To generate this data, we first produced 6 label probability distributions D_i, $i = 1, \ldots, 6$, as follows. For each distribution D_i, we created 8 classes C_j, $j = 1, \ldots, 8$, to which we randomly assigned a unique label from the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Then, for each class C_j, we obtained the probability p_{ij} of a vertex in D_i having the label of C_j with the formula

$$p_{ij} = \frac{\left(1 + \frac{i-1}{\alpha}\right)^{j-1}}{\sum_{k=1}^{8} \left(1 + \frac{i-1}{\alpha}\right)^{k-1}},$$

using $\alpha = 5$ as parameter. As shown in Figure 5.2, the probability distributions obtained in this way are increasingly skewed. Thus, for the first distribution, all labels are equally probable. However, in the last distribution, the label of class C_8 has a 50% chance of being on a vertex, while the probability of having a vertex with the label of class C_1 is only 0.4%. For each of these 6 distributions, we then generated 3 datasets using the random generator developed by Karypis and Kuramochi (Kuramochi et Karypis, 2001), each dataset containing 10000 unlabeled graphs. As parameters, we used an average size of the database graphs of 15, an average number of frequent subgraphs of 25, and an average size of the frequent graphs of 15. These values are fairly standard for benchmarking, and ensure that most of the CPU time is spent on support calculation. Finally, we labelled the vertices of the generated datasets using their respective distribution. The graph edges were all given the same label.

We then tested four frequent subgraph mining algorithms on these datasets. The first algorithm, referred to as *heuristic* in the results, is based on an algorithm called SYGMA, that we developed to find the frequent connected subgraphs of datasets having a lim-

[3] A graph G is connected if there is a path connecting any two vertices of G.

Class	Distribution					
	D_1	D_2	D_3	D_4	D_5	D_6
C_1	12.5	6.1	2.9	1.4	0.7	0.4
C_2	12.5	7.3	4.1	2.3	1.3	0.8
C_3	12.5	8.7	5.7	3.7	2.4	1.6
C_4	12.5	10.5	8.0	5.9	4.3	3.1
C_5	12.5	12.6	11.2	9.4	7.7	6.3
C_6	12.5	15.1	15.6	15.0	13.8	12.5
C_7	12.5	18.1	21.9	24.0	24.9	25.1
C_8	12.5	21.7	30.6	38.4	44.9	50.2

Figure 5.2 Probability (%) of having a vertex with the label of classes C_j, $j = 1, \ldots, 8$, for distributions D_i, $i = 1, \ldots, 6$.

ited number of labels (Desrosiers et al., 2007b). Like SYGMA, the *heuristic* algorithm transforms the search space into a rooted tree using a parent function p, and explores this tree depth-first. However, *heuristic* differs from SYGMA in the fact that it uses background knowledge to define p. In this case, the background knowledge used is the number of edges, in the database, with incident vertices of given labels. This information is obtained, with little added cost, while reading the database. Let G be a graph explored in the search and let φ be a permutation of the vertices of G. The code of G, under φ, is the string obtained by considering the elements of the adjacency matrix of a graph G, following the order of the vertices in φ. Likewise, the canonical code of G is the lexicographically minimal code, obtained under any permutation. This code can be obtained efficiently using, for instance, McKay's NAUTY algorithm. Let φ^* be a permutation leading to the canonical code, and denote $\varphi^*(v)$ the position of a vertex v in φ^*. We define a total order on the edges of G using the precedence operator $\prec_E$, defined as follows. Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be two edges such that $\varphi^*(u_1) < \varphi^*(v_1)$ and $\varphi^*(u_2) < \varphi(v_2)^*$, we have $e_1 \prec_E e_2$ iff either one of these two cases is true:

1. $\varphi^*(u_1) < \varphi^*(u_2)$
2. $u_1 = u_2$ and $\varphi^*(v_1) < \varphi^*(v_2)$.

This order has the property that equivalent edges in two isomorphic graphs are ordered in the same fashion. Since our depth-first search only explores connected graphs, a graph is explored only if its parent is connected. Denote $G-\{e\}$ the graph obtained by removing from G an edge e and all the vertices that become isolated after this removal, i.e. the vertices that are only incident to edge e. We say that e is a disconnecting edge if $G-\{e\}$ is unconnected. We then define the parent of G, in the *heuristic* algorithm, as the graph produced by removing from G the first non-disconnecting edge, following $\prec_E$, whose incident vertex labels are most common in the database, i.e. for which the number of edges in the database having the same pair of labels is the greatest. Let e_1 and e_2 be any two edges of G and denote N_1 and N_2, respectively, the number of edges of the database that have the same label pair as e_1 and e_2. The heuristic function h, in this case, is such that $h(G-\{e_1\}) < h(G-\{e_2\})$ if $N_1 > N_2$. If e_1 and e_2 are topologically equivalent edges, i.e. if there is an automorphism mapping the vertices of one edge to those of the other, then $G-\{e_1\} \simeq G-\{e_2\}$. Since e_1 and e_2 necessarily have the same label pair, we have $h(G-\{e_1\}) = h(G-\{e_2\})$. Thus, h satisfies the requirement to evaluate to the same value for isomorphic graphs.

The second algorithm, called *random*, is another implementation of our approach for which the parent of a graph is selected randomly. In order that isomorphic graphs have the same parent, we re-initialize the random number generator for every visited graph G, using the canonical code of G. We then enumerate the non-disconnecting edges of G following $\prec_E$, and generate, for each of these edges, a random number. The parent of G is then obtained by removing from G the first enumerated edge for which the random number was highest.

The third algorithm, named *prefix*, is our own implementation of the depth-first search (DFS) coding scheme of GSPAN (Yan et Han, 2002). Briefly, this scheme assigns to each graph G a code obtained by concatenating the vertex indexes and labels of the edges of G, following the order in which these edges where added to G. A depth-first search

exploration of the graph space is then made, in such a way that the graphs are visited in ascending code values. Thus, when visiting a graph G, if the code of G is not minimal, then a graph isomorphic to G has already been visited in the search and G is pruned. The *prefix* algorithm uses the same support computation procedures as *heuristic* and *random*.

Finally, the last algorithm, called *dw-closure*, is a variation of *prefix* that explores the search space so that the subgraphs of a graph G are explored before G, thus allowing to check the downward closure of G. The search strategy employed by *dw-closure*, called reverse prefix search (Cohen et Gudes, 2004), differs from the one used by *prefix* in the fact that the extensions of a graph are explored in descending code values, and that the frequent graphs found in the search are stored. Again, *dw-closure* uses the same procedures to compute the support of a graph as the other three algorithms.

Figures 5.3 and 5.4 report, for each of the six distributions, the average results over their three datasets. On the left side are shown the percentage of visited graphs that are downward closed and that are frequent, as found by the *dw-closure* algorithm for decreasing support thresholds[4]. The smaller the ratio of downward closed graphs is, the more graphs can be pruned by checking downward closure. Because it tends to avoid exploring graphs that are not downward closed, this ratio also serves as an indication of the potential benefits of the *heuristic* algorithm. The right side of these figures gives the runtimes, in seconds, of the four tested algorithms, also for decreasing support values. From these results, we can make several observations. First, we notice that the percentage of visited graphs that are frequent remains fairly constant (values range between 7% and 13%), for all distributions and support thresholds. On the other hand, the ratio of visited graphs that are downward closed decreases as the label distribution becomes more skewed, and as the support threshold is lowered. As a consequence, the efficiency of the *heuristic* and *dw-closure* algorithm, compared to the *random* and *prefix* algorithms,

[4]The threshold values are given as a percentage of the dataset graphs.

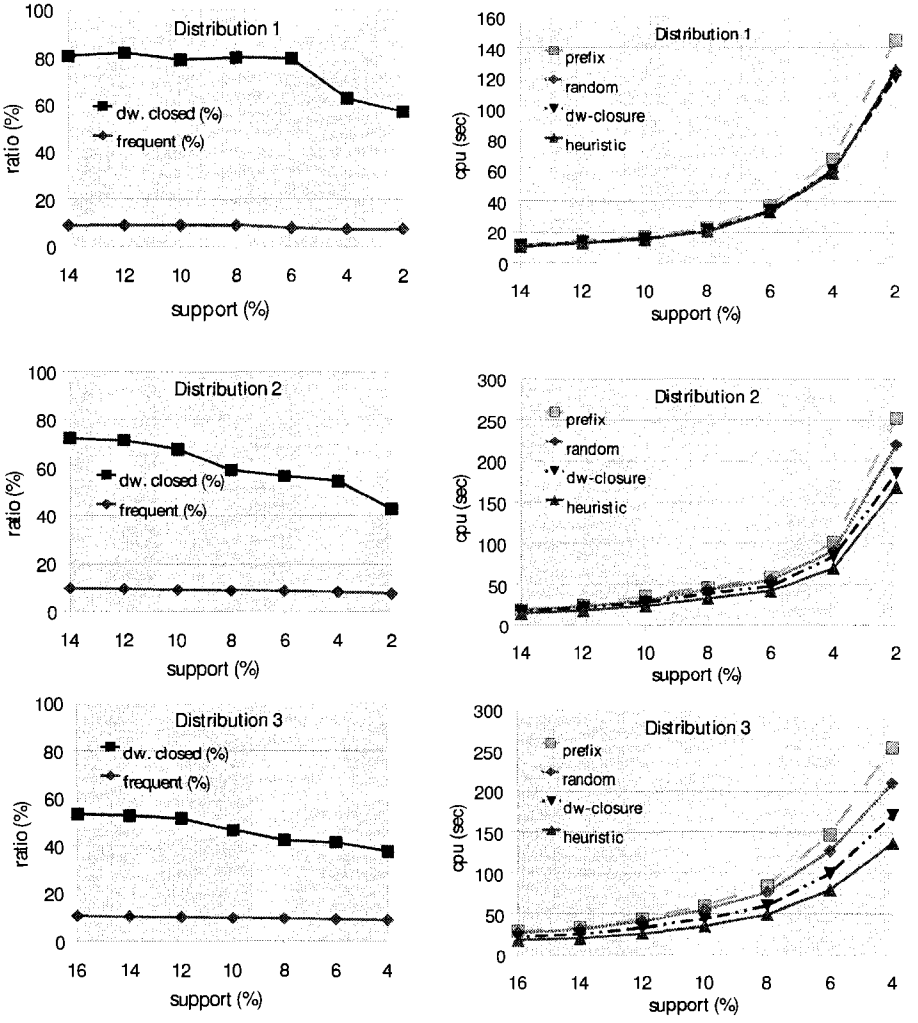


Figure 5.3 Ratio of downward closed and frequent visited graphs (*left*) and runtimes of the tested algorithms (*right*), for synthetic datasets using label distributions D_1, D_2, D_3.

increases in the same way. Thus, for the equiprobable distribution, D_1, the runtimes of all four algorithms are roughly the same. However, for the most skewed distribution, D_6, both *heuristic* and *dw-closure* algorithms present a two-fold speedup over the other two algorithms. We also observe that the *random* algorithm is somewhat faster than the *prefix* algorithm (5% to 17% faster), due to the fact that the fixed lattice exploration order of *prefix* is not well suited for the data. We finally notice that our *heuristic* approach is faster (up to 22% faster) than the *dw-closure* algorithm, in all but one case (distribution D_1, support threshold 2%). Although one might think that checking the downward clo-

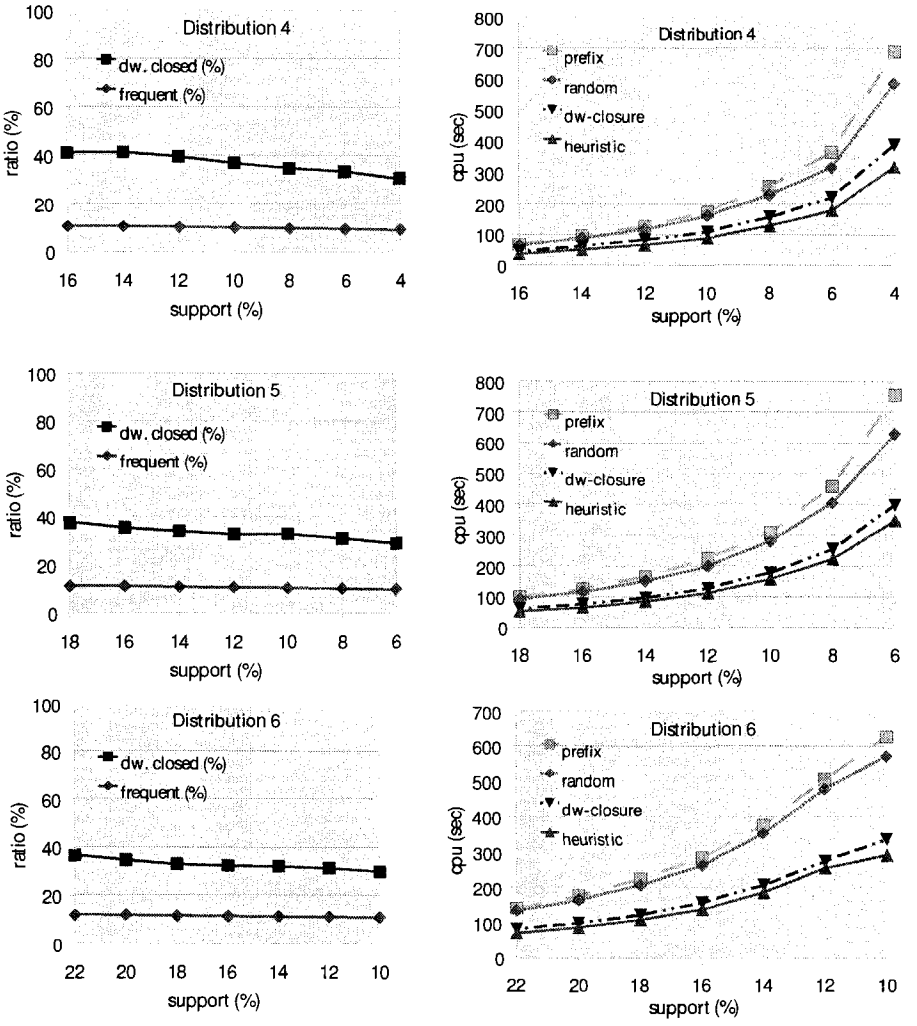


Figure 5.4 Ratio of downward closed and frequent visited graphs (*left*) and runtimes of the tested algorithms (*right*), for synthetic datasets using label distributions D_4, D_5, D_6.

sure is what slows down the second algorithm, this is not the case since this operation is negligible compared to support computation. In fact, the *heuristic* algorithm performs less subgraph isomorphisms tests than *dw-closure*, eventhough *dw-closure* is the algorithm that computes the support of the least number of graphs. This surprising result can be explained as follows. Since a graph G is contained by a database graph only if its parent is, the number of subgraph isomorphism tests required to compute the support of G is bounded by the support of its parent. By selecting the parent of G as the graph least likely to be frequent, the *heuristic* algorithm thus tends to reduce the number of

subgraph isomorphism tests required for G.

5.3.2 Real-life data

In the second experiment, we evaluated our approach on a real-life dataset from the field of chemoinformatics. This dataset, which contains a set of 340 chemical compounds modelled as labeled graphs, was devised as a benchmark for the Predictive Toxicology Evaluation (PTE) challenge (Srinivasan et al., 1997). As we have done in the previous experiment, we tested four algorithms, *heuristic*, *random*, *prefix* and *dw-closure*, on the task of finding the frequent connected subgraphs of this dataset[5].

For this experiment, we have modified our *heuristic* algorithm to exploit some common characteristics of this type of data: the frequent subgraphs are mostly cycle-free connected graphs, whose vertices have a low degree[6]. We illustrate how this information was used in our algorithm with a small example. Consider the graph shown on the left side of Figure 5.5, that we will denote G. The values shown beside each vertex of G are the index (1,2,3,4 or 5) and the label (a or b) of this vertex. Since the edges of G are non-disconnecting, they must either be part of a cycle (edges $(3,4)$, $(3,5)$, $(4,5)$ in this example) or have a vertex incident to no other edge (here $(1,3)$, $(2,4)$). If we remove an edge contained in a cycle, such as $(3,4)$, this cycle will not be present in the resulting graph. However, if we remove an edge of the second type, e.g. $(1,3)$, the graph we obtain will keep all its cycles. Because cycle-free graphs are more likely to be frequent, and since we want as parent $p(G)$ of G one of its least frequent subgraphs, $p(G)$ should thus be obtained by removing from G an edge incident to a vertex of lowest degree. This approach also has the benefit that the parent graphs will have vertices of higher degree, and thus, will have less chance of being frequent. In this example, vertices 1 and 2 are

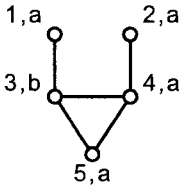
[5]The edge labels of the dataset were discarded for this experiment.

[6]The degree of a vertex is the number of edges incident to this vertex.

the only ones with the lowest degree of 1. Thus the parent of G should be obtained by removing an edge incident to one of these vertices, i.e. either $(1,3)$ or $(2,4)$. To make our *heuristic* algorithm even more efficient, we also used background information on the vertex labels. Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be two non-disconnecting edges of G (suppose, without loss of generality, that $deg(u_1) \leq deg(v_1)$ and $deg(u_2) \leq deg(v_2)$), and denote N_1, N_2, once more, the number of edges of the dataset that have incident vertices with the same labels as e_1 and e_2. The heuristic function h used in our algorithm is such that $h(G - \{e_1\}) < h(G - \{e_2\})$ if either one of the following cases is true:

1. $deg(u_1) < deg(u_2)$
2. $deg(u_1) = deg(u_2)$ and $N_1 > N_2$
3. $deg(u_1) = deg(u_2)$ and $N_1 = N_2$ and $deg(v_1) < deg(v_2)$.

Suppose that, in our example, the number of database edges whose incident vertices have the corresponding pair of labels is as shown on the right-side of Figure 5.5. In this case, since the label pair (a, a) of edge $(2, 4)$ is more frequent than the label pair (a, b) of edge $(1, 3)$ (300 occurrences versus 200) the parent of G will be the graph $G - \{(2, 4)\}$.



Label pair	Occurrences
(a, a)	300
(a, b)	200
(b, b)	150

Figure 5.5 A labeled graph (*left*) and the number of database edges having vertices with the given labels (*right*).

Figure 5.6 summarizes the results of this experiment. On the top left are shown the percentages of visited graphs that are frequent and downward closed, as found by *dw-closure*, for decreasing values of support threshold. On the top right are presented the percentage of graphs, visited by all four algorithms, that are cycle-free. The graphic

located at the bottom left gives the runtimes of the algorithms, again for decreasing support threshold values. Finally, at the bottom right are shown the number of subgraph isomorphism tests performed by all four algorithms. As in the first experiment, we notice that, while the ratio of frequent graphs is fairly constant for all support threshold values, the ratio of downward closed graphs decreases as the support threshold lowers. However, due to the particular nature of the data, the ratio of downward closed graphs is much lower than it was for the synthetic datasets. Because of this, we can thus expect a greater performance increase for *dw-closure* and *heuristic*, compared to what we observed for synthetic data. Also, from the ratio of visited graphs that are cycle-free, we observe that both the *prefix* and *dw-closure* algorithms are well adapted to limit the search to this type of graph (ratios ranging from 88% to 85%). Moreover, comparing the *random* and *heuristic* algorithms, we can see that the heuristic function helps in avoiding graphs with cycles. Thus, while the *random* and *heuristic* algorithms have similar ratios for a support threshold of 24% (respectively 85% and 88%), the *heuristic* algorithm does a much better job at avoiding such graphs (ratio of 92% versus 65% for *random*), for a support threshold of 3%. Furthermore, as we expected, *dw-closure* and *heuristic* clearly outperform *prefix* and *random*. Thus, *heuristic* is 8 to 15 times faster than *prefix* for all support threshold values. Finally, as we did for the first experiment, we notice that *heuristic* is somewhat faster than *dw-closure* (up to twice faster for a support threshold of 3%), although, this time, the number of subgraphs isomorphism tests performed by the two algorithms is very comparable. Since the dataset only contains 340 graphs (versus 10000 for the synthetic datasets), the downward closure check of the *dw-closure* algorithm accounts for a greater portion of the algorithm's runtime, and could, therefore, explain this performance gap.

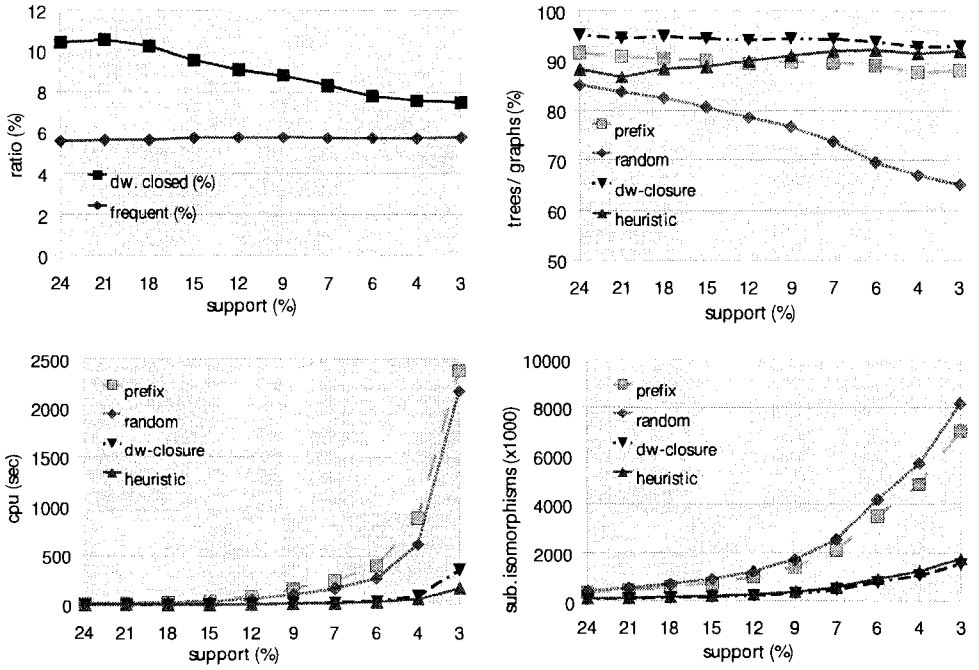


Figure 5.6 Ratio of downward closed and frequent visited graphs (*top left*), ratio of visited graphs that are trees (*top right*), runtimes (*bottom left*) and number of subgraphs isomorphism tests performed (*bottom right*), for the tested algorithms on the Predictive Toxicology Evaluation (PTE) dataset.

5.4 Conclusion

We have presented, in this paper, a simple and general strategy that improves the task of finding the frequent patterns of a database containing structured data. This novel approach uses background information on the frequent patterns, in the form of a heuristic function that transforms the search space into a rooted tree such that the parent of a pattern is as unlikely as possible to be frequent. This allows to avoid exploring a great number of infrequent patterns and, consequently, to reduce the number of costly support computations. To evaluate our approach, we have tested it on a well-known specialization of the frequent pattern mining problem, the frequent subgraph mining problem, where the task is to find the connected graphs for which the support in the database is greater than a given threshold. These tests were carried out on two types of data: syn-

thetic datasets that have a skewed distribution of vertex labels, and a real-life dataset from the Predictive Toxicology Evaluation (PTE) challenge. The results obtained for these tests have shown our approach to be more efficient than a specialized technique using depth-first search (DFS) coding, and to be as powerful as the more complex strategy of testing downward closure.

CHAPTER 6

AUTOMATED GENERATION OF CONJECTURES ON FORBIDDEN SUBGRAPH CHARACTERIZATION

6.1 Introduction

Traditionally, the process of discovering new knowledge in graph theory was carried out by mathematicians, with little assistance from computers. Yet, in recent years, mathematicians in that field have turned to computers to find some very important results. A famous illustration of this is the proof to the four color conjecture, which was done in large part by computers (Appel et al., 1977; Robertson et al., 1997). Since then, computers have played an increasing role in the discovery of new knowledge in graph theory, and many tools have been proposed for this task. One of the first computer programs for this purpose is GRAFFITI, developed by Fajtlowicz (Fajtlowicz, 1988a), which has generated over a thousand conjectures as algebraic equations involving graph invariants. Another more recent and equally prolific tool to generate conjectures involving graph invariants is AUTOGRAPHIX (AGX), proposed by Caporossi and Hansen (Caporossi et Hansen, 2000). This last program, which applies the Variable Neighborhood Search metaheuristic (Mladenovic et Hansen, 1997) to find extremal graphs, can also be used to find graph satisfying various constraints, to find structural conjectures, to refute conjectures and to suggest proofs.

Although automating the generation of conjectures has been the aim of many works, almost all these focused on generating conjectures in the form of relations on graph invariants. Yet, as recently suggested by Hansen et al. in (Hansen et al., 2005), there are many interesting results in graph theory that take a different form. One of them, known

as *forbidden subgraph characterization* (FSC), describes a class of graphs in terms of the subgraphs that these graphs are not allowed to have. A well known FSC, due to Chudnovsky et al. (Chudnovsky et al., 2006), characterizes perfect graphs as the graphs which do not have as induced subgraph any odd cycle containing five or more vertices, or its complement. Another important FSC, due to Beineke (Beineke, 1970), characterizes line graphs using nine forbidden graphs, shown in Figure 6.1.

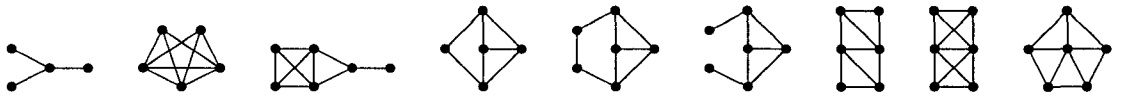


Figure 6.1 A forbidden subgraph characterization of line graphs.

In this paper, we present some new methods to automatically generate conjectures on FSCs. The rest of the paper is structured as follows. We first introduce some preliminary concepts that will help to understand the rest of the paper. We then describe our methods, by considering three problems: finding sufficient conditions for an FSC, finding necessary conditions for an FSC, and finding actual FSCs. We then show how these methods can be used in practice to generate conjectures, and illustrate this by reproducing some known results, as well as generating new ones. Finally, we end this paper with a short summary of our work.

6.2 Preliminary concepts and definitions

Let $\mathcal{G}$ be the set containing all graphs. A class of graphs $\mathcal{C} \subseteq \mathcal{G}$ is a possibly infinite set of graphs that share a common property. Let $\mathcal{H}$ be a set of graphs, we say that a graph G is $\mathcal{H}$-free if there is no graph of $\mathcal{H}$ isomorphic to one of its induced subgraphs, and write $\mathcal{G}_{\mathcal{H}}$ the set of all such graphs. Using this terminology, an FSC of $\mathcal{C}$ is a set of graphs $\mathcal{H}$ such that $\mathcal{G}_{\mathcal{H}} = \mathcal{C}$. As we will see, not every class of graphs has FSCs. For

classes that do not have an FSC, we are often interested in finding some weaker rules allowing to partially characterize the graphs of these classes. These rules come in two forms: *sufficient conditions* (SFSC) and *necessary* conditions (NFSC). Let $\mathcal{C}$ be the class of graphs to characterize and $\mathcal{H}$ be a set of forbidden subgraphs. Sufficient conditions can be expressed as follows: if a graph G is $\mathcal{H}$-free, then it is part of $\mathcal{C}$. Thus, a sufficient condition is such that $\mathcal{G}_{\mathcal{H}} \subseteq \mathcal{C}$. However, sufficient conditions do not fully describe $\mathcal{C}$. Indeed, if G is not $\mathcal{H}$-free, we cannot use this type of condition to determine if G is in $\mathcal{C}$ or not. On the other hand, necessary conditions can be expressed as follows: if a graph G is in $\mathcal{C}$ then it is $\mathcal{H}$-free. This implies that $\mathcal{G}_{\mathcal{H}} \supseteq \mathcal{C}$. Again, necessary conditions offer a partial description of $\mathcal{C}$: if G is not in $\mathcal{C}$ then it can either be $\mathcal{H}$-free or not.

Let G be a graph, and $W = \{v_1, v_2, \ldots, v_q\}$ be a subset of $V(G)$. We write $G[W]$ or, when the context is clear, $\langle v_1, v_2, \ldots, v_q \rangle$ the subgraph of G induced by W. Furthermore, let H be another graph, we write $G \simeq H$ when G is isomorphic to H, and $H \subseteq G$ when H is isomorphic to an induced subgraph of G. The following elementary properties will be used later on to prove more complex results.

Property 1. *Let G_1, G_2, G_3 be three graphs, and $\mathcal{H}_1$, $\mathcal{H}_2$ be two sets of forbidden subgraphs.*

(a). If $G_1 \subseteq G_2$ and $G_2 \subseteq G_3$ then $G_1 \subseteq G_3$.

(b). If $G_1 \not\subseteq G_2$ and $G_3 \subseteq G_2$ then $G_1 \not\subseteq G_3$.

(c). If $G_1 \not\subseteq G_2$ and $G_1 \subseteq G_3$ then $G_3 \not\subseteq G_2$.

(d). If $G_1 \subseteq G_2$ then $\mathcal{G}_{\{G_1\}} \subseteq \mathcal{G}_{\{G_2\}}$.

(e). $\mathcal{G}_{\mathcal{H}_1 \cup \mathcal{H}_2} = \mathcal{G}_{\mathcal{H}_1} \cap \mathcal{G}_{\mathcal{H}_2}$.

(f). If $\mathcal{H}_1 \subseteq \mathcal{H}_2$ then $\mathcal{G}_{\mathcal{H}_1} \supseteq \mathcal{G}_{\mathcal{H}_2}$.

(g). If $G_1 \subseteq G_2$ then $\mathcal{G}_{\{G_1\}} = \mathcal{G}_{\{G_1, G_2\}}$.

To lighten the presentation, we will, in the rest of this paper, refer to induced subgraphs simply as subgraphs.

6.3 Sufficient conditions

Formally, we write an SFSC as G is $\mathcal{H}$-free $\Rightarrow$ $G \in \mathcal{C}$. This expression is logically equivalent to $G \in \overline{\mathcal{C}} \Rightarrow \exists H \in \mathcal{H}$ s.t. $H \subseteq G$, where $\overline{\mathcal{C}} = \mathcal{G} \setminus \mathcal{C}$ is the complement of the graph class $\mathcal{C}$. The task of finding an SFSC can thus be defined as follows: find a set of graphs $\mathcal{H}$ such that

$$\forall G \in \overline{\mathcal{C}},\ \exists H \in \mathcal{H} \text{ s.t. } H \subseteq G.$$

While a graph class $\mathcal{C}$ can have many SFSCs, these may not be equally useful. For instance, $\mathcal{H} = \overline{\mathcal{C}}$ is an SFSC of $\mathcal{C}$, but is as complex as the class $\mathcal{C}$ itself. Moreover, let H be the graph composed of a single vertex, $\mathcal{H} = \{H\}$ is an SFSC of $\mathcal{C}$ since H is a subgraph of all graphs of $\overline{\mathcal{C}}$. However, $\mathcal{H}$ offers no real information on $\mathcal{C}$, since each graph of $\mathcal{C}$ also has H as subgraph, i.e. $\mathcal{C} \cap \mathcal{G}_{\mathcal{H}} = \emptyset$. To help us find useful SFSCs, we need to introduce two measures: *complexness* and *tightness*. The complexness of an SFSC roughly evaluates the amount of information needed by this SFSC to describe $\mathcal{C}$. Let $\mathcal{H}$, $\mathcal{H}'$ be two SFSCs of $\mathcal{C}$. We say that $\mathcal{H}$ is more complex than $\mathcal{H}'$ if $|\mathcal{H}| > |\mathcal{H}'|$, or in the case where $\mathcal{H} = \{H\}$ and $\mathcal{H}' = \{H'\}$, if $|V(H)| > |V(H')|$. On the other hand, the tightness of an SFSC evaluates how well it describes $\mathcal{C}$. We say that an SFSC $\mathcal{H}$ is tighter than another SFSC $\mathcal{H}'$ if $\mathcal{G}_{\mathcal{H}'} \subseteq \mathcal{G}_{\mathcal{H}}$. Thus, an SFSC $\mathcal{H}$ is maximally tight if $\mathcal{G}_{\mathcal{H}} = \mathcal{C}$. We see that the concepts of complexness and tightness are related: a tighter SFSC can generally be obtained by increasing its complexness. Hence, the measure of an SFSC's usefulness is a compromise between the minimization of its complexness and the maximization of its tightness.

From the above definitions, we can impose an additional constraint on the selection of an

SFSC $\mathcal{H}$. Suppose $\mathcal{H}$ contains two graphs H and H' such that $H' \subset H$. From Property 1.(g), we know that $\mathcal{H}' = \mathcal{H} \setminus \{H\}$ is an SFSC of $\mathcal{C}$ as tight as $\mathcal{H}$. However, $\mathcal{H}'$ is less complex than $\mathcal{H}$ since it contains one less graph. We can thus limit our search to the sets of graphs $\mathcal{H}$ such that

$$\forall H \in \mathcal{H}, \ \nexists H' \in \mathcal{H} \text{ s.t. } H' \subset H.$$

In the rest of the paper, we will call *minimal* a set $\mathcal{H}$ that satisfies the above property, and denote $\min(\mathcal{H}) \subseteq \mathcal{H}$ the subset of $\mathcal{H}$ that is minimal.

Proposition 1. *Let $\mathcal{C}$ be a class of graphs and $\mathcal{H}$ be a minimal SFSC of $\mathcal{C}$ with maximum tightness, then $\mathcal{H} = \min(\overline{\mathcal{C}})$.*

Proof. Let $\mathcal{H} = \min(\overline{\mathcal{C}})$. We first show that $\mathcal{H}$ is an SFSC of $\mathcal{C}$. Suppose it is not, then there is a graph $G \in \overline{\mathcal{C}} \setminus \mathcal{H}$ such that $\nexists H \in \mathcal{H}$, $H \subset G$. However, G would then be, by definition, in $\mathcal{H}$, which is a contradiction. We next show that $\mathcal{H}$ is the only SFSC with maximum tightness. Let $\mathcal{H}'$ be any minimal SFSC of $\mathcal{C}$. For each $H' \in \mathcal{H}'$, if $\nexists G \in \overline{\mathcal{C}}$ such that $H' \subseteq G$ then, clearly, the tightness of $\mathcal{H}'$ is not maximum since $\mathcal{H}' \setminus \{H'\}$ is an SFSC of greater tightness. Otherwise, if $H' \in \mathcal{C}$, let $\mathcal{B}$ denote the graphs of $\overline{\mathcal{C}}$ which have H' as subgraph. By definition, $\mathcal{H}'' = \mathcal{H}' \setminus \{H'\} \cup \mathcal{B}$ is an SFSC of $\mathcal{C}$. Furthermore, from Properties 1.(d) and 1.(e), we have that $\mathcal{G}_{\{H'\}} \subseteq \mathcal{G}_{\mathcal{B}}$ and that $\mathcal{G}_{\mathcal{H}'} \subseteq \mathcal{G}_{\mathcal{H}''}$. Finally, since $H' \notin \mathcal{G}_{\mathcal{H}'}$ and $H' \in \mathcal{G}_{\mathcal{H}''}$, we conclude that $\mathcal{G}_{\mathcal{H}'} \subset \mathcal{G}_{\mathcal{H}''}$ and therefore that $\mathcal{H}''$ is tighter than $\mathcal{H}'$. The remaining case is $H' \in \overline{\mathcal{C}}$. Suppose that $H' \notin \mathcal{H}$, then by definition there is a graph $H \in \mathcal{H}$ such that $H \subset H'$. Furthermore, since $\mathcal{H}'$ is an SFSC of $\mathcal{C}$, there must be a graph $H'' \in \mathcal{H}'$ such that $H'' \subseteq H \subset H'$, which contradicts the minimality of $\mathcal{H}'$. Therefore, we have that $H' \in \mathcal{H}$, and thus, either $\mathcal{H}'$ does not have maximum tightness or $\mathcal{H}' = \mathcal{H}$. □

6.3.1 Single graph SFSC

Before we tackle the general task of finding an SFSC having an arbitrary number of graphs, we first consider the simpler case of finding an SFSC of a graph class $\mathcal{C}$, containing a single graph. In this case, the problem can be formulated as finding a set $\mathcal{H} = \{H\}$ such that $\forall G \in \overline{\mathcal{C}},\ H \subseteq G$. Thus, H is a common subgraph of the graphs in $\overline{\mathcal{C}}$. Furthermore, let H, H' be two graphs such that $H \subset H'$. From Property 1.(*d*), we have $\mathcal{G}_{\{H\}} \subset \mathcal{G}_{\{H'\}}$ and therefore that $\{H'\}$ is tighter than $\{H\}$. Hence, if we want to maximize the tightness, we should find a common subgraph H which is maximal w.r.t. inclusion. This principle serves as the main idea of our first algorithm, detailed in Figure 6.2. Let $\mathcal{L}$ be a finite set of graphs and denote maxCS($\mathcal{L}$) the set of common subgraphs of $\mathcal{L}$, the single graph SFSC algorithm searches for a graph H_k which is contained in all graphs of $\overline{\mathcal{C}}$, in the following way. Starting with a graph H_0 chosen in $\overline{\mathcal{C}}$ and a set $\mathcal{L}$ of representative graphs of $\overline{\mathcal{C}}$ containing only H_0, the algorithm searches, at every iteration k, for a graph $G_k \in \overline{\mathcal{C}}$ which does not contain H_k. If such a graph exists, the algorithm then adds G_k to the representatives of $\overline{\mathcal{C}}$, i.e. $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$, and finds a new graph H_{k+1}, which is a maximal common subgraph of $\mathcal{L}_{k+1}$. Otherwise, if no such graph exists, we know that H_k is a subgraph of all graphs of $\overline{\mathcal{C}}$, and thus is an SFSC of $\mathcal{C}$.

Proposition 2. *The single graph SFSC algorithm, shown in Figure 6.2, produces, in a finite number of steps, a common subgraph of $\overline{\mathcal{C}}$ which is maximal w.r.t. inclusion.*

Proof. We first prove that the algorithm terminates in a finite number of steps. Consider any two graphs H_i, H_j such that $i < j$. Since H_j is a common subgraph of $\mathcal{L}_j$, and $G_i \in \mathcal{L}_j$, we have $H_j \subseteq G_i$ (*1*). Moreover, since $G_i \in \overline{\mathcal{C}} \cap \mathcal{G}_{\{H_i\}}$, we know that $H_i \not\subseteq G_i$ (*2*). Combining (*1*) and (*2*), we get $H_i \neq H_j$. Furthermore, since H_i and H_j are maximum common subgraphs of $\mathcal{L}_i$ and $\mathcal{L}_j$, and since $\mathcal{L}_i \subset \mathcal{L}_j$, we have that $|V(H_j)| \leq |V(H_i)|$. Finally, since there is a finite number of graphs of $|V(H_i)|$ or less

Single graph SFSC algorithm

Input: A graph class $\mathcal{C}$.
Output: A single graph SFSC $\mathcal{H}$ of $\mathcal{C}$.

Choose any H_0 in $\overline{\mathcal{C}}$;
Let $\mathcal{L}_0 := \{H_0\}$, and $k := 0$;

while $\exists G_k \in \overline{\mathcal{C}} \cap \mathcal{G}_{\{H_k\}}$ **do**
  Let $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$;
  Choose H_{k+1} in maxCS($\mathcal{L}_{k+1}$) ;
  Let $k := k + 1$;
return $\mathcal{H} = \{H_k\}$;

Figure 6.2 Algorithm to find an SFSC containing a single forbidden subgraph.

vertices, the algorithm will terminate in a finite number of steps. Next, we prove that the graph H_k returned by the algorithm is a common subgraph of $\overline{\mathcal{C}}$. Suppose it is not. Then, there exists a graph $G_k \in \overline{\mathcal{C}}$ that does not have H_k as subgraph. However, this is a contradiction since the algorithm terminates only when no such graph exists. Finally, we show that H_k is maximal w.r.t. inclusion. Suppose this is not the case. Then, there exists a common subgraph H' of $\overline{\mathcal{C}}$, such that $H_k \subset H'$. Since H' is a common subgraph of $\overline{\mathcal{C}}$, it is also a common subgraph of $\mathcal{L}_k \subseteq \overline{\mathcal{C}}$. However, this contradicts the fact that H_k is a maximum common subgraph of $\mathcal{L}_k$. □

6.3.2 Multiple graph SFSC

From Proposition 1, we know that if a graph H, obtained by the single graph SFSC algorithm, is not in $\overline{\mathcal{C}}$, then we can find a tighter SFSC containing more than one forbidden subgraph. Let M be the maximum number of graphs we want to have in the SFSC. If $|\min(\overline{\mathcal{C}})| \leq M$, the algorithm should return $\min(\overline{\mathcal{C}})$ since this SFSC has maximum tightness. Otherwise, we search for an SFSC of $\mathcal{C}$ in the following way. We first find a set $\mathcal{M}$ containing $M-1$ of the smallest graphs of $\min(\overline{\mathcal{C}})$. Since $\mathcal{M} = \min(\mathcal{M}) \subset \min(\overline{\mathcal{C}})$, by

definition, $\mathcal{M}$ is an SFSC of $\mathcal{G}_{\mathcal{M}} \supset \mathcal{C}$ that has maximum tightness. We then find a common subgraph H of $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{M}}$. Finally, since a graph of $\mathcal{M}$ may contain H, to minimize the complexness, we return the SFSC $\mathcal{H} = \min(\mathcal{M} \cup \{H\})$. Denote by $\text{findMin}(\mathcal{C}, L)$ the procedure that returns the set $\min(\mathcal{C})$, if $|\min(\mathcal{C})| \leq L$, or otherwise returns L of the smallest graphs of $\min(\mathcal{C})$. The multiple graph SFSC algorithm, shown in Figure 6.3, finds an SFSC for a graph class $\mathcal{C}$, containing at most M graphs.

Multiple graph SFSC algorithm

Input: A graph class $\mathcal{C}$ and an integer $M > 0$.
Output: An SFSC $\mathcal{H}$ of $\mathcal{C}$ containing at most M graphs.

Let $\mathcal{M} := \text{findMin}(\overline{\mathcal{C}}, M-1)$ and choose any H_0 in $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{M}}$;
Let $\mathcal{L}_0 := \{H_0\}$, and $k := 0$;
while $\exists G_k \in \overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{M} \cup \{H_k\}}$ **do**
  Let $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$;
  Choose H_{k+1} in $\text{maxCS}(\mathcal{L}_{k+1})$;
  Let $k := k + 1$;
return $\mathcal{H} = \min(\mathcal{M} \cup \{H_k\})$;

Procedure `findMin` $(\mathcal{C}, L)$

Input: A graph class $\mathcal{C}$ and an integer $L > 0$.
Output: At most L of the smallest graphs of $\min(\mathcal{C})$.

Let $\mathcal{M}_0 := \emptyset$ and $l := 0$;
while $\mathcal{C} \cap \mathcal{G}_{\mathcal{M}_l} \neq \emptyset$ *and* $l < L$ **do**
  Choose $H_l \in \mathcal{C} \cap \mathcal{G}_{\mathcal{M}_l}$ with minimum order ;
  Let $\mathcal{M}_{l+1} := \mathcal{M}_l \cup \{H_l\}$;
  Let $l := l + 1$;
return $\mathcal{M} = \mathcal{M}_l$;

Figure 6.3 Algorithm to find an SFSC containing at most M forbidden subgraphs.

Proposition 3. *The multiple graph SFSC algorithm of Figure 6.3 produces, in a finite number of steps, an SFSC of $\mathcal{C}$ containing at most M graphs.*

Proof. We first show that the algorithm terminates in a finite number of steps. Clearly,

the procedure that returns $\text{findMin}(\overline{\mathcal{C}}, M-1)$ will terminate in at most $M-1$ iterations. Furthermore, from Proposition 2, we know that finding the last graph can be done in a finite number of steps. Thus, the algorithm is finite. We next prove that the set $\mathcal{H} = \min(\mathcal{M} \cup \{H_k\})$ returned by the algorithm is an SFSC of at most M graphs. Consider once again the procedure findMin. Since we have $|\mathcal{M}_l| = l$, at every iteration l, this procedure will return a set $\mathcal{M}$ containing at most $M-1$ graphs. Furthermore, suppose that $\mathcal{M} \cup \{H_k\}$ is not an SFSC. Then, there exists a graph $G \in \overline{\mathcal{C}}$ that does not have H_k nor any graph of $\mathcal{M}_k$ as subgraph. However, this is a contradiction since the algorithm terminates only when no such graph exists. Thus, $\mathcal{M} \cup \{H_k\}$ is an SFSC of $|\mathcal{M}|+1 \leq M$ graphs. Finally, since $\min(\mathcal{M} \cup \{H_k\}) \subseteq \mathcal{M} \cup \{H_k\}$, we have, by definition, that $\mathcal{H}$ is an SFSC of at most M graphs. □

Proposition 4. *Let H be a forbidden subgraph obtained by the single graph SFSC algorithm, and $\mathcal{H} = \mathcal{M} \cup \{H_k\}$ be the multiple graph SFSC returned by the above algorithm. If $H \subset H_k$ then $\mathcal{H}$ is a tighter SFSC than $\{H\}$.*

Proof. Since $\{H\}$ is an SFSC and $\mathcal{M} \subseteq \overline{\mathcal{C}}$, we know that $H \subseteq H'$, for all $H' \in \mathcal{M}$, and, from Properties 1.(*d*) and 1.(*e*), that $\mathcal{G}_{\{H\}} \subseteq \mathcal{G}_{\mathcal{M}}$. Furthermore, since $H \subset H_k$, we have $\mathcal{G}_{\{H\}} \subset \mathcal{G}_{\{H_k\}}$, and thus that $\mathcal{G}_{\{H\}} \subset \mathcal{G}_{\mathcal{M}} \cap \mathcal{G}_{\{H_k\}} = \mathcal{G}_{\mathcal{H}}$. Therefore $\mathcal{H}$ is a tighter SFSC than $\{H\}$. □

6.4 Necessary conditions

Let $\mathcal{C}$ be a graph class, an NFSC is a set of graphs $\mathcal{H}$ such that all graphs in $\mathcal{C}$ are $\mathcal{H}$-free. Finding an NFSC thus amounts to finding $\mathcal{H}$ such that

$$\forall G \in \mathcal{C},\ \nexists H \in \mathcal{H} \text{ s.t. } H \subseteq G.$$

This implies that $\mathcal{H} \subseteq \overline{\mathcal{C}}$. As we did for SFSCs, we can use the criteria of complexness and tightness to guide our search of an NFSC. However, since an NFSC $\mathcal{H}$ is such that $\mathcal{G}_{\mathcal{H}} \supseteq \mathcal{C}$, the definition of tightness differs from the one given for sufficient conditions. Let $\mathcal{H}$, $\mathcal{H}'$ be two NFSCs, we say that $\mathcal{H}$ is tighter than $\mathcal{H}'$ if $\mathcal{G}_{\mathcal{H}} \subset \mathcal{G}_{\mathcal{H}'}$. As we have done for sufficient conditions, we first present a method that finds single graph NFSCs and then generalize this method to multiple graph NFSCs.

6.4.1 Single graph NFSC

The task of finding a single graph NFSC of a graph class $\mathcal{C}$ corresponds to finding a graph H such that, for all $G \in \mathcal{C}$, $H \not\subseteq G$. Thus, H is a "common non-subgraph" of the graphs in $\mathcal{C}$. Moreover, let H and H' be two graphs such that $H \subset H'$. From Property 1.(d), we have that $\mathcal{G}_{\{H\}} \subset \mathcal{G}_{\{H'\}}$ and, therefore, that $\{H\}$ is tighter than $\{H'\}$. Thus, to maximize the tightness, H should be a common non-subgraph which is minimal w.r.t. inclusion. Our single graph NFSC algorithm, detailed in Figure 6.4, is based on this idea. Let $\overline{\mathcal{G}}_{\mathcal{H}} = \mathcal{G} \setminus \mathcal{G}_{\mathcal{H}}$ be the set of graphs containing at least one graph of $\mathcal{H}$, and denote $\mathrm{minCNS}(\mathcal{L})$ the set of common non-subgraphs of a set $\mathcal{L}$, which have a minimum number of vertices. Starting with a graph H_0 that has a single vertex and an empty set $\mathcal{L}_0$ of representative graphs of $\mathcal{C}$, at each iteration k, we try to find a graph $G_k \in \mathcal{C}$ which contains H_k. If such a graph exists, then H_k is not an NFSC. We then improve H_k by adding G_k to the set of representatives of $\mathcal{C}$, i.e. $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$, and finding a graph H_{k+1} that is not a subgraph of any graph of $\mathcal{L}_{k+1}$. Finally, we repeat this process until no graph of $\mathcal{C}$ contains H_k, i.e. when $\mathcal{C} \cap \overline{\mathcal{G}}_{\{H_k\}} = \emptyset$, and return $\mathcal{H} = \{H_k\}$. In the case where $\mathcal{C}$ has no NFSC, we may always be able to find $G_k \in \mathcal{C}$ containing H_k, and thus, the algorithm may never stop. To avoid this problem, we impose a limit N on the number of vertices of our NFSC.

Proposition 5. *The single graph NFSC algorithm of Figure 6.4 returns, in a finite num-*

Single graph NFSC algorithm

Input: A graph class $\mathcal{C}$ and an integer $N > 0$.

Output: A single graph NFSC $\mathcal{H}$ of $\mathcal{C}$ having at most N vertices, or a graph of more than N vertices.

Let H_0 be a single vertex graph, $\mathcal{L}_0 := \emptyset$, and $k := 0$;

while $\exists G_k \in \mathcal{C} \cap \overline{\mathcal{G}}_{\{H_k\}}$ *and* $|V(H_k)| \leq N$ **do**

  Let $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$;

  Choose H_{k+1} in minCNS($\mathcal{L}_{k+1}$) ;

  Let $k := k + 1$;

return $\mathcal{H} = \{H_k\}$;

Figure 6.4 Algorithm to find an NFSC containing a single forbidden subgraph.

ber of steps, a common non-subgraph of $\mathcal{C}$ which is minimal w.r.t. inclusion and has at most N vertices, if one exists. Otherwise, the algorithm returns a graph that has more than N vertices.

Proof. We first prove that the algorithm terminates in a finite number of steps. Consider any two graphs H_i and H_j such that $i < j$. Since H_j is a common non-subgraph of $\mathcal{L}_j$, and $G_i \in \mathcal{L}_j$, we have $H_j \not\subseteq G_i$ (*1*). Moreover, since $G_i \in \mathcal{C} \cap \overline{\mathcal{G}}_{\{H_i\}}$, we know that $H_i \subseteq G_i$ (*2*). Combining (*1*) and (*2*), we get $H_i \neq H_j$. Furthermore, since H_i and H_j are minimum common non-subgraphs of $\mathcal{L}_i$ and $\mathcal{L}_j$, and since $\mathcal{L}_i \subset \mathcal{L}_j$, we have that $|V(H_j)| \geq |V(H_i)|$. Finally, since there is a finite number of graphs of N or less vertices, the algorithm will terminate in a finite number of steps. Next, we prove that, if the graph H_k returned by the algorithm is such that $|V(H_k)| \leq N$, then it is a common non-subgraph of $\mathcal{C}$. Suppose it is not. Then, there exists a graph $G_k \in \mathcal{C}$ that has H_k as subgraph. However, this is a contradiction since the algorithm terminates only when no such graph exists, or if $|V(H_k)| > N$. Finally, we show that H_k is minimal w.r.t. inclusion. Suppose this is not the case. Then, there exists a common non-subgraph H' of $\mathcal{C}$ such that $H' \subset H_k$. Since H' is a common non-subgraph of $\mathcal{C}$, it is also a common

non-subgraph of $\mathcal{L}_k \subseteq \mathcal{C}$. However, this contradicts the fact that H_k is a minimum common non-subgraph of $\mathcal{L}_k$. □

Although the algorithm guarantees to find in a finite number of steps a single graph NFSC, if one exists, we can accelerate its convergence using the fact that all NFSCs $\mathcal{H}$ are such that $\mathcal{H} \subseteq \overline{\mathcal{C}}$. Thus, instead of starting with a graph H_0 composed of a single vertex, we can start with H_0 as the smallest graph of $\overline{\mathcal{C}}$. Furthermore, let $\text{minCNS}(\mathcal{L}, \mathcal{C})$ be the set containing the smallest graph of a class $\mathcal{C}$ that are not contained in any graph of $\mathcal{L}$, we can choose H_{k+1} in $\text{minCNS}(\mathcal{L}, \overline{\mathcal{C}})$ instead of $\text{minCNS}(\mathcal{L})$.

6.4.2 Multiple graph NFSC

As it was the case for sufficient conditions, we can sometimes improve the tightness of an NFSC by increasing its complexness.

Proposition 6. *Let $\mathcal{C}$ be a given graph class and let $\mathcal{N}$ be the set of single graph NFSCs, i.e. $\mathcal{N} = \{H \in \overline{\mathcal{C}} \mid \nexists G \in \mathcal{C} \text{ s.t. } H \subset G\}$. If $\mathcal{H}$ is a minimal NFSC of maximum tightness then $\mathcal{H} = \min(\mathcal{N})$.*

Proof. Let $\mathcal{H} = \min(\mathcal{N})$. We first show that $\mathcal{H}$ is an NFSC of $\mathcal{C}$. Suppose it is not, then there is a graph $G \in \mathcal{C} \setminus \mathcal{H}$ such that $\exists H \in \mathcal{H}$, $H \subset G$, which contradicts the definition of $\mathcal{H}$. We next show that $\mathcal{H}$ is the only NFSC with maximum tightness. Let $\mathcal{H}'$ be any minimal NFSC of $\mathcal{C}$, and let H' be any graph of $\mathcal{H}'$. By definition, we know that $H' \in \mathcal{N}$. Suppose that H' is not in $\mathcal{H}$, since $\mathcal{H} = \min(\mathcal{N})$, there is a graph $H \in \mathcal{H}$ such that $H \subset H'$. Furthermore, from Properties 1.(*d*) and 1.(*e*), we have that $\mathcal{G}_{\{H\}} \subset \mathcal{G}_{\{H'\}}$, and, thus, that the NFSC $\mathcal{H}''$ produced by substituting H' for H in $\mathcal{H}'$, i.e. $\mathcal{H}'' = (\mathcal{H}' \setminus \{H'\}) \cup \{H\}$, is tighter than $\mathcal{H}'$. Therefore, $\mathcal{H}'$ is maximally tight only if $\mathcal{H}' \subseteq \mathcal{H}$. Suppose finally that $\mathcal{H}' \subset \mathcal{H}$. Then, by Property 1.(*f*), we have that $\mathcal{G}_{\mathcal{H}'} \supset \mathcal{G}_{\mathcal{H}}$,

and, thus, that $\mathcal{H}$ is tighter than $\mathcal{H}'$. Consequently, either $\mathcal{H}'$ does not have maximum tightness or $\mathcal{H}' = \mathcal{H}$. □

Following the above proposition, the NFSC of a class $\mathcal{C}$ with maximum tightness $\mathcal{H}$ is simply the minimal set containing all individual graphs that are, by themselves, NFSCs. To find $\mathcal{H}$, we extend the previous algorithm such that it finds all the single graph NFSCs, as shown in Figure 6.5. We start with an empty set $\mathcal{M}_0$ of forbidden subgraphs, an empty set $\mathcal{L}_0$ of representative graphs of class $\mathcal{C}$, and a graph H_0 composed of a single vertex. We can also select H_0 as one of the smallest graph of $\overline{\mathcal{C}}$, to speed-up the convergence. Then, at each iteration k, we look for a graph $G_k \in \mathcal{C}$ that contains H_k. If such a graph exists, we add it to the set of representatives of $\mathcal{C}$, i.e. we set $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$. Otherwise, H_k is a single graph NFSC and we add it to the set of forbidden subgraphs, i.e. we set $\mathcal{M}_{k+1} := \mathcal{M}_k \cup \{H_k\}$. Then, we find the smallest graph H_{k+1} that is not contained in any graph of $\mathcal{L}_{k+1}$, nor contains any graph of $\mathcal{M}_{k+1}$, i.e. $H_{k+1} \in \text{minCNS}(\mathcal{L}_{k+1}, \mathcal{G}_{\mathcal{M}_{k+1}})$. Again, if we wish to speed-up the convergence of the algorithm, we can restrict our search to $\overline{\mathcal{C}}$. We repeat this process until the order of H_k exceeds a given limit N, and return $\mathcal{H} = \mathcal{M}_k$.

Proposition 7. *The multiple graph NFSC algorithm of Figure 6.5 produces, in a finite number of steps, a set $\mathcal{H}$ containing all single graph NFSCs of at most N vertices.*

Proof. We first prove that the algorithm terminates in a finite number of steps. At every iteration k, only one of the following three conditions is satisfied: *(1)* $|V(H_k)| > N$, *(2)* $\exists G_k \in \mathcal{C} \cap \overline{\mathcal{G}}_{\{H_k\}}$, or *(3)* $\nexists G_k \in \mathcal{C} \cap \overline{\mathcal{G}}_{\{H_k\}}$. If the condition *(2)* is verified, then, since $H_k \subseteq G_k$ and $H_{k+1} \not\subseteq G_k$, we have that $H_k \neq H_{k+1}$. Moreover, because H_k and H_{k+1} are minimum common non-subgraphs of $\mathcal{L}_k$ and $\mathcal{L}_{k+1}$, and since $\mathcal{L}_k \subset \mathcal{L}_{k+1}$, we have that $|V(H_{k+1})| \geq |V(H_k)|$. However, if condition *(3)* is verified, then we have $H_k \in \mathcal{M}_{k+1}$ and $H_{k+1} \in \mathcal{G}_{\mathcal{M}_{k+1}}$, and therefore $H_{k+1} \neq H_k$. Also, because H_k and H_{k+1} are minimum common non-subgraphs of $\mathcal{L}_k$ and $\mathcal{L}_{k+1}$, and since $\mathcal{L}_k = \mathcal{L}_{k+1}$, we have

Multiple graph NFSC algorithm

Input: A graph class $\mathcal{C}$ and an integer $N > 0$.
Output: A set $\mathcal{H}$ containing all single graph NFSCs of at most N vertices.

Let H_0 be the graph containing a single vertex ;
Let $\mathcal{M}_0 := \emptyset$, $\mathcal{L}_0 := \emptyset$, and $k := 0$;

while $|V(H_k)| \leq N$ **do**
  if $\exists G_k \in \mathcal{C} \cap \overline{\mathcal{G}}_{\{H_k\}}$ **then** let $\mathcal{L}_{k+1} := \mathcal{L}_k \cup \{G_k\}$ and $\mathcal{M}_{k+1} := \mathcal{M}_k$;
  else let $\mathcal{L}_{k+1} := \mathcal{L}_k$ and $\mathcal{M}_{k+1} := \mathcal{M}_k \cup \{H_k\}$;
  Choose H_{k+1} in minCNS$(\mathcal{L}_{k+1}, \mathcal{G}_{\mathcal{M}_{k+1}})$;
  Let $k := k + 1$;
return $\mathcal{H} = \mathcal{M}_k$;

Figure 6.5 Algorithm to find a set containing all single graphe NFSCs of at most N vertices.

$|V(H_{k+1})| \geq |V(H_k)|$. Finally, since there is a finite number of graphs with N or less vertices, condition *(1)* will eventually be verified and the algorithm will terminate.

We next show that the algorithm is sound and complete. Since a graph H is only added to $\mathcal{H}$ if $\mathcal{C} \cap \overline{\mathcal{G}}_{\{H\}} = \emptyset$, we know that $\mathcal{C} \subseteq \mathcal{G}_{\{H\}}$ and thus that $\{H\}$ is an NFSC of $\mathcal{C}$. Furthermore, suppose there are two graphs $H_i, H_j \in \mathcal{H}$ such that $H_i \subset H_j$. Since, $|V(H_i)| < |V(H_j)|$, we know that $i < j$. Moreover, since $H_i \in \mathcal{M}_j$ and $H_j \in \mathcal{G}_{\mathcal{M}_j}$, we know that $H_i \not\subseteq H_j$ which is a contradiction. Therefore, the set $\mathcal{H}$ is minimal and the algorithm is sound. Moreover, suppose there exists a NFSC $\{H\}$ with $V(H) \leq N$, that is not found by the algorithm. By definition, we know that $H \in \text{minCNS}(\mathcal{C}, \mathcal{G}_{\mathcal{H}})$. Also, because minCNS finds the common non-subgraphs of minimum order, there must be an iteration k such that $H \notin \text{minCNS}(\mathcal{L}_k, \mathcal{G}_{\mathcal{M}_k})$. However, since $\mathcal{L}_k \subseteq \mathcal{C}$ and $\mathcal{M}_k \subseteq \mathcal{H}$, for every iteration k, we have $\text{minCNS}(\mathcal{C}, \mathcal{G}_{\mathcal{H}}) \subseteq \text{minCNS}(\mathcal{L}_k, \mathcal{G}_{\mathcal{M}_k})$, and thus $H \notin \text{minCNS}(\mathcal{C}, \mathcal{G}_{\mathcal{H}})$, which contradicts our initial supposition. Therefore, the algorithm is complete. □

6.5 Necessary and sufficient conditions

Before we present a method that finds FSCs, we need to answer some important questions. Firstly, we want to know under what conditions does a graph class have an FSC. The following proposition gives a well known result on the existence of an FSC for a given class of graphs, see e.g. (Greenwell et al., 1973).

Proposition 8. *A graph class $\mathcal{C}$ has an FSC if and only if it is hereditary.*

Proof. Let $\mathcal{H}$ be an FSC of $\mathcal{C}$. For any $G \in \mathcal{C}$, we know that G contains no graph of $\mathcal{H}$. Furthermore, for any $G' \subset G$, from Property 1.(b), we have that G' does not contain any graph of $\mathcal{H}$. Therefore, $G' \in \mathcal{G}_{\mathcal{H}} = \mathcal{C}$ and $\mathcal{C}$ is hereditary. Suppose conversely that $\mathcal{C}$ is an hereditary class of graphs, and let G be a graph of $\mathcal{C}$. Since every subgraph of G belongs to $\mathcal{C}$, we know that no subgraph of G belongs to $\overline{\mathcal{C}}$, and thus that $G \in \mathcal{G}_{\overline{\mathcal{C}}}$. Therefore, we have that $\mathcal{C} \subseteq \mathcal{G}_{\overline{\mathcal{C}}}$ (*1*). Moreover, let H be any graph of $\mathcal{G}_{\overline{\mathcal{C}}}$. We know that $H \notin \overline{\mathcal{C}}$, and therefore that $H \in \mathcal{C}$. Consequently, we have that $\mathcal{G}_{\overline{\mathcal{C}}} \subseteq \mathcal{C}$ (*2*). Combining (*1*) and (*2*), we have $\mathcal{C} = \mathcal{G}_{\overline{\mathcal{C}}}$, and thus, $\overline{\mathcal{C}}$ is an FSC of $\mathcal{C}$. □

The next question is equally important: *does a graph class $\mathcal{C}$ have a unique FSC and, if so, what is it ?* This question is answered by the following propositions.

Proposition 9. *Let $\mathcal{H}$ and $\mathcal{H}'$ be two minimal forbidden subgraph characterizations, then $\mathcal{G}_{\mathcal{H}} = \mathcal{G}_{\mathcal{H}'}$ if and only if $\mathcal{H} = \mathcal{H}'$.*

Proof. Since $\mathcal{H} = \mathcal{H}' \Rightarrow \mathcal{G}_{\mathcal{H}} = \mathcal{G}_{\mathcal{H}'}$ is trivial, we only show that $\mathcal{G}_{\mathcal{H}} = \mathcal{G}_{\mathcal{H}'} \Rightarrow \mathcal{H} = \mathcal{H}'$. Suppose that $\mathcal{H} \neq \mathcal{H}'$, then there exists $H \in \mathcal{H}$ such that $H \notin \mathcal{H}'$, or $H' \in \mathcal{H}'$ such that $H' \notin \mathcal{H}$. Without loss of generality, suppose the first case in true. If there is no $H' \in \mathcal{H}'$ such that $H' \subset H$ then, for all $H'' \in \mathcal{H}'$, we know that $H \notin \mathcal{G}_{\{H\}}$ and that $H \in \mathcal{G}_{\{H''\}}$. Finally, from Property 1.(e), we have that $H \notin \mathcal{G}_{\mathcal{H}}$ and $H \in \mathcal{G}_{\mathcal{H}'}$, and

therefore that $\mathcal{G}_{\mathcal{H}} \neq \mathcal{G}_{\mathcal{H}'}$. Otherwise, let H' be a graph in $\mathcal{H}'$ such that $H' \subset H$. There is no $H'' \in \mathcal{H}$ such that $H'' \subseteq H'$, otherwise we would have $H'' \subset H$ which contradicts the minimality of $\mathcal{H}$. Therefore, we have that $H' \notin \mathcal{G}_{\{H'\}}$ and that $H' \in \mathcal{G}_{\{H''\}}$, for all $H'' \in \mathcal{H}$. Finally, from Property 1.(e), we have that $H' \notin \mathcal{G}_{\mathcal{H}'}$ and $H' \in \mathcal{G}_{\mathcal{H}}$ and therefore that $\mathcal{G}_{\mathcal{H}} \neq \mathcal{G}_{\mathcal{H}'}$. □

Proposition 10. *Let $\mathcal{C}$ be a class of graphs. If $\mathcal{C}$ has an FSC, then the unique minimal FSC of $\mathcal{C}$ is* $\min(\overline{\mathcal{C}})$.

Proof. If $\mathcal{C}$ has an FSC, we know from Proposition 8 that it is hereditary and that $\overline{\mathcal{C}}$ is an FSC of $\mathcal{C}$. Moreover, by definition, we know that $\min(\overline{\mathcal{C}})$ is also an FSC of $\mathcal{C}$ that is minimal. Finally, we know from Proposition 9 that $\mathcal{C}$ has a single minimal characterization, which can only be $\min(\overline{\mathcal{C}})$. □

Consider an hereditary class of graphs $\mathcal{C}$ and let G be a graph in $\overline{\mathcal{C}}$. Furthermore, let $G' = G - \{v\}$ be a graph produced by removing from G any vertex $v \in V(G)$ and its incidents edges. If $G' \in \mathcal{C}$, since $\mathcal{C}$ is hereditary, all subgraphs of G' are also in $\mathcal{C}$. Therefore, we know that a graph G is minimal in $\overline{\mathcal{C}}$ if all its subgraph produced by removing a vertex are in $\mathcal{C}$. This is the main principle behind our algorithm to find FSCs, shown in Figure 6.6. At each iteration k, we find a graph $G_k \in \overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{M}_k}$, where $\mathcal{M}_0 = \emptyset$. Then, we search for a vertex v of G_k such that $G_k - \{v\}$, is still in $\overline{\mathcal{C}}$. If no such vertex exists, then G_k is minimal and we add it to $\mathcal{M}_k$. Otherwise, we remove v from G_k and look for another vertex with the same properties. We repeat this process until $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{M}_k} = \emptyset$, and return the set $\mathcal{H} = \mathcal{M}_k$. However, since an FSC can have an infinite number of graphs, we need to limit the size of the FSC to guarantee that the algorithm terminates in a finite number of steps. This limit is given as the parameter M of the algorithm.

Proposition 11. *The FSC algorithm of Figure 6.6 returns, in a finite number of steps, the set $\mathcal{H} = \min(\overline{\mathcal{C}})$, if $|\min(\overline{\mathcal{C}})| \leq M$, or a set $\mathcal{H}$ containing M graphs of* $\min(\overline{\mathcal{C}})$.

FSC algorithm

Input: A graph class $\mathcal{C}$ and an integer $M > 0$.
Output: An FSC $\mathcal{H}$ of $\mathcal{C}$ having at most M graphs, or a set containing more than M graphs.

Let $\mathcal{M}_0 := \emptyset$ and $k := 0$;
while $\exists G_k \in \overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{M}_k}$ *and* $k \leq M$ **do**
  while $\exists v \in V(G_k)$ *s.t.* $G_k - \{v\} \in \overline{\mathcal{C}}$ **do** $G_k := G_k - \{v\}$;
  Let $\mathcal{M}_{k+1} := \mathcal{M}_k \cup \{G_k\}$;
  Let $k := k + 1$;
return $\mathcal{H} = \mathcal{M}_k$;

Figure 6.6 Algorithm to find an FSC of at most M forbidden subgraphs.

Proof. Let G_i and G_j be two graphs, such that $i < j$. Since $G_i \in \mathcal{M}_j$ and $G_j \in \mathcal{G}_{\mathcal{M}_j}$, we have that $G_i \not\subseteq G_j$ and therefore that $G_i \neq G_j$. Moreover, since, at every iteration k, we have $|\mathcal{M}_k| = k$, the algorithm will terminate in at most M steps. Furthermore, let G_k be any graph of $\mathcal{H}$. Suppose that G_k is not minimal, then there is a vertex $v \in V(G_k)$ such that $G_k - \{v\} \in \overline{\mathcal{C}}$, which is a contradiction since G_k is added to $\mathcal{H}$ only if no such vertex exists. Therefore, $\mathcal{H} \subseteq \min(\overline{\mathcal{C}})$. Finally, we know that the algorithm terminates only when $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{H}} = \emptyset$ or $|\mathcal{H}| = M$. In the first case, we know that $\mathcal{H}$ is an FSC of $\mathcal{C}$, and since it is minimal, we know from Proposition 9 that it is unique, and thus $\mathcal{H} = \min(\overline{\mathcal{C}})$. Otherwise, $\mathcal{H}$ contains M graphs of $\min(\overline{\mathcal{C}})$. □

6.6 Automated conjecture generation

In the last sections, we have presented some exact methods to find FSCs of a class of graphs $\mathcal{C}$, or conditions to have an FSC. Although these methods guarantee to find FSCs, SFSCs or NFSCs of $\mathcal{C}$, if they exist, this might be impossible in practice. Thus, in many of our algorithms, we need to determine whether or not the sets $\mathcal{C} \cap \overline{\mathcal{G}}_{\mathcal{H}}$ and $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{H}}$

are empty, where $\mathcal{H}$ is a non-empty set of graphs. Since $\mathcal{C}$ and $\overline{\mathcal{C}}$ can be infinite, this may very well be an undecidable problem. To overcome this problem, we impose some restrictions on the graphs considered by our algorithms. Let $\mathcal{G}^n$ denote the graphs of $\mathcal{G}$ that have at most n vertices. We limit the graphs considered by our algorithms to $\mathcal{G}^n$, and use $\mathcal{G}^n \cap \mathcal{C}$ and $\mathcal{G}^n \cap \overline{\mathcal{C}}$ instead of $\mathcal{C}$ and $\overline{\mathcal{C}}$. Consequently, the guarantee that the results obtained by our algorithm are FSCs or conditions to have an FSC no longer holds. However, we can still use these algorithms to generate conjectures based on the hypothesis that if these results are true for graphs of n or less vertices, then they must be true for all graphs.

Limiting the order of graphs considered by our algorithms makes finding graphs of a given class practicable. Yet, finding these graphs is still a difficult task. Let $\mathcal{C}$ be a given class of graphs, and n be a given integer, we are interested in finding a graph of $\mathcal{C}$ that has at most n vertices. In the next sections, we present two approaches for this task: an enumerative approach and a heuristic one.

6.6.1 Enumerative approach

In the enumerative approach, we generate one by one the graphs of $\mathcal{G}^n$ until we find a graph in $\mathcal{C}$. While this approach might seem somewhat inefficient, it is practical for two reasons. First, it guarantees to find a graph in $\mathcal{G}^n \cap \mathcal{C}$, if one exists. Second, although the number of unlabeled graphs increases exponentially with their maximum order, the majority of results on forbidden subgraphs in the literature involve graphs having 10 or less vertices, which can be exhaustively enumerated quite rapidly.

To enumerate the graphs of n or less vertices, we use the method proposed by McKay (McKay, 1998). This method employs a canonical representation of a graph G, which partially orders the vertices of G in a way that is independent of their label. This partial

ordering is then used to transform the lattice of graphs into a rooted tree with a function p that uniquely maps a graph G to a parent graph $p(G)$. The root of this tree is the graph containing a single vertex, and the parent $p(G)$ of G is obtained by removing from G a minimum vertex according to the partial ordering. By traversing this tree in a depth-first manner, we can thus enumerate the graphs without generating isomorphic copies. An important advantage of this approach is that it allows to efficiently prune the search space. To illustrate this, suppose we want to find a graph of $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{H}}$, and let G be any graph. If $G \notin \mathcal{G}_{\mathcal{H}}$, then there is a graph $H \in \mathcal{H}$ such that $H \subseteq G$. Since all supergraphs[1] of G will also contain H, we can avoid the exploration of the subtree rooted at G without fear of missing any graphs of $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{H}}$.

6.6.2 Heuristic approach

It some cases, the FSCs and conditions to have an FSC for a graph class $\mathcal{C}$ may be out of reach of the enumerative approach. We then need to trade the completeness of that approach to gain the ability of finding graphs that have more vertices. This sacrifice is a reasonable one, since we already limit the order of graphs, and thus, cannot do better than find conjectures.

The heuristic approach we use is similar to the one employed by AGX to find extremal graphs (Caporossi et Hansen, 2000). In this approach, the task of finding a graph of $\mathcal{C}$ is modeled as an optimization problem where the objective function f determines the membership of a graph G in $\mathcal{C}$. For example, we can express this function such that $f(G) \leq 0$ if and only if G belongs to $\mathcal{C}$, and use any heuristic search method to minimize f until we find $f(G) \leq 0$. In AGX, the heuristic used is the *Variable Neighborhood Search* (Mladenovic et Hansen, 1997). This heuristic starts with a random graph $G \in \mathcal{G}^n$ and explores the neighborhood containing the graphs that can be obtained from G

[1] A graph G is a supergraph of a graph H if and only if H is a subgraph of G.

by considering every possible group of 2 to 4 vertices, and then adding or removing edges between vertices of this group. If no neighbor graph improving G is found, the neighborhood is then widened by increasingly adding a random perturbation to G before exploring its neighborhood, until a neighbor improving G is found, or until the maximum neighborhood size is attained or a given time limit is reached. Furthermore, if a neighbor improving G is found, the search is then recentered on this neighbor and the original neighborhood is restored. Using this approach, AGX was able to find extremal graphs of up to 50 vertices.

To illustrate this approach, consider a set of graphs invariants $\{\gamma_1, \gamma_2, \ldots, \gamma_M\}$, and a class containing graphs G such that

$$a_1\gamma_1(G) \; + \; a_2\gamma_2(G) \; + \; \ldots \; + \; a_M\gamma_M(G) \; \leq \; b,$$

where $a_i \in \mathbb{R}$, $1 \leq i \leq M$ and $b \in \mathbb{R}$. Furthermore, let $\mathcal{H} = \{H_1, H_2, \ldots, H_N\}$ be a set of forbidden subgraphs, and denote $\mathrm{emb}(H, G)$ the number of different embeddings, i.e. subgraph isomorphisms, of H in G. The task of finding G in $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{H}}$ can be formulated as maximizing

$$f(G) = -b \; + \; \sum_{i=1}^{M} a_i\gamma_i(G) \; - \; B\sum_{j=1}^{N} \mathrm{emb}(H_j, G),$$

where B is big constant chosen such that any non-zero value of the right sum dominates the value of the left sum minus b. If we find G such that $f(G) > 0$, we know that G is in $\overline{\mathcal{C}} \cap \mathcal{G}_{\mathcal{H}}$.

6.7 Experimental results

In this section, we use our algorithms to find results related to the concepts of independence, domination and irredundance of graphs. Before presenting the results, we need to define these concepts.

An *independent* (or *stable*) set S is a set of pairwise non-adjacent vertices. The *independence number* of a graph G, written $\alpha(G)$, is the maximum cardinality of an independent set of G, and the *independent domination number* of G, denoted $i(G)$, is the minimum cardinality of a maximal independent set of G. Furthermore, a *dominating set* T is a set of vertices such that each vertex of $V(G) \setminus T$ is adjacent to at least one vertex of T. The *domination number* of G, written $\gamma(G)$, is the minimum cardinality of a dominating set of G, and we denote $\Gamma(G)$ the maximum cardinality of a minimal dominating set of G. Moreover, let $X \subseteq V$, a vertex $x \in X$ is irredundant in X if it is isolated in X or if it has a private neighbor, i.e. a vertex $y \in V \setminus X$ such that x is the only vertex of X adjacent to y. The set X is irredundant if all its vertices are irredundant. We denote $IR(G)$ and $ir(G)$, respectively, the maximum cardinality of an irredundant set of G and the minimum cardinality of a maximal irredundant set of G. A famous result known as the *domination chain*, e.g. (Haynes et al., 1998), states that the following relations hold for all graphs $G \in \mathcal{G}$:

$$ir(G) \leq \gamma(G) \leq i(G) \leq \alpha(G) \leq \Gamma(G) \leq IR(G).$$

In the next two sections, we search for SFSCs and NFSCs on the classes of graphs that satisfy or not some of these relations at equality. This is a two step process. First, we use our algorithms to find conjectures. Working under the assumption that the forbidden subgraphs have at most 10 vertices, we use the enumerative approach to find specific graphs in these algorithms. We then demonstrate by hand the conjectures found by our

algorithms.

6.7.1 Conjectures on SFSC

As a first experiment, we set out to find SFSCs for the classes of graphs which satisfy at equality the relations of the domination chain. This task has been the subject of previous work by graph theorists. In (Allan et Laskar, 1978), Allan and Laskar have shown that if a graph G does not contain the first graph of Figure 6.7, known as the "claw", as subgraph, then $\gamma(G) = i(G)$. Furthermore, in (Favaron, 1986), Favaron, based on some earlier work by Bollobas and Cockayne (Bollobas et Cockayne, 1979), conjectured that if G does not contain as subgraph any of the graphs of Figure 6.8, then $ir(G) = \gamma(G)$. This conjecture was later proved by Puech in (Puech, 1998), and by Volkmann and Zverovich in (Volkmann et Zverovich, 2002). Favaron also showed in (Favaron, 1986) that if G does not contain the first nor the second graph of Figure 6.7, known as the "deer", then $ir(G) = i(G)$, and that if G does not contain any graph of the same figure, then $\Gamma(G) = IR(G)$.

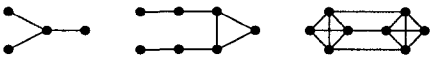


Figure 6.7 Graphs used in the SFSCs for graphs satisfying at equality some relations of the domination chain.

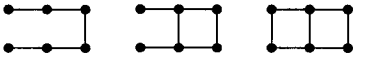


Figure 6.8 A SFSC for the class of graphs G such that $ir(G) = \gamma(G)$.

We first generated conjectures on SFSCs for the class of graphs G such that $\gamma(G) = i(G)$. Figure 6.9 shows a particular execution of our algorithm using $N = 1$ as maximum

number of graphs and $\mathcal{G}^{10}$ as search space. At iteration $k = 0$, the algorithm finds a graph H_0 such that $\gamma(H_0) < i(H_0)$ and a graph G_0 that does not contain H_0 and such that $\gamma(G_0) < i(G_0)$. The algorithm then finds a graph H_1 that is a common subgraph of both H_0 and G_0. Since the algorithm cannot find an H_1-free graph G_1 such that $\gamma(G_1) < i(G_1)$, it return $\mathcal{H} = \{H_1\}$, which is the SFSC proposed by Allan and Laskar in (Allan et Laskar, 1978).

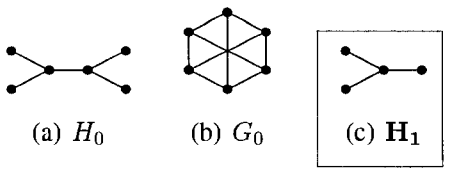


Figure 6.9 An illustration of the SFSC algorithm, for the class of graphs G such that $\gamma(G) = i(G)$, using $N = 1$ as maximum number of graphs and $\mathcal{G}^{10}$ as search space. Graph H_1 is an SFSC that was originally found by Allan and Laskar in (Allan et Laskar, 1978).

Note that we can obtain the same SFSC for higher values of N, as shown in Figure 6.10. In this example, where $N = 2$ is used, the algorithm first finds a graph $F \in \overline{\mathcal{C}}$ of minimum order, such that $\gamma(F) < i(F)$, and then searches for a maximum common subgraph of $\overline{\mathcal{C}} \cap G_{\{F\}}$. Since the algorithm cannot find a graph G_1 that does not contain F or H_1, and such that $\gamma(G_1) < i(G_1)$, it stops with $\{F, H_1\}$ as SFSC. However, since $H_1 \subset F$, the algorithm removes F from the SFSC and, once more, returns $\mathcal{H} = \{H_1\}$.

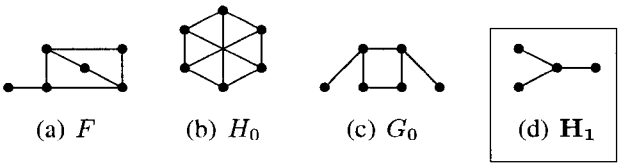


Figure 6.10 Another illustration of the SFSC algorithm, for the class of graphs G such that $\gamma(G) = i(G)$, using $N = 2$ as maximum number of graphs and $\mathcal{G}^{10}$ as search space. Graph H_1 is an SFSC that was originally found by Allan and Laskar in (Allan et Laskar, 1978).

We have also used our algorithm to find conjectures on SFSCs for the class of graphs G such that $ir(G) = \gamma(G)$. In the execution shown in Figure 6.11, our algorithm generated the conjecture that the graphs F and H_3 of Figure 6.11, known as the "fork" and the "deer', form an SFSC for this class of graphs. As we now prove, this conjecture is a novel result that strengthens Favaron's result, presented in (Favaron, 1986), that $ir(G) = \gamma(G)$ if G is "claw"-free and "deer"-free.

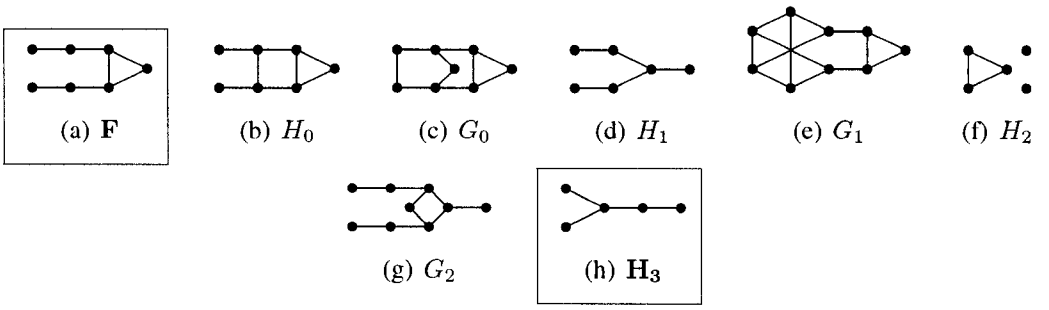


Figure 6.11 An illustration of the SFSC algorithm, for the class of graphs G such that $ir(G) = \gamma(G)$, using $N = 2$ as maximum number of graphs and $\mathcal{G}^{10}$ as search space. Graphs F and H_3 form a novel SFSC that strengthens a previous result proposed by Favaron in (Favaron, 1986).

Theorem 1. *If a graph G does not contain the graph F or H_3 of Figure 6.11, known respectively as the "fork" and the "deer", then $ir(G) = \gamma(G)$.*

Proof. Let I be a maxmial irredundant set with $|I| = ir(G)$, and let A be the set of vertices in $V \setminus I$ that are not adjacent to any vertex of I. If A is empty then I is a dominating set and we have $\gamma(G) \leq ir(G)$. Thus, we assume that $A \neq \emptyset$ and denote $N_X(v)$ the set of vertices in X that are adjacent to a vertex v. Furthermore, for a vertex $x \in I$, we denote $P(x)$ the private neighbors of x, i.e. the vertices $y \in V \setminus I$ such that $N_I(y) = \{x\}$. Since I is irredundant, $P(x) \neq \emptyset$ for all $x \in I$ such that $N_I(x) \neq \emptyset$. Let $W \subseteq I$ be the subset of vertices in I which are redundant in $I \cup A$, and for any $w \in W$, denote $A(w)$ the subset of vertices $a \in A$ such that w is redundant in $I \cup \{a\}$. Note that $\bigcup_{w \in W} A(w) = A$ and that $A(w) \neq \emptyset$ for every $w \in W$. Also, for each $w \in W$,

every vertex $a \in A(w)$ is adjacent to every vertex $u \in P(w)$, or else w would still be irredundant in $I \cup \{a\}$, which contradicts the definition of $A(w)$. Moreover, $P(w)$ forms a clique, otherwise there would be two non adjacent vertices u_1 and u_2 in $P(w)$ such that $I \cup \{u_1\}$ is irredundant (since any vertex in $A(w)$ is a private neighbor of u_1 while u_2 is a private neighbor of w), a contradiction.

We now prove that every connected component of the subgraph $G[W]$ is a clique. This is obviously true for connected components with at most 2 vertices. So consider any connected component with at least three vertices, and assume it is not a clique. Then there are three vertices w_1, w_2 and w_3 in this component such that w_2 is adjacent to w_1 and w_3 while w_1 and w_3 are not adjacent. But then $\langle w_1, w_2, w_3, u, a\rangle \simeq H_3$ for all $u \in P(w_2)$ and $a \in A(w_2)$.

The next observation is that, for every $w \in W$ and $u \in P(w)$ the vertices in $N_A(u)$ induce a clique. Indeed, suppose there are two non adjacent vertices a_1 and a_2 in $N_A(u)$. Then, since $N_I(w) \neq \emptyset$, there is a vertex $x \in I$ adjacent to w, which means that $\langle w, x, u, a_1, a_2\rangle \simeq H_3$, a contradiction. As a consequence, $A(w)$ forms a clique for all $w \in W$.

Note also that if C is the vertex set of a connected component of $G[W]$ with at least 3 vertices, then $\bigcup_{w\in C} A(w)$ induces a clique in G. Indeed, suppose this is not the case and consider two non adjacent vertices a_1 and a_2 in $A(w_1) \cup A(w_2)$. Moreover, consider any $u_1 \in P(w_1)$ and $u_2 \in P(w_2)$. If both a_1 and a_2 are adjacent to u_1 or u_2 then they are adjacent to each other since $N_A(u_1)$ and $N_A(u_2)$ induce cliques, a contradiction. Otherwise, suppose, without loss of generality, that a_1 is adjacent only to u_1 and a_2 only to u_2. Then, vertices u_1 and u_2 are not adjacent, otherwise $\langle w_1, u_1, u_2, a_1, a_2\rangle \simeq H_3$. Furthermore, consider any vertex $w_3 \in C \setminus \{w_1, w_2\}$. As shown above, $\{w_1, w_2, w_3\}$ forms a clique and, thus, $\langle w_1, w_2, w_3, u_1, u_2, a_1, a_2\rangle \simeq F$, a contradiction.

In what follows, the connected component of $G[W]$ with one vertex will be said to be of type 1, while those with at least 3 vertices will be said to be of type 2. We split the connected components with 2 vertices w_1, w_2 into three disjoint groups.

1. If there is a vertex in $P(w_1) \cup P(w_2)$ adjacent to all vertices in $A(w_1) \cup A(w_2)$, then the component is said to be of type 3.
2. If every vertex in $P(w_1) \cup P(w_2)$ has a non neighbor in $A(w_1) \cup A(w_2)$, and there exist two non adjacent vertices $u \in P(w_i)$ and $a \in A(w_j)$ $(j \neq i)$ such that a is adjacent to all vertices in $A(w_i)$, then the component is said to be of type 4.
3. If every vertex in $P(w_1) \cup P(w_2)$ has a non neighbor in $A(w_1) \cup A(w_2)$, and for all non adjacent vertices $u \in P(w_i)$ and $a \in A(w_j)$ $(j \neq i)$ there is a vertex $a' \in A(w_i)$ not adjacent to a, then the component is said to be of type 5.

We now create a set I' from I as follows.

1. For every connected component $C = \{w\}$ of type 1, we remove w from I and add a vertex $u \in P(w)$.
2. For every connected component C of type 2, we remove a vertex $w \in C$ from I and add a vertex $a \in A(w)$.
3. For every connected component $C = \{w_1, w_2\}$ of type 3, one of the vertices in C, say w_1, has a private neighbor $u \in P(w_1)$ which is adjacent to all vertices in $A(w_1) \cup A(w_2)$. We remove w_1, from I and replace it by u.
4. For every connected component $C = \{w_1, w_2\}$ of type 4, there exist two non adjacent vertices $u \in P(w_i)$ and $a \in A(w_j)$ $(j \neq i)$ such that a is adjacent to all vertices in $A(w_i)$. We remove w_j from I and replace it by a.
5. For every connected component $C = \{w_1, w_2\}$ of type 5, we remove w_1 and w_2 from I and replace them by a vertex $u_1 \in P(w_1)$ and a vertex $u_2 \in P(w_2)$.

Since $|I'| = |I|$ it is now sufficient to prove that I' is a dominating set. So let v be a vertex that does not belong to I'.

1. If $v \in I$ then v was removed from a connected component C of one of the above five types. If C is of type 1 or 5, then v is adjacent to the neighbor u of v that was added to I'. Else, C contains at least one neighbor w of v that was not removed from I.

2. If $v \in \bigcup_{w \in W} P(w)$ then let w be the vertex in W such that $v \in P(w)$. If $w \in I'$, then w is a neighbor of v in I'. Otherwise, w was removed from a connected component C. If C is of type 1, 3 or 5, then w was replaced by a vertex in $P(w)$, and v has a neighbor in I' since $P(w)$ is a clique. If C is of type 2 or 4, then w was replaced by a vertex in $A(w)$, and v has a neighbor in I' since v is adjacent to all the vertices in $A(w)$.

3. If $v \in A$ then let w be a vertex in W such that $v \in A(w)$ and let C be the connected component of $G[W]$ containing w. If C is of type 1 or 3, then a vertex w was replaced by a vertex $u \in P(w)$, and v has a neighbor in I' since u is adjacent to all vertices in $\bigcup_{x \in C} A(x)$. If C is of type 2 or 4, then a vertex in C was replaced by a vertex $a \in \bigcup_{x \in C} A(x)$, and v has a neighbor in I' since a is adjacent to all other vertices in $\bigcup_{x \in C} A(x)$. Finally, if C is of type 5, then the vertices w_1 and w_2 have been replaced by $u_1 \in P(w_1)$ and $u_2 \in P(w_2)$, which means that v has a neighbor in I' since $v \in N_A(u_1) \cup N_A(u_2)$.

4. If $v \notin \left(I \cup A \cup \bigcup_{w \in W} P(w)\right)$ then v is adjacent to at least two vertices in I. Indeed, suppose it is not adjacent to any vertex of I. Then, $I \cup \{v\}$ would be irredundant, which contradicts the maximality of I. Also, v cannot be adjacent to only one vertex of W, or else it would be a private neighbor of this vertex, a contradiction. Thus, v is adjacent to at least two vertices of W. If one of these neighbors belongs to I' then there is nothing to prove. Thus, assume v has no

neighbor in I' and consider any neighbor $w \in I \setminus I'$ of v. Vertex w belongs to a connected component C of $G[W]$ and was removed from I.

(a) If C is of type 1 or 3, then w was replaced by a vertex $u \in P(w)$. Let a be any vertex in $A(w)$. Vertex v is not adjacent to a, or else $I \cup \{v\}$ would be irredundant, contradicting the maximality of I. Since $N_I(w) \neq \emptyset$, there is a vertex $w' \in I$ adjacent to w. Furthermore, we know that $w' \notin W$, otherwise it would be in C, which contradicts the type of C. Consequently, w' is in I', and v is not adjacent to w'. If v is adjacent to u, then v has a neighbor in I'. Otherwise, $\langle v, w, w', u, a \rangle \simeq H_3$, a contradiction.

(b) If C is of type 2, then let x be any vertex in $I \setminus \{w\}$ adjacent to v. Such a vertex necessarily exists since $v \notin P(w)$. If x is in $C \setminus \{w\}$, then it is in I' and v is adjacent to a vertex of I'. Else, there exists a connected component $C' \neq C$ of $G[W]$ containing x. Consider any two vertices w' and w'' in $C \setminus \{w\}$, and let u be any vertex in $P(w')$ and a any vertex in $A(w')$. Vertex v is not adjacent to u, otherwise $\langle v, w, w'', u, x \rangle \simeq H_3$. Also, v is not adjacent to a, or else $\langle v, w, w', x, a \rangle \simeq H_3$. Hence $\langle v, w, w', w'', x, u, a \rangle \simeq F$, a contradiction.

(c) If C is of type 5, then the vertices w_1 and w_2 were replaced by $u_1 \in P(w_1)$ and $u_2 \in P(w_2)$. Since no vertex in $P(w_2)$ is adjacent to all vertices in $A(w_1)$, consider a vertex $a_1 \in A(w_1)$ not adjacent to u_2. Moreover, since a_1 is not adjacent to all vertices in $A(w_2)$, consider any vertex $a_2 \in A(w_2)$ not adjacent to a_1. Vertex v is not adjacent to a_1 nor to a_2, or else $I \cup \{v\}$ would be irredundant. Also, we may assume that v is not adjacent to u_1 nor u_2, otherwise v would have a neighbor in I'. Furthermore, we know that u_1 is not adjacent to a_2, or else $\langle w_1, w_2, u_1, a_1, a_2 \rangle \simeq H_3$, and therefore that u_1 is not adjacent to u_2, otherwise $\langle w_1, u_1, u_2, a_1, a_2 \rangle \simeq H_3$. Also, since w is one of the two vertices w_1 or w_2, we know that v is adjacent to at least one of

them, say w_1. But then $\langle v, w_1, w_2, u_1, a_1 \rangle \simeq H_3$ (if v is not adjacent to w_2) or $\langle v, w_1, w_2, u_1, u_2, a_1, a_2 \rangle \simeq F$, a contradiction.

(d) If C is of type 4, then let x be any vertex in $I \setminus \{w\}$ adjacent to v, and let $C' \neq C$ be the connected component of $G[W]$ with $x \in C'$. We have shown above that if C' is of type 1, 2, 3 or 5, then v has a neighbor in I'. So assume C' is also of type 4. Let w' be the second vertex in C, x' be the second vertex in C', and u be any vertex in $P(w)$. Since w is not in I' and because C is of type 4, we know that w' is in I'. Moreover, assume that v is not adjacent to w', or else v would be adjacent to a vertex in I'. Likewise, assume that x' is in I' and is not adjacent to v, otherwise v would have a neighbor in I'. Furthermore, since C is of type 4, w was replaced by a vertex a adjacent to all vertices in $A(w')$, and not adjacent to a vertex $u' \in P(w')$. Since no vertex of $P(w)$ is adjacent to all the vertices in $A(w')$, consider any $a' \in A(w')$ not adjacent to u. If v is adjacent to a then v has a neighbor in I', so assume v and a are not adjacent. Vertex v is then necessarily adjacent to u, or else $\langle v, w, w', u, a \rangle \simeq H_3$. Also, v is not adjacent to u', otherwise $\langle v, w, u', x, x' \rangle \simeq H_3$, and u is not adjacent to u', or else $\langle v, u, u', x, a \rangle \simeq H_3$. But then $\langle v, w, w', x, x', u, u' \rangle \simeq F$, a contradiction.

□

If we combine the SFSCs we obtained for the classes $ir(G) = \gamma(G)$ and $\gamma(G) = i(G)$, we get the SFSC proposed by Favaron: if a graph G is "claw"-free and "deer"-free then $ir(G) = i(G)$. However, if the SFSC algorithm randomly selects the maximum common subgraphs, at each iteration, we can obtain a completely different SFSC. For instance, in the execution shown in Figure 6.12, our algorithm found a path of 5 vertices, which was shown by Puech in (Puech, 1998) to be an SFSC.

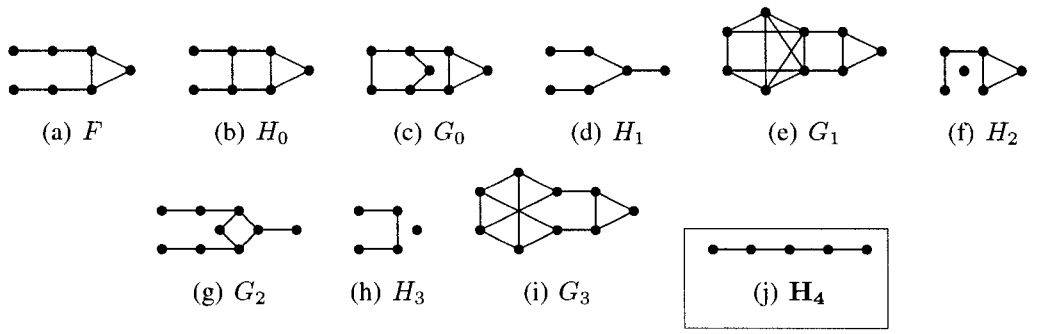


Figure 6.12 Another illustration of the SFSC algorithm, for the class of graphs G such that $ir(G) = \gamma(G)$, using $N = 2$ as maximum number of graphs and $\mathcal{G}^{10}$ as search space. Graph H_4 was shown be an SFSC by Puech in (Puech, 1998).

6.7.2 Conjectures on NFSC

Following what we have done for SFSCs, we look in this section for NFSCs on the classes of graphs satisfying at inequality some of the relations of the domination chain.

As a first experiment, we used our algorithm to find conjectures on NFSCs for the class of graphs G such that $\gamma(G) < i(G)$. Figure 6.13 shows a particular execution of the algorithm for this task, using $\mathcal{G}^8$ as search space. At iteration $k = 0$, the algorithm finds a graph G_0 such that $\gamma(G_0) < i(G_0)$. Since $\mathcal{L}_1$ only contains G_0, the algorithm then finds a minimum order graph H_1 not included in G_0. At the next iteration, the algorithm then finds a graph G_1 containing H_1 and such that $\gamma(G_1) < i(G_1)$, and a graph H_2 of minimum order that is not a subgraph of G_0 or G_1. This process is repeated until the algorithm reaches iteration $k = 5$, where it finds a 5 vertex clique H_5, as the minimum order graph not included in $\{G_0, G_1, G_2, G_3, G_4\}$. However, the algorithm does not find a graph G_5 containing H_5 and such that $\gamma(G_5) < i(G_5)$, and thus returns $\mathcal{H} = \{H_5\}$. Repeating this experiment using $\mathcal{G}^9$ and $\mathcal{G}^{10}$ as search space, our algorithm has found complete graphs of 6 and 7 vertices, suggesting the novel NFSC that a graph G of n vertices is $K_{(n-3)}$-free if $\gamma(G) < i(G)$. We now demonstrate this result.

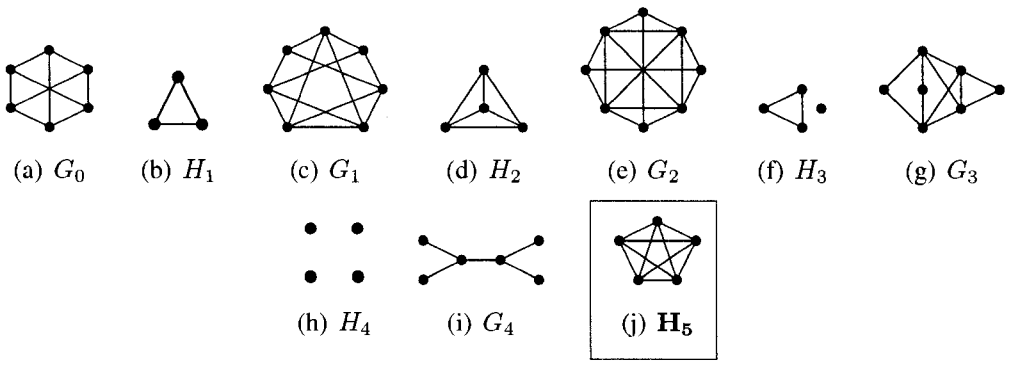


Figure 6.13 An illustration of the NFSC algorithm, for the class of graphs G such that $\gamma(G) < i(G)$, using $\mathcal{G}^8$ as search space.

Theorem 2. *Let G be any graph of order n. If $\gamma(G) < i(G)$ then G is $K_{(n-3)}$-free.*

Proof. We will show that $i(G) \leq \gamma(G)$, if G contains a $K_{(n-3)}$. This will be done by showing that a minimum cardinality dominating set can be transformed into an independant dominating set of the same cardinality.

Let $D = \{w_1, w_2, \ldots, w_m\}$ be a minimum cardinality dominating set of G, i.e. $|D| = \gamma(G)$. If $E(G[D]) = \emptyset$, then D is an independent set, and we have $i(G) \leq |D| = \gamma(G)$. Otherwise, let (w_1, w_2) be any edge of $E(G[D])$, and let $P(w)$ be the private neighbors of w, i.e. the vertices $y \in V \setminus D$ such that $N_D(y) = \{x\}$. We know that $P(w_i) \neq \emptyset$, $i = 1, 2$, otherwise $D' = D \setminus \{w_i\}$ would be a dominating set of one less vertex than D, which contradicts the minimality of D. We next show that at least one of $P(w_1)$ or $P(w_2)$ is a complete graph. Suppose this is not the case, then there are four vertices $u_1 \in P(w_1)$, $v_1 \in P(w_1)$, $u_2 \in P(w_2)$, $v_2 \in P(w_2)$ such that u_1, v_1 are non-adjacent, and u_2, v_2 are non-adjacent. However, $\langle u_1, v_1, w_1, u_2, v_2, w_2 \rangle$ contains no triangle, and thus G contains no $K_{(n-3)}$, which is a contradiction. Therefore, $P(w_1)$ or $P(w_2)$ forms a complete graph, without loss of generality, suppose it is $P(w_1)$. Let u_1 be any vertex of $P(w_1)$. The set $D' = (D \setminus \{w_1\}) \cup \{u_1\}$ is a dominating set of cardinality

$|D'| = |D|$ such that $E(G[D']) < E(G[D])$. By repeating this process iteratively at most $n - 3$ times, we will necessarily get an independant dominating set D'', such that $i(G) \leq |D''| = |D| = \gamma(G)$. □

In the next experiment, we searched for NFSCs on the class of graphs G such that $ir(G) < \gamma(G)$. Figure 6.14 shows an execution of our NFSC algorithm, using $\mathcal{G}^8$ as search space, where a complete graph of 5 vertices is found. Again, when repeating this experiment with G^9 and G^{10}, our algorithm found, although not systematically, complete graphs of 6 and 7 vertices, which suggested a new theorem and corollary:

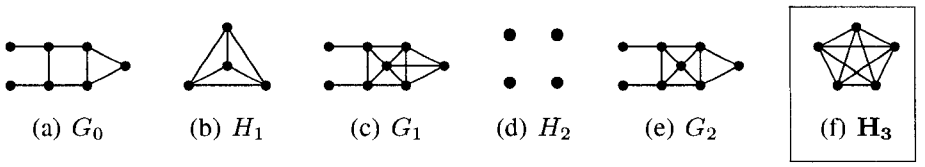


Figure 6.14 An illustration of the NFSC algorithm, for the class of graphs G such that $ir(G) < \gamma(G)$, using $\mathcal{G}^8$ as search space.

Theorem 3. *Let G be any graph of order n. If $ir(G) < \gamma(G)$ then G is $K_{(n-3)}$-free.*

Proof. Similar to what was done for the last theorem, we will show that $\gamma(G) \leq ir(G)$ if G contains a $K_{(n-3)}$, by showing that a minimum cardinality maximal irredundant set can be transformed into a dominating set of the same cardinality.

Let I be a minimum cardinality maximal irredundant set of G, i.e. $|I| = ir(G)$, and let A be the set of vertices not dominated by I. If $A = \emptyset$, then I is a dominating set, and we have $\gamma(G) \leq |I| = ir(G)$. Otherwise, consider any vertex $a \in A$. There is a non-isolated vertex $w \in I$ which is redundant in $I \cup \{a\}$, otherwise I would not be maximal. Let $P(w)$ denote the private neighbors of w. We know that $P(w)$ is not empty since w is non-isolated in I. Furthermore, we know that a is adjacent to all the vertices of $P(w)$, otherwise w would be irredundant in $I \cup \{a\}$. We also know that $P(w)$ forms

a complete graph, otherwise there are two vertices u_1, u_2 in $P(w)$ such that $I \cup \{u_1\}$ is irredundant (since u_1 has a as private neighbor and w the vertex u_2, thus contradicting the maximality of I). Denote $A(w) \subseteq A$ the vertices a such that w is redundant in $I \cup \{a\}$, and let W be the vertices of I that are redundant in $I \cup A$.

Consider any vertex $w_1 \in W$ such that $|A(w_1)| = \max_{w \in W} |A(w)|$. We show that $A(w_1) = A$. Suppose this is not the case, then consider any vertex $a_2 \in A \setminus A(w_1)$, and let w_2 be any vertex redundant in $I \cup \{a_2\}$. Since a_2 belongs to $A(w_2)$ but not to $A(w_1)$, and since $|A(w_1)| \geq |A(w_2)|$, there is a vertex $a_1 \in A(w_1)$ that does not belong to $A(w_2)$. Let u_1 be a vertex in $P(w_1)$ that is not adjacent to a_2, and u_2 be a vertex in $P(w_2)$ that is not adjacent to a_1. Then, $\langle a_1, u_1, w_1, a_2, u_2, w_2 \rangle$ contains no triangle, and thus G contains no $K_{(n-3)}$, a contradiction. Therefore, $A(w_1) = A$ and, since any vertex of $A(w_1)$ is adjacent to every vertex of $P(w_1)$ and because $P(w_1)$ forms a complete graph, the set $I' = (I \setminus \{w_1\}) \cup \{u_1\}$ is a dominating set of G. Thus, we have that $\gamma(G) \leq |I'| = |I| = ir(G)$. □

Corollary 1. *Let G be any graph of order n. If $ir(G) < i(G)$ then G is $K_{(n-3)}$-free.*

6.7.3 Conjectures on FSC

Hereditary graph classes that can be characterized with a limited number of forbidden subgraphs are not that common in graph theory. Furthermore, those that are known have been studied in depth, and have already been characterized with forbidden subgraphs. Thus, rather than searching for new FSCs, we will, in this section, try to reproduce some known results.

A *split graph* is a graph in which the vertices can be partitioned into a clique and an independent set. This class of graphs has some very interesting properties that have been studied, among others, by Foldes, Hammer and Simeone (Foldes et Hammer, 1977;

Hammer et Simeone, 1981). One of these properties is that split graphs can be recognized in linear time, using the sequence of degrees of each vertex. Let G be graph with n vertices, we order the degrees of the vertices of G, by non-increasing value, to form a sequence $d_1 \geq d_2 \geq \ldots \geq d_n$. Let m be the largest value of i such that $d_i \geq i - 1$, then G is a split graph if and only if

$$m(m-1) - \sum_{i=1}^{m} d_i + \sum_{j=m+1}^{n} d_j \leq 0.$$

If this is the case, the m vertices with the largest degrees form a maximum clique in G, and the remaining vertices an independent set.

Using the above test to determine whether or a graph is a split graph and $\mathcal{G}^8$ as search space, we have found the minimal FSC for this class, shown in Figure 6.15, containing two cycles of 4 and 5 vertices, and a graph made of two disjoint edges. This FSC is a well known result, first described by Foldes and Hammer in (Foldes et Hammer, 1977).

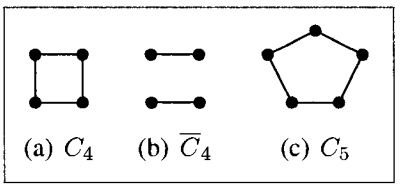


Figure 6.15 An illustration of the FSC algorithm for the class of split graphs, using $\mathcal{G}^8$ as search space.

6.8 Conclusion

We have presented, in this paper, some methods to automate the discovery of new conjectures on forbidden subgraph characterization. The first two algorithms that were described allow to find sufficient conditions to have a characterization, in the form of a

set of graphs $\mathcal{H}$ such that a graph G is in a graph class $\mathcal{C}$ if G is $\mathcal{H}$-free. While the first of these algorithms returns a single forbidden subgraph, the second allows to find sufficient conditions involving multiple subgraphs. Since a given graph class can have a great number of sufficient conditions, we described two measures, complexness and tightness, that were used to select the forbidden subgraphs. Furthermore, using these same measures, we have developed two other algorithms to find necessary conditions in the form of a set of graphs $\mathcal{H}$ such that all graphs of a given class $\mathcal{C}$ are $\mathcal{H}$-free. Lastly, we presented an algorithm that finds actual characterizations.

Although these algorithms find forbidden subgraph characterizations, or conditions to have such characterizations, we have seen that this can be impossible in practice, due to the infinite number of graphs. However, we have shown that, by bounding the order of graphs under consideration, we could use our algorithms to find conjectures. Following this idea, we have used our algorithms to reproduce some important results, as well as to find new ones, on the domination, independence and irredundance of graphs. In particular, our algorithms conjectured two new results which we proved were correct: a n vertex graph G is such that $ir(G) = \gamma(G)$ if it is "fork"-free and "deer"-free, and that G is $K_{(n-3)}$-free if $ir(G) < i(G)$.

CHAPITRE 7

DISCUSSION GÉNÉRALE

Nous faisons, dans ce chapitre, un bref résumé des résultats obtenus dans le contexte de cette thèse, et comparons ces résultats avec les objectifs de recherche initialement définis.

Nous avons présenté, dans le premier article, un nouvel algorithme appelé SYGMA améliorant la découverte des sous-graphes fréquents d'un ensemble de graphes contenant un nombre restreint d'étiquettes différentes. Cette amélioration a été obtenue, en partie, à l'aide d'une stratégie efficace d'énumération, faisant l'usage d'invariants de graphe. Cette stratégie nous a permis de définir deux techniques permettant de détecter rapidement les graphes redondants et non-redondants durant l'exploration. De plus, l'information sur la symétrie d'un graphe, obtenue par SYGMA lors de l'exploration, a permis de développer des techniques pouvant identifier certains graphes non-fréquents, évitant ainsi plusieurs calculs coûteux. Par ailleurs, nous avons pu évaluer, dans la partie expérimentale de l'article, la performance de notre algorithme en le comparant à un des algorithmes le plus connus pour ce problème, l'algorithme GSPAN. Dans un premier temps, nous avons comparé ces deux algorithmes sur la tâche d'énumérer tous les graphes ayant un nombre limité de sommets et d'étiquettes. Grâce à cette expérience, nous avons pu constater que la stratégie d'énumération de SYGMA est beaucoup plus efficace que celle de GSPAN, en particulier pour les graphes sans étiquettes, où notre algorithme est jusqu'à 75 fois plus rapide que GSPAN. Par ailleurs, nous avons également employé ces deux algorithmes pour trouver les sous-graphes fréquents de plusieurs jeux de données, générés aléatoirement et provenant d'une application réelle du domaine de la chimie computationnelle. Encore une fois, les résultats de ces expériences ont montré

SYGMA de loin supérieur à GSPAN pour les jeux de données ne comportant aucune étiquette (jusqu'à 110 fois plus rapide que GSPAN), et un gain non négligeable pour les instances ayant peu d'étiquettes. En somme, nous avons atteint notre premier objectif de développer une méthode efficace pour trouver les sous-graphes fréquents d'une base de graphes ayant peu d'étiquettes.

Dans le deuxième article, nous avons présenté une stratégie, à la fois générale et simple, pour améliorer la découverte des patrons fréquents. Cette stratégie utilise de l'information de fond sur les patrons fréquents, sous la forme d'une fonction heuristique, permettant de définir la topologie de l'espace de recherche de manière à minimiser le nombre de calculs de support dans la base de données lors de l'exploration. Afin d'évaluer notre stratégie, nous avons testé celle-ci sur le problème de la découverte des sous-graphes fréquents. Nous avons tout d'abord présenté une technique pour appliquer notre stratégie au cas des sous-graphes fréquents, similaire à celle employée par SYGMA. Ensuite, nous avons proposé une simple heuristique utilisant la fréquence des étiquettes dans la base de données, et avons testé cette heuristique sur des jeux de tests aléatoires générés de manière a introduire un biais croissant dans la distribution des étiquettes. Lors de ces tests, nous avons comparé notre méthode avec trois autres méthodes : la première faisant l'exploration d'un espace de recherche dont la topologie a été définie de manière aléatoire, la seconde utilisant la technique d'exploration de l'algorithme GSPAN, et la dernière employant une technique d'exploration complexe permettant de tester la fermeture descendante des graphes explorés. Comme prévu, nous avons constaté un gain en vitesse de notre méthode et de l'algorithme vérifiant la fermeture descendante, par rapport aux deux autres algorithmes, proportionnel au biais dans les données. Dans une autre expérience, nous avons testé les quatre mêmes algorithmes sur le même jeu de données réelles que celui utilisé dans le premier article. Les graphes de ce jeu de données représentant des molécules, nous avons modifié légèrement notre heuristique pour exploiter certaines caractéristiques de ce type de données. Encore une fois, les résultats ont

montré un net avantage de notre méthode sur la méthode explorant l'espace de topologie aléatoire ainsi que sur l'algorithme GSPAN (de 8 à 15 fois plus rapide). Nous avons également observé un gain en vitesse de notre technique par rapport à la technique beaucoup plus complexe de vérifier la fermeture descendante. Somme toute, nous croyons avoir atteint l'objectif de développer une nouvelle technique pour réduire le nombre de calculs de support dans la découverte des patrons fréquents.

Finalement, dans le troisième article, nous avons introduit de nouvelles méthodes pour automatiser la génération de conjectures sur la caractérisation par sous-graphes interdits (CSI). Les deux premières méthodes décrites dans l'article servent à générer des conditions suffisantes pour avoir une CSI, la première produisant un seul sous-graphe interdit, et la seconde un ensemble contenant un nombre de sous-graphes interdits fourni par l'usager. De même, nous avons décrit deux autres méthodes pour obtenir des conditions nécessaires pour une avoir une CSI, sous la forme d'un ou de plusieurs sous-graphes interdits. Enfin, nous avons présenté une méthode pour obtenir de vraies CSI. Par ailleurs, nous avons vu que ces méthodes nécessitent l'exploration d'un espace infini de graphes, et avons proposé deux stratégies pour les implémenter : l'une transformant la recherche d'un graphe en problème d'optimisation et employant une métaheuristique pour résoudre ce problème, et l'autre faisant l'énumération de tous les graphes de taille limitée. Dans la section expérimentale de l'article, nous avons utilisé ces méthodes pour reproduire des résultats connus de la théorie des graphes, ainsi que pour trouver de nouveaux résultats portant sur les notions d'irredondance, de stabilité et de domination d'un graphe. Entre autres, notre programme a généré une conjecture sous la forme d'une condition suffisante pour avoir une CSI, plus forte que celle proposée dans la littérature :

> pour tout graphe G, la taille du plus petit ensemble irredondant de G, i.e. $ir(G)$ est égale à la taille de son plus petit ensemble dominant $\gamma(G)$, si G ne contient pas de *cerf* ni de *fourche* comme sous-graphe induit.

Notre programme a également produit une nouvelle conjecture portant sur une condition

nécessaire :

> pour tout graphe G d'ordre n, G ne contient pas de clique à n-3 sommets comme sous-graphe induit si $ir(G) < i(G)$.

Nous avons validé ces conjectures en les démontrant formellement.

Avec ces résultats, nous pouvons conclure que l'objectif de développer des méthodes pour automatiser la génération de conjectures sur la CSI a été atteint.

CONCLUSION

Bien que nous ayons atteint nos objectifs de recherche, il reste encore beaucoup à faire dans la découverte des patrons fréquents. Alors que ce problème a été utilisé avec grand succès dans plusieurs applications de la bioinformatique et de la chimie computationnelle, son emploi dans des applications d'autres domaines, notamment le Web sémantique, semble très prometteur. Un autre domaine où la découverte des patrons fréquents semble avoir beaucoup de potentiel est la vision par ordinateur. Comme les données traitées par les applications de ce domaine ont souvent la forme de graphes sans étiquette, il serait intéressant de voir les bénéfices qu'aurait notre algorithme SYGMA sur ces données. Par ailleurs, le calcul du support d'un graphe dans la base de données étant l'opération la plus coûteuse de la découverte des sous-graphes fréquents, il serait bénéfique de développer des méthodes plus efficaces pour résoudre le problème d'isomorphisme de sous-graphe. Une autre approche intéressante serait de relaxer la définition du support d'un graphe pour être le nombre de graphes de la base de données ayant un sous-graphe similaire à ce graphe, où la mesure de similarité pourrait être paramétrable. Cette approche permettrait d'employer des méthodes heuristiques pour le problème d'isomorphisme de sous-graphe, augmentant ainsi l'efficacité des algorithmes pour la découverte des sous-graphes fréquents et, par conséquent, la taille des bases de données pouvant être traitées. Dans le même ordre d'idée, il serait désirable d'explorer d'autres métriques que la fréquence pour mesurer l'intérêt d'un graphe, ou de façon plus générale, un patron. Hormis l'algorithme SUBDUE de Cook et Holder (Cook et Holder, 1994), qui a connu un succès appréciable dans le domaine, cette idée n'a pas réellement été explorée à ce jour. L'approche présentée au chapitre 5, où l'on choisit le parent d'un patron à l'aide d'une heuristique, se porterait sans doute bien à cette idée, vu sa grande flexibilité. Enfin, il serait intéressant d'employer la découverte des patrons fréquents sur d'autres types de données plus complexes que les graphes, comme les hypergraphes ou les rela-

tions en logique du premier ordre. Étant générique, l'approche présentée au chapitre 5 serait également utile à ce problème.

La génération automatisée de conjectures en mathématiques est également une discipline ayant un avenir prometteur, particulièrement en théorie des graphes où la diversité des relations observées, ainsi que la complexité des preuves récentes, exige un apport considérable de l'ordinateur. Malgré le nombre impressionnant de théorèmes obtenus à l'aide de l'ordinateur, le nombre de résultats en théorie des graphes attendant d'être découverts est presque infini. Afin de faciliter la découverte de ces résultats, il serait bon d'avoir un répertoire centralisé des résultats déjà connus de ce domaine. Cela permettrait à plusieurs systèmes de bénéficier de ces connaissances et éviterait la génération en double d'un même résultat. Des efforts dans cette direction ont déjà faits, par exemple, le document *Written-on-the-wall* (Fajtlowicz, 2008; Fajtlowicz et DeLaVina, 2008) contenant un grand nombre de conjectures sur des relations entre invariants de graphe, et le projet ISGCI (Brandstädt et al., 2003) qui catalogue une grande quantité de relations d'inclusions de classes de graphes. De même, il serait utile d'établir des mesures quantitatives et qualitatives permettant d'évaluer l'intérêt d'une conjecture. La résolution de ce problème, abordé par Hansen dans (Hansen et al., 2005; Hansen, 2005) et par Larson dans (Larson, 2002), permettrait de guider les efforts mis dans le processus de découverte vers des résultats utiles. Nous avons d'ailleurs été confrontés à ce problème lors du développement de nos méthodes générant des conditions suffisantes ou nécessaires sur la caractérisation d'une classe impliquant plus d'un sous-graphe interdit. À ce titre, il serait intéressant d'explorer plus à fond l'aspect théorique du problème, et de proposer d'autres critères évaluant la valeur d'une CSI à plusieurs sous-graphes interdits. Par ailleurs, pour prolonger le travail fait dans cette thèse, nos méthodes pourraient être employées pour générer des CSI ou des conditions pour avoir une CSI sur d'autre classes de graphes. Ce processus pourrait même être automatisé. Par exemple, si les classes pour lesquelles on cherche des CSI contiennent des graphes satisfaisant à égalité certaines relations sur les

invariants de graphe, il suffirait de générer ces relations de manière automatique et de lancer, pour chacune d'elles, les méthodes de génération de conjectures. Une autre extension possible de cette thèse serait d'utiliser nos méthodes pour générer d'autres types de conjectures. On pourrait, par exemple, trouver des relations entre différentes classes de graphes pour lesquelles on possède des CSI ou des conditions pour avoir un CSI, en observant les sous-graphes interdits de ces CSI. Enfin, dans le but d'automatiser au complet la découverte de résultats en théorie des graphes, il faudrait également automatiser la démonstration des conjectures générées. Bien que la démonstration de conjectures soit, à ce jour, un processus très complexe que l'on doit faire à la main, des travaux récents, voir e.g. (Colton, 2002; Colton, 1999), donnent espoir que ce processus sera un jour fait par l'ordinateur.

RÉFÉRENCES

AGRAWAL, R., IMIELINSKI, T., et SWAMI, A. N. (1993). Mining association rules between sets of items in large databases. Dans BUNEMAN, P. et JAJODIA, S., éditeurs, *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., pages 207–216.

ALLAN, R. et LASKAR, R. (1978). On domination and independent domination numbers of a graph. *Discrete Mathematics*, **23**, 73–76.

APPEL, K., HAKEN, W., et KOCH, J. (1977). Every planar map is four colorable. *Illinois : Journal of Mathematics*, **21**, 439–567.

ASAI, T., ABE, K., KAWASOE, S., ARIMURA, H., SAKAMOTO, H., et ARIKAWA, S. (2002). Efficient substructure discovery from large semi-structured data. Dans *Proc. of the 2nd Annual SIAM Symposium on Data Mining*, pages 158–174.

BAILEY, D. H. (2000). Integer relation detection. *Computing in Science and Engineering*, **2**(1), 24–28.

BARRUS, M. D. (2004). *A forbidden subgraph characterization problem and a minimal-element subset of universal graph classes.* Thèse de Ph.D., Brigham Young University.

BEINEKE, L. (1970). Characterizations of derived graphs. *Journal of Combinatorial Theory*, **9**, 129–135.

BERGE, C. (1963). Perfect graphs. Dans INSTITUTE, C. I. S., éditeur, *Six papers on graph theory : 1-21.*

BOLLOBAS, B. et COCKAYNE, E. (1979). Graph-theoric parameters concerning domination, independence and irredundance. *Graph Theory*, **3**, 241–249.

BORGELT, C., BERTHOLD, M. R., et PATTERSON, D. E. (2005). Molecular fragment mining for drug discovery. Dans *Proc. of Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 8th European Conference*, volume 3571 de *Lecture Notes in Computer Science*, Barcelona, Spain, pages 1002–1013. Springer.

BRANDSTÄDT, LE, V., SZYMCZAK, T., SIEGEMUND, F., et DE RIDDER, H. (2003). Information system on graph class inclusions v2.0. `http://wwwteo.informatik.uni-rostock.de/isgci/index.html`.

BRANDSTÄDT, A., LE, V. B., et SPINRAD, J. P. (1999). *Graph classes : A survey*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

BRATKO, I. (2001). *Prolog programming for artificial intelligence*. Addison-Wesley, 3ème édition.

BRIGHAM, R. et DUTTON, R. (1983). Ingrid : A software tool for extremal graph theory research. *Congressus Numerantium*, **39**, 337–352.

BRIGHAM, R. et DUTTON, R. (1985). A compilation of relations between graph invariants. *Networks*, **15**, 73–107.

BRIGHAM, R. et DUTTON, R. (1991). A compilation of relations between graph invariants, supplement 1. *Networks*, **21**, 421–455.

BRIGHAM, R., DUTTON, R., et F., G. (1989). Ingrid : A graph invariant manipulator. *J. Symbolic Computation*, **7**, 163–177.

BUNTINE, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, **36**(2), 149–176.

CAPOROSSI, G., CVETKOVIC, D., GUTMAN, I., et HANSEN, P. (1999a). Variable neighborhood search for extremal graphs 2 : Finding graphs with extremal energy. *Journal of Chemical Information and Computer Sciences*, **39**, 984–996.

CAPOROSSI, G., GUTMAN, I., et HANSEN, P. (1999b). Variable neighborhood search for extremal graphs 4 : Chemical trees with extremal connectivity index. *Computers and Chemistry*, **23**, 469–477.

CAPOROSSI, G. et HANSEN, P. (2000). Variable neighborhood for extremal graphs 1 : The system AutoGraphiX. *Discrete Mathematics*, **212**, 29–44.

CAPOROSSI, G. et HANSEN, P. (2004). Variable neighborhood search for extremal graphs 5 : Three ways to automate finding conjectures. *Computational Intelligence*, **276**, 81–94.

CARRINGTON, P. J., SCOTT, J., et WASSERMAN, S., éditeurs (2005). *Models and methods in social network analysis*. Structural Analysis in the Social Sciences. Cambridge University Press.

CHAKRABARTI, S. (2002). *Mining the Web : Discovering knowledge from hypertext data*. Morgan-Kauffman.

CHEN, Y., YANG, L. H., et WANG, Y. G. (2004). Incremental mining of frequent xml query patterns. Dans *Proc. of 4th IEEE Int. Conf. on Data Mining (ICDM'04)*, Los Alamitos, CA, USA, pages 343–346. IEEE Computer Society.

CHRISTOPHE, J., DEWEZ, S., DOIGNON, J., ELLOUMI, S., FASBENDER, G., GRÉGOIRE, P., HUYGENS, D., LABBÉ, M., MÉLOT, H., et YAMAN, H. (2008). Linear inequalities among graph invariants : Using GraPHedron to uncover optimal relationships. *Networks*. Accepted for publication.

CHUDNOVSKY, M., ROBERTSON, N., SEYMOUR, P., et THOMAS, R. (2006). The strong perfect graph theorem. *Annals of Mathematics*, **164**, 51–229.

COHEN, M. et GUDES, E. (2004). Diagonally subgraphs pattern mining. Dans *DMKD '04 : Proc. of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 51–58. ACM.

COLTON, S. (1999). Refactorable numbers – a machine invention. *Journal of Integer Sequences*, **2**.

COLTON, S. (2002). The HR program for theorem generation. Dans *Proc. of the Eighteenth Conf. on Automated Deduction.*

COLTON, S., BUNDY, A., et WALSH, T. (1999). Automatic concept formation in pure mathematics. Dans *Proc. of IJCAI 1999*, pages 786–793.

COOK, D. J. et HOLDER, L. B. (1994). Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, **1**, 231–255.

CORDELLA, L. P., FOGGIA, P., SANSONE, C., et VENTO, M. (2004). A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **26**(10), 1367–1372.

CVETKOVIC, D., KRAUS, L., et SIMIC, S. (1981). Discussing graph theory with a computer I : Implementation of graph theoretic algorithms. Rapport technique, Univ. Beograd Publ. Eletrotehn. Fak.

CVETKOVIC, D. et PEVAC, I. (1983a). Discussing graph theory with a computer II : Theorems suggested by the computer. *Publ. Inst. Math. (Beograd)*, **33**(47), 29–33.

CVETKOVIC, D. et PEVAC, I. (1983b). Discussing graph theory with a computer III : Machine theorem proving. *Publ. Inst. Math. (Beograd)*, **34**(48), 37–47.

CVETKOVIC, D., SIMIC, S., CAPOROSSI, G., et HANSEN, P. (2001). Variable neighborhood search for extremal graphs 3 : On the largest eigenvalue of color-constrained trees. *Linear and Multilinear Algebra*, **49**(2), 143–160.

DEHASPE, L. et TOIVONEN, H. (1999). Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, **3**(1), 7–36.

DELAVINA, E., FAJTLOWICZ, S., et WALLER, B. (2005). On some conjectures of Griggs and Graffiti. *Graphs and Discovery DIMACS : Series in Discrete Mathematics and Theoretical Computer Science*, **69**, 119–125.

DESHPANDE, M., KURAMOCHI, M., et KARYPIS, G. (2002). Automated approaches for classifying structures. Dans *Proc. of the 2002 Workshop on Data Mining in Bioinformatics (BIOKDD'02)*, Edmonton, Canada, pages 11–18.

DESROSIERS, C., GALINIER, P., HANSEN, P., et HERTZ, A. (2007a). Automated generation of conjectures on forbidden subgraph characterization. Rapport technique G-2007-48, Les Cahiers du GERAD.

DESROSIERS, C., GALINIER, P., HANSEN, P., et HERTZ, A. (2007b). Sygma : Reducing symmetry in graph mining. Rapport technique G-2007-12, Les Cahiers du GERAD.

DESROSIERS, C., GALINIER, P., HANSEN, P., et HERTZ, A. (2008). Using heuristics to speed up frequent pattern mining. Rapport technique G-2008-13, Les Cahiers du GERAD.

DŽEROSKI, S. (2003). Multi-relational data mining : an introduction. *SIGKDD Ex-*

ploration Newsletter, **5**(1), 1–16.

DZEROSKI, S. et LAVRAC, N. (1994). *Inductive logic programming : Techniques and applications*. Ellis Horwood.

EPSTEIN, S. (1988). Learning and discovery : One system's search for mathematical knowledge. *Computational Intelligence*, **4**(1), 42–53.

EPSTEIN, S. et SRIDHARAN, N. (1991). Knowledge representation for mathematical discovery : Three experiments in graph theory. *Journal of Applied Intelligence*, **1**, 7–33.

FAJTLOWICZ, S. (1987). On conjectures of Graffiti, II. *Congressus Numerantium*, **60**, 189–197.

FAJTLOWICZ, S. (1988a). On conjectures of Graffiti. *Discrete Mathematics*, **72**, 113–118.

FAJTLOWICZ, S. (1988b). On conjectures of Graffiti, III. *Congressus Numerantium*, **66**, 23–32.

FAJTLOWICZ, S. (1990). On conjectures of Graffiti, IV. *Congressus Numerantium*, **70**, 231–240.

FAJTLOWICZ, S. (1995). On conjectures of Graffiti, V. Dans *Proc. of the Quadrennial Conf. on the Theory and Applications of Graphs*, volume 1, pages 367–376.

FAJTLOWICZ, S. (2008). Written on the Wall. `http://math.uh.edu/~clarson/wow.zip`.

FAJTLOWICZ, S. et DELAVINA, E. (2008). Written on the Wall - II. `http://cms.dt.uh.edu/faculty/delavinae/research/wowII`.

FAUDREE, R., FLANDRIN, E., et RYJACEK, Z. (1997). Claw-free graphs – A survey. *Discrete Mathematics, Journal of graph theory*, **169**(4), 87–147.

FAVARON, O. (1986). Stability, domination and irredundance in a graph. *Journal of Graph Theory*, **10**, 429–438.

FOLDES, S. et HAMMER, P. L. (1977). Split graphs. Dans *Proc. of the Eighth Southeastern Conf. on Combinatorics, Graph Theory and Computing*, pages 311–315. Congressus Numerantium.

FORTIN, S. (1996). The graph isomorphism problem. Rapport technique 96-20, University of Alberta, Edomonton, Alberta, Canada.

GAREY, M. R. et JOHNSON, D. S. (1990). *Computers and intractability : A guide to the theory of NP-completeness.* W. H. Freeman & Co., New York, NY, USA.

GREENWELL, D. L., HEMMINGER, R., et KLERLEIN, J. (1973). Forbidden subgraphs. Dans *Proc. of the Fourth Southeastern Conf. on Combinatorics, Graph Theory and Computing*, pages 389–394. Congressus Numerantium.

HAMMER, P. L. et SIMEONE, B. (1981). The splittance of a graph. *Combinatorica*, **1**(3), 275–284.

HANSEN, P. (2005). How far is, should and could be conjecture-making in graph theory an automated process ? Dans FAJTLOWICZ, S., FOWLER, P. W., HANSEN, P., JANOWITZ, M. F., et ROBERTS, F. S., éditeurs, *Graphs and Discovery, volume 69 of DIMACS : Series in discrete mathematics and theoretical computer science*, pages 180–230. American Mathematical Society, DIMACS.

HANSEN, P., AOUCHICHE, M., CAPOROSSI, G., MÉLOT, H., et STEVANOVIC, D. (2005). What forms do interesting conjectures have in graph theory ? *Graphs and Dis-*

covery, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, **69**, 231–252.

HAYNES, T. W., HEDETNIEMI, S., et SLATER, P. (1998). *Fundamentals of domination in graphs*. Marcel Dekker.

HU, H., YAN, X., YU, H., HAN, J., et ZHOU, X. (2005). Mining coherent dense subgraphs across massive biological networks for functional discovery. Dans *Proc. of 2005 Int. Conf. Intelligent Systems for Molecular Biology (ISMB'05)*, Ann Arbor, MI, pages 213–221.

HUAN, J., WANG, W., BANDYOPADHYAY, D., SNOEYINK, J., PRINS, J., et TROPSHA, A. (2004a). Mining spatial motifs from protein structure graphs. Dans *Proc. of 8th Int. Conf. Research in Computational Molecular Biology (RECOMB)*, San Diego, CA, pages 308–315.

HUAN, J., WANG, W., PRIN, J., et YANG, J. (2004b). Spin : mining maximal frequent subgraphs from graph databases. Dans *KDD '04 : Proc. of the tenth ACM SIGKDD int. conf. on Knowledge discovery and data mining*, New York, NY, USA, pages 581–586. ACM.

HUAN, J., WANG, W., et PRINS, J. (2003). Efficient mining of frequent subgraph in the presence of isomorphism. Dans *Proc. of the 3rd IEEE Int. Conf. on Data Mining (ICDM)*, pages 549–552.

INOKUCHI, A., WASHIO, T., et MOTODA, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. Dans *Proc. of the 4th European Conf. on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer-Verlag.

KIM, D. H., YUN, I. D., et LEE, S. U. (2006). Boundary-trimmed 3d triangular mesh segmentation based on iterative merging strategy. *Pattern Recognition*, **39**(5), 827–838.

KOYUTURK, M., GRAMA, A., et SZPANKOWSKI, W. (2004). An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, **2**, 200–207.

KRAMER, S., DE RAEDT, L., et HELMA, C. (2001). Molecular feature mining in HIV data. Dans *Proc. of the seventh ACM SIGKDD int. conf. on Knowledge discovery and data mining (KDD '01)*, New York, NY, USA, pages 136–143. ACM.

KURAMOCHI, M. et KARYPIS, G. (2001). Frequent subgraph discovery. Dans *Proc. of the First IEEE Conf. on Data Mining*, pages 313–320.

KURAMOCHI, M. et KARYPIS, G. (2005). Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, **11**(3), 243–271.

LARSON, C. (2002). Intelligent machinery and mathematical discovery. Dans *Graph Theory Notes of the New York Academy of Science, XLII*, pages 8–17.

LENAT, D. (1979). On automated scientific theory formation : A case study using the AM program. *Machine Intelligence*, **9**, 251–283.

LENAT, D. (1984). Automated theory formation in mathematics. Dans BLEDSOE et LOVELAND, D., éditeurs, *Automated Theorem Proving : After 25 Years*, pages 287–314. American Mathematical Society.

LIU, B. (2007). *Web data mining : Exploring hyperlinks, contents, and usage data.* Data-Centric Systems and Applications. Springer.

MANNILA, H. et TOIVONEN, H. (1997). Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, **1**(3), 241–258.

MCCUNE, W. (1997). Solution of the Robbins problem. *Journal of Automated Reasoning*, **19**(3), 263–276.

McKay, B. (1981). Practical graph isomorphism. *Congressus Numeratium*, **30**, 45–87.

McKay, B. (1998). Isomorph-free exhaustive generation. *Journal of Algorithms*, **26**, 306–324.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence*, **18**(2), 203–226.

Mladenovic, N. et Hansen, P. (1997). Variable neighborhood search. *Computers and Operations Research*, **24**(11), 1097–1100.

Muggleton, S. (1992). *Inductive logic programming*. Academic Press.

Mélot, H. (2007). Facet defining inequalities among graph invariants : the system GraPHedron. *Discrete Applied Mathematics*. In press.

Nakatsuji, A., Sugaya, Y., et Kanatani, K. (2005). Optimizing a triangular mesh for shape reconstruction from images. *IEICE - Transactions on Information and Systems*, **E88-D**(10), 2269–2276.

Nijssen, S. et Kok, J. N. (2001). Faster association rules for multiple relations. Dans *IJCAI*, pages 891–896.

Nijssen, S. et Kok, J. N. (2004). The gaston tool for frequent subgraph mining. Dans *Proc. of the Int. Workshop on Graph-Based Tools (Grabats 2004)*, pages 281–285. Elsevier.

Page, D. L., Koschan, A., et Abidi, M. A. (2003). Perception-based 3d triangle mesh segmentation using fast marching watersheds. Dans *Proc. of 2003 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR 2003)*, Madison, WI, USA, pages 27–32.

PEI, J., HAN, J., MORTAZAVI-ASL, B., PINTO, H., CHEN, Q., DAYAL, U., et HSU, M. (2001). PrefixSpan : Mining sequential patterns efficiently by prefix-projected pattern growth. Dans *Proc. of the 17th Int. Conf. on Data Engineering (ICDE '01)*, Washington, DC, USA, pages 215–224. IEEE Computer Society.

PLOTKIN, G. (1969). A note on inductive generalization. *Machine Intelligence*, **5**, 153–163.

PUECH, J. (1998). Irredundance perfection and p6-free graphs. *Journal of Graph Theory*, **29**, 239–255.

PUNIN, J., KRISHNAMOORTHY, M., et ZAKI, M. J. (2001). Web usage mining : languages and algorithms. Dans *Studies in Classification, Data Analysis, and Knowledge Organization*. Springer-Verlag.

REITTU, H. et NORROS, I. (2007). Random graph models of communication network topologies.

ROBERTSON, N., SANDERS, D., SEYMOUR, P., et THOMAS, R. (1997). The four-colour theorem. *Journal of Combinatorial Theory*, **70**(B 1), 2–44.

SCHMIDT, D. C. et DRUFFEL, L. E. (1976). A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM*, **23**(3), 433–445.

SRINIVASAN, A., KING, R. D., MUGGLETON, S. H., et STERNBERG, M. (1997). The predictive toxicology evaluation challenge. Dans *Proc. of the Fifteenth Int. Joint Conf. on Artificial Intelligence (IJCAI-97)*, pages 1–6. Morgan-Kaufmann.

STERNBERG, M. J. E., KING, R. D., SRINIVASAN, A., et MUGGLETON, S. (1995). Drug design by machine learning. Dans *Machine Intelligence 15*, pages 328–338.

TERMIER, A., ROUSSET, M.-C., et SEBAG, M. (2002). Treefinder : a first step towards xml data mining. Dans *Proc. of Int. Conf. on Data Mining ICDM'02*, Maebashi, Japan, pages 450–457.

TERMIER, A., ROUSSET, M.-C., et SEBAG, M. (2004). Dryade : A new approach for discovering closed frequent trees in heterogeneous tree databases. Dans *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004)*, pages 543–546. IEEE Computer Society.

ULLMANN, J. (1976). An algorithm for subgraph isomorphism. *J. ACM*, **23**(1), 31–42.

VOLKMANN, L. et ZVEROVICH, V. (2002). A proof of a conjecture on irredundance perfect graphs. *Journal of Graph Theory*, **41**, 292–306.

WANG, C., HONG, M.-S., WANG, W., et SHI, B.-L. (2004a). Chopper : efficient algorithm for tree mining. *Journal of Computer Science and Technology*, **19**(3), 309–319.

WANG, C., WANG, W., PEI, J., ZHU, Y., et SHI, B. (2004b). Scalable mining of large disk-based graph databases. Dans *Proc. of the tenth ACM SIGKDD int. conf. on Knowledge discovery and data mining (KDD '04)*, New York, NY, USA, pages 316–325. ACM.

WANG, C., ZHU, Y., WU, T., WANG, W., et SHI, B. (2005a). Constraint-based graph mining in large database. Dans *Web Technologies Research and Development - APWeb 2005*, pages 133–144.

WANG, J., ZAKI, M., TOIVONEN, H., et SHASHA, D., éditeurs (2005b). *Data mining in bioinformatics*. Springer.

WASSERMAN, S. et FAUST, K. (1994). *Social network analysis*. Cambridge University

Press, Cambridge.

WONG, E. (1992). Model matching in robot vision by subgraph isomorphism. *Pattern Recognition*, **25**(3), 287–303.

WÖRLEIN, M., MEINL, T., FISCHER, I., et PHILIPPSEN, M. (2005). A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. Dans *PKDD*, pages 392–403.

WOS, L. (1996). *The automation of reasoning : An experimenter's notebook with OTTER tutorial.* Academic Press Professional, Inc., San Diego, CA, USA.

XIA, Y. et YANG, Y. (2005). Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering*, **17**(2), 190–202.

YAN, X. et HAN, J. (2002). gSpan : Graph-based substructure pattern mining. Dans *Proc. of the Int. Conf. on Machine Learning (ICML 2002)*, pages 721–724.

YAN, X. et HAN, J. (2003). CloseGraph : Mining closed frequent graph patterns. Dans *Proc. of Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, Washington, DC, pages 286–295.

YAN, X., YU, P., et HAN, J. (2005). Substructure similarity search in graph databases. Dans *Proc. of 2005 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'05)*, Baltimore, MD, pages 766–777.

YANG, L. H., LEE, M. L., et HSU, W. (2003). Efficient mining of xml query patterns for caching. Dans *Proc. of the 29th int. conf. on Very large data bases (VLDB'2003)*, pages 69–80. VLDB Endowment.

YOSHIDA, K. et MOTODA, H. (1995). CLIP : concept learning from inference patterns. *Artificial Intelligence*, **75**(1), 63–92.

ZAKI, M. J. (2002). Efficiently mining frequent trees in a forest. Dans *KDD '02 : Proc. of the eighth ACM SIGKDD int. conf. on Knowledge discovery and data mining*, New York, NY, USA, pages 71–80. ACM.

ZAKI, M. J., NADIMPALLY, V., BARDHAN, D., et BYSTROFF, C. (2004). Predicting protein folding pathways. *Bioinformatics*, **20**(1), 386–393.

ZHU, F., YAN, X., HAN, J., et YU, P. (2007). gPrune : A constraint pushing framework for graph pattern mining. Dans *Proc. 2007 Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'07)*, Nanjing, China, pages 388–400.

ZHU, H., ZANG, H., ZHU, K., et MUKHERJEEM, B. (2003). A novel generic graph model for traffic grooming in heterogeneous WDM mesh networks. *IEEE/ACM Transactions on Networking*, **11**(2), 285–299.