UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE
OFFERTE À
l'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
EN VERTU D'UN PROTOCOLE D'ENTENTE
AVEC L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL

par

Martin Charlton

Fragmentation de graphes et applications au génie logiciel.

Décembre 2005

Bibliothèque Paul-Émile-Boulet

UQAC Université du Québec à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Résumé.

Ce mémoire porte sur la fragmentation de graphes et ses applications au génie logiciel. Après une présentation du domaine d'application et de la problématique, nous traitons d'un certain nombre d'algorithmes de fragmentation. Ces algorithmes, issus de la littérature scientifique, sont analysés selon notre problématique. Pour répondre à nos besoins, nous introduisons la fragmentation par noyaux. Cette approche produit une partition en identifiant des sommets jugés centraux à des sous-graphes possibles afin de construire des noyaux. Les différents sommets du graphe sont ensuite distribués dans la partition selon leur relation avec les noyaux. La dernière partie de ce mémoire aborde la mise en oeuvre et l'utilisation de la fragmentation par noyaux. La méthode est appliquée au génie logiciel. Nous obtenons des partitions sur des graphes de systèmes logiciels. Les partitions sont analysées pour trouver les points forts et les défaillances de la fragmentation par noyaux, ainsi que les améliorations possibles.

Remerciements.

J'ai toujours trouvé ridicules les vedettes qui, lors de galas, s'empêtrent dans leurs remerciements, ne savent plus de qui parler et nomment plein de gens tout en donnant l'impression d'en oublier la moitié. Maintenant que j'ai à écrire des remerciements, je comprends un peu mieux pourquoi... et je m'excuse envers elles.

Donc, en premier lieu, je dois remercier mon directeur, Yves Chiricota. C'est peut-être cliché, mais c'est quand même lui qui a été mon guide et patron dans les temps amusants (la recherche) et moins amusants (la rédaction) de ces dernières années. Je tiens aussi à remercier ces personnes dont l'apport a été crucial : mes parents (un autre cliché) et mes collègues Michel et Alexandre, qui ont bénévolement donné de leur temps à plus d'une occasion. Afin de ne pas sortir des clichés, je remercie mon comité de lecture. Ce n'est pas par complaisance, mais simplement parce qu'ils vont mettre plusieurs heures à réviser mon travail.

Pour ce qui est des autres personnes pour qui je sens une certaine obligation maintenant que je vois tout le chemin parcouru... Il y a d'abord mes collègues du GRI, en particulier François, Éric, Éric, Simon, Sarah, Bruno, Yannick, Jean-Luc et Nancy. Ils m'ont apporté une belle ambiance et de l'aide durant mon labeur. Il y a aussi les membres du projet Lyb (Éric, Jean-Michel, Fred, Mathieu, Samuel...) qui m'ont permis de vider mon fiel lorsque les choses n'allaient pas comme je voulais. Finalement, un dernier merci pour mes amis de Tomo qui, sans avoir aidé, m'ont quand même apporté beaucoup dans les derniers mois.

Si jamais j'oublie une personne, et ceux qui connaissent ma mémoire comprennent

que c'est très probable, je m'en excuse et lui envoie télépathiquement ma gratitude.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

Liste des Algorithmes

INTRODUCTION.

Un graphe est un objet mathématique formé de noeuds et d'arêtes, les arêtes reliant certains des noeuds entre eux. Les graphes sont utilisés pour représenter l'information relative à plusieurs domaines. À partir du moment où l'on est en présence d'entités en relations les unes avec les autres, il est naturel d'encoder l'information sous forme de graphes. Par exemple, en biologie, des graphes représentent les « voies métaboliques » (figure 1). Les noeuds de ces graphes correspondent à des enzymes et on place un arc d'un noeud vers un autre si les enzymes associées à ces noeuds sont impliquées dans des réactions chimiques communes. On a recours aux graphes pour illustrer l'information inhérente à plusieurs autres domaines, dont la sociologie (graphes de relations entre individus), la chimie (représentations moléculaires), etc...

Dans le cadre de notre travail, nous nous consacrerons à certaines familles de graphes issues du domaine du génie logiciel. Bien que l'on retrouve des graphes en conception de logiciels, ceux qui nous intéressent plus particulièrement sont les graphes issues de l'ingénierie inverse et de la maintenance de logiciels. Avec le temps, un logiciel peut se retrouver avec une documentation incomplète, inadéquate ou même parfois absente. Dans de tels cas, le code source devient une source d'informations indispensable aux programmeurs. Il existe plusieurs manières d'exprimer l'information contenue dans le code source sous forme de graphes. Comme un code source peut contenir des centaines de milliers, voire des millions de lignes de codes, les graphes produits tendent à être gros et il faut trouver des moyens d'assister le programmeur

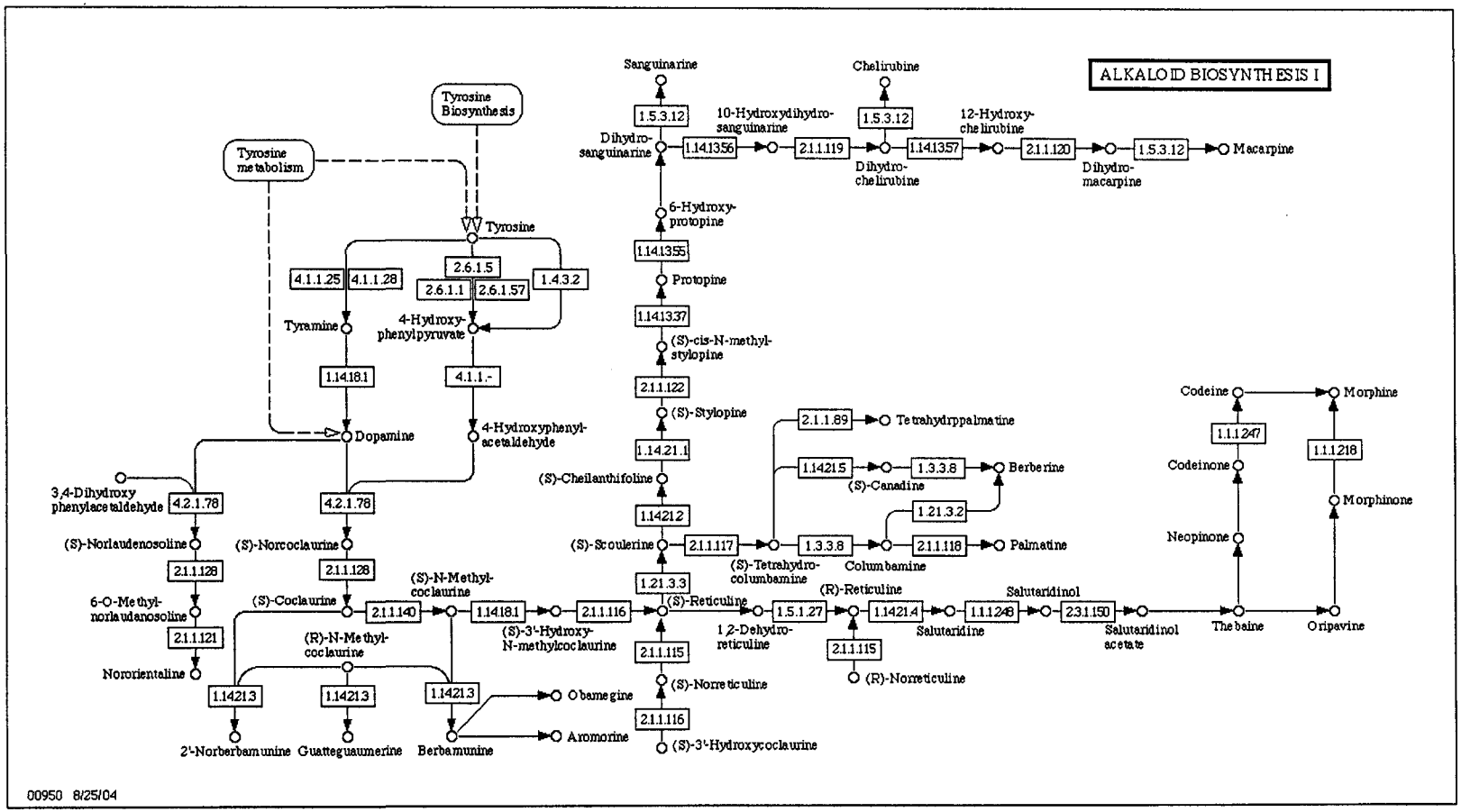


FIG. 1 – Voies métaboliques de la biosynthèse des alcaloïdes (Image tirée de KEGG [40, 41]).

dans leur interprétation. Cette étude porte sur l'analyse et l'introduction d'outils servant à visualiser et à comprendre l'information encodée par des graphes de grande taille (plusieurs centaines de noeuds).

Il est possible de simplifier un graphe par des regroupements dans ces noeuds. Une technique pour obtenir de tels regroupement consiste à fragmenter le graphe. Ainsi, chaque sous-graphe de la partition obtenue est un regroupement de noeuds du graphe de départ. Le graphe quotient de la partition, qui sert à illustrer les relations entre sous-graphes, devient une forme d'abstraction du graphe de départ.

Plusieurs algorithmes de fragmentation, issues d'une grande variété de domaines d'application, sont documentées dans la littérature scientifiques. Cette étude porte un regards sur certains d'entre-eux. Nous déterminerons s'ils produisent des partitions qui rencontrent nos critères d'une bonne partition par rapport à notre objectif de simplification de graphes du génie logiciel. Nous tenterons par la suite de créer notre propre algorithme, dans l'espoir que celui-ci comble mieux nos objectifs que les algorithmes étudiés.

Cette étude débutera par un survol de notions de la théorie des graphes qui s'avèreront nécessaires à la bonne compréhension du lecteur. En premier lieu, il y aura les définitions les plus élémentaires. Par la suite, différents concepts liés à la connectivité et aux distances dans le graphe seront introduits. Nous verrons aussi la fragmentation de graphes, des principes de visualisation des grands graphes ainsi que d'autres exemples de domaines d'applications des graphes.

La deuxième partie de l'étude porte sur les problèmes de compréhension de l'information en ingénierie inverse. On y verra pourquoi il est nécessaire d'effectuer de la maintenance dans un système logiciel ainsi que les conséquences à long terme de cette maintenance sur le logiciel et sur sa documentation. Nous ferons le lien entre la problématique d'ingénierie inverse et la théorie des graphes, et fixerons nos objectifs par rapport à la fragmentation de graphes.

En troisième partie, nous regarderons les algorithmes de la littérature mentionnés plus haut et nous en ferons l'analyse par rapport aux objectifs que nous nous serons fixés.

Finalement, les deux dernières parties traiterons de l'élaboration et la mise-en-

oeuvre de notre propre algorithme de fragmentation de graphes : la fragmentation par noyaux. Nous commencerons par une explication des différentes idées et des concepts de cet algorithme. Nous poursuivrons avec son fonctionnement détaillé. Par la suite, nous verrons son implémentation dans un système logiciel et quelques cas expérimentaux. Finalement, nous terminerons avec une analyse des résultats obtenus.

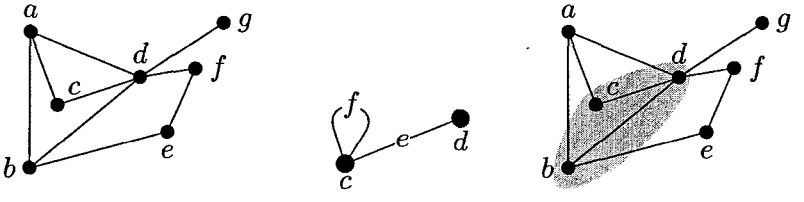
CHAPITRE 1

ÉLÉMENTS DE LA THÉORIE DES GRAPHES.

Les graphes étantà la base même de ce travail, nous leur consacrerons un chapitre. La théorie des graphes est une branche très vaste des mathématiques. Nous ne présenterons ici que les notions nécessaires à cette étude, définies à partir de plusieurs sources ([9, 10, 47]). Nous commencerons par des définitions de base, avant de voir la fragmentation des graphes ainsi que leur affichage. Ensuite, nous démontrerons l'utilité des graphes dans l'affichage d'informations structurées à l'aide de quelques exemples.

1.1 Quelques définitions.

Tout d'abord, distinguons paire d'éléments et couple d'éléments. Une *paire* est un ensemble de deux éléments [1] alors qu'un *couple* est une liste ordonnée de deux éléments. Un *graphe* G est un couple (V, E) où l'ensemble V contient les *sommets* (noeuds) du graphe et où l'ensemble E contient des paires de noeuds appelées des *arêtes*. Le nombre de sommets d'un graphe (la cardinalité de V) est son *ordre*. La figure 1.1(a) illustre un graphe.

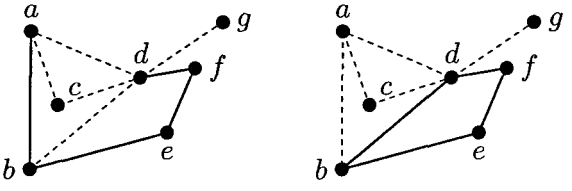


(a) Un graphe. (b) Arêtes. (c) Voisinage de a.

FIG. 1.1 – Définitions.

On dit d'un sommet qu'il est *incident* à une arête s'il est à l'une des extrémités de celle-ci. Une arête est incidente aux sommets qu'elle contient. Si elle est incidente à un

seul sommet (le même sommet constituant les deux extrémités), l'arête est appelée boucle. Prenons pour exemple la figure 1.1(b). Les sommets c et d sont incidents à e et l'arête e est incidente à c et à d. Le sommet c est incident à l'arête f, qui est alors elle-même incidente à c et est aussi une boucle. Deux sommets incidents à une même arête sont *adjacents* ou *voisins*, donc c est voisin de d et vice-versa. L'ensemble des voisins d'un sommet s est désigné par le terme *voisinage* de s, que l'on écrit $N(s)$. Le *degré* d'un sommet est égal au nombre d'arêtes incidentes à ce sommets. Le voisinage du sommet a du graphe 1.1(a) est $\{b, c, d\}$ (figure 1.1(c)) et le degré de a est 3.



(a) Chaîne simple. (b) Cycle.

FIG. 1.2 – Définitions.

Une suite finie de sommets $(v_1, v_2, v_3, ..., v_n)$ de G telle qu'il existe toujours une arête entre v_i et v_{i+1} pour $1 \leq i < n$, est une *chaîne* d'extrémité initiale v_1 et d'extrémité finale v_n. Si toutes ses arêtes $a_i = \{v_i, v_{i+1}\}$ sont distinctes, la chaîne est dite *simple*. Une chaîne dont le même sommet est à la fois l'extrémité initiale et l'extrémité finale est un *pseudo-cycle*. Si en plus cette chaîne est simple, on l'appelle *cycle*. La figure 1.2(a) représente la chaîne simple (d, f, e, b, a) alors que la figure 1.2(b) représente le cycle (d, f, e, b, d).

Un graphe où il existe une chaîne entre chaque paire de ses sommets est un graphe *connexe*.

1.1.1 Distances et voisinages dans le graphe.

Soit $A(v_1, v_2)$, l'ensemble des chaînes possibles dans G entre $v_1 \in V$ et $v_2 \in V$. Considérons aussi $U = (u_1, u_2, ..., u_n) \in A(v_1, v_2)$, une chaîne quelconque telle que $u_1 = v_1$ et $u_n = v_2$. La longueur de U, $l(U)$, est son nombre d'arêtes. Si U est la plus courte chaîne de $A(v_1, v_2)$ (donc la plus courte entre v_1 et v_2), ont dit que $l(U)$ est la *distance* entre v_1 et v_2. Si

une telle chaîne n'existe pas, la distance est infinie. La distance entre un sommet et lui-même est toujours de 0 (formule (1.1)).

$$d(v_1, v_2) = \begin{cases} 0 & \text{si } v_1 = v_2 \\ \infty & \text{si } A = \emptyset \\ \min_{U \in A} l(U) & \text{sinon} \end{cases} \tag{1.1}$$

La distance se mesure aussi entre un sommet et un groupe de sommets. Pour le sous-ensemble $V' \subseteq V$, la distance entre $v \in V$ et V' s'écrit $d_G(v, V')$. Sa valeur égale la plus petite distance entre v et un sommet de V' :

$$d_G(v, V') = \min_{u \in V'} d(v, u) \tag{1.2}$$

Regardons quelques caractéristiques liées à la distance dans un graphe connexe $G = (V, E)$. Considérons la distance maximale entre un sommet v et les autres sommets de V :

$$R(v) = \max_{w \in V} d(v, w) \tag{1.3}$$

Une valeur relativement petite de $R(v)$ implique un sommet central (près du centre) au graphe. Une valeur relativement grande signifie un sommet périphérique [11]. La valeur minimale de $R(v)$ parmi tous les sommets de V :

$$R_0 = \min_{v \in V} R(v) \tag{1.4}$$

est appelée rayon du graphe et un sommet v_0, tel que $R(v_0) = R_0$, est un centre du graphe. Un graphe peut avoir plusieurs centres. Dans un graphe complet, tous les sommets sont des centres. La distance maximale entre toutes les paires de sommets, calculée comme suit :

$$D(G) = \max_{v,w \in V} d(v, w) = \max_{v \in V} R(v) \tag{1.5}$$

est appelée le diamètre du graphe. Pour un graphe non-connexe, le diamètre est infini.

En utilisant le concept de distance, *Henning* [35] donne une définition élargie du voisinage d'un sommet. En plus de notre graphe G, considérons un entier positif non-nul k. Le *k-voisinage* d'un sommet v de G, ou $N_k(v)$, est l'ensemble des sommets de $V-v$ à une distance d'au plus k de v. Par exemple, pour l'illustration 1.3, le 2-voisinage de d est $\{b, c, e, f\}$. Quand on parle du voisinage d'un sommet (sans l'entier), cela désigne son 1-voisinage. Le k-voisinage *clos* d'un sommet v, ou $N_k[v]$, inclut v avec son k-voisinage. Dans l'exemple précédent, le 2-voisinage clos de d est $\{b, c, d, e, f\}$

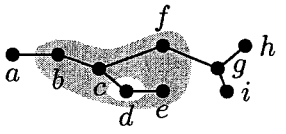


FIG. 1.3 – Le 2-voisinage de d.

Ces définitions peuvent s'étendre à des sous-ensembles de sommets. Considérons un sous-ensemble $V' \subset V$ d'un graphe $G = (V, E)$. On définit $N_k(V')$, le k-voisinage de V', comme étant l'ensemble suivant :

$$N_k(V') = (\bigcup_{v \in V'} N_k(v)) - V' \tag{1.6}$$

et $N_k[V']$, le k-voisinage clos de V', par :

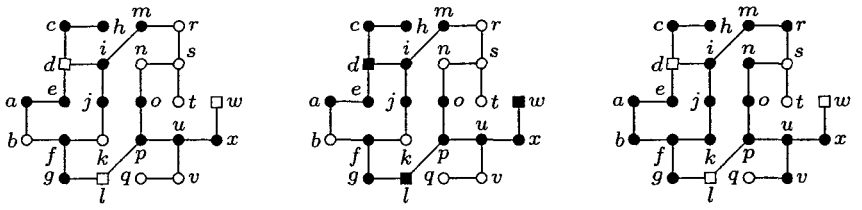
$$N_k[V'] = (\bigcup_{v \in V'} N_k[v]) \tag{1.7}$$

La figure 1.4 illustre les deux définitions précédentes avec $V' = \{d, l, w\}$, les sommets pleins étant dans les voisinages illustrés.

1.1.2 Quelques variétés de graphes.

Un graphe qui n'a pas de boucles :

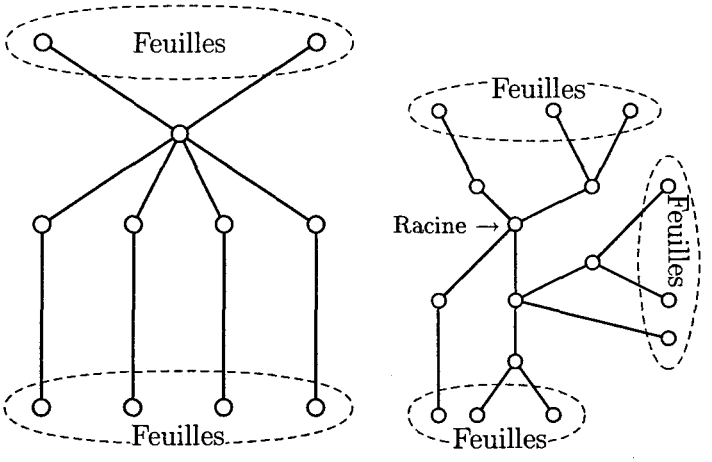
$$\{x, y\} \in E \Rightarrow x \neq y \tag{1.8}$$



(a) $N_2(V')$. (b) $N_2[V']$. (c) $N_3(V')$.

FIG. 1.4 – Quelques k-voisinages du sous-ensemble $V' = \{d, l, w\}$.

est dit *simple*. Un graphe connexe sans cycles est un *arbre* et ses noeuds de degré 1 sont ses feuilles (figure 1.5(a)). Une *arborescence* est un arbre dont l'un des sommets est pointé(figure 1.5(b)). Ce sommet est la *racine* de l'arborescence.



(a) Un arbre. (b) Une arborescence.

FIG. 1.5 – Exemples d'arbres.

Un graphe *complet* est un graphe qui contient une arête pour chaque paire de sommets :

$$x, y \in V \Leftrightarrow \{x, y\} \in E \tag{1.9}$$

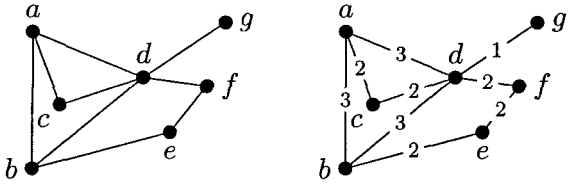
Soit L un ensemble d'étiquettes. Un graphe $G = (V, E)$ est *étiqueté* s'il existe une fonction $\lambda : V \to L$ ou $\lambda : E \to L$ qui associe une étiquette à chaque sommet ou à chaque arête.

Deux graphes $G = (V, E)$ et $G' = (V', E')$ sont *isomorphes* s'il existe un isomorphisme de G à G'. Un *isomorphisme* est une fonction bijective $\psi : V \rightarrow V'$ telle que pour deux sommets u et v de V :

$$\{u, v\} \in E \Leftrightarrow \{\psi(u), \psi(v)\} \in E' \tag{1.10}$$

Graphe valué.

Un graphe est *valué* (ou *pondéré*) lorsqu'une fonction $\gamma : E \rightarrow \Re$, dite *fonction de coût* ou *valuation*, associe à chaque arête du graphe une valeur numérique, le *coût* (ou *poids*) [47]. Par exemple, supposons une fonction $d^- : E \rightarrow \Re$ qui donne à une arête une valeur égale au plus petit degré de ses sommets incidents. Si l'on applique la valuation d^- sur le graphe 1.6(a), on obtient le graphe valué 1.6(b).



(a) Un graphe G. (b) G valué par d^-.

FIG. 1.6 – Graphe valué.

La longueur pondérée d'une chaîne $U = (u_1, u_2, u_3, ..., u_n)$ d'un graphe valué $G = (V, E)$ est égale à la somme du poids des arêtes de cette chaîne :

$$l_\gamma(U) = \sum_{i=1}^{n-1} \gamma\{u_i, u_{i+1}\} \tag{1.11}$$

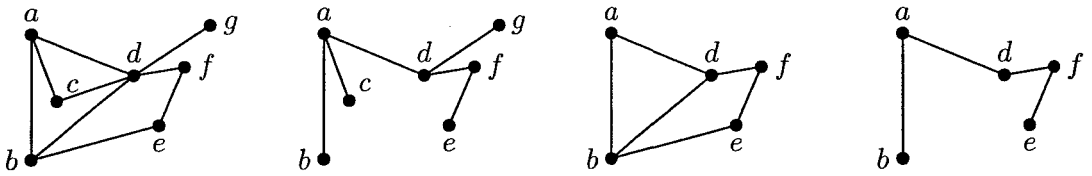
La distance pondérée entre deux sommets u et v est définie par :

$$d_\gamma(u, v) = \min\{l_\gamma(U) | U \text{ est une chaîne de } u \text{ à } v.\} \tag{1.12}$$

Par exemple, dans le graphe de la figure 1.6(b), $l(\{a, c, d, f, e\})$ vaut 8 et $d_\gamma(a, e)$ vaut 5.

Graphes dérivés.

Voici quelques graphes produits à partir d'un autre graphe. Pour les exemples, le graphe de départ sera celui de la figure 1.7(a).



(a) Graphe. (b) Graphe partiel. (c) Sous-graphe. (d) Sous-graphe partiel.

FIG. 1.7 – Un graphe et des graphes qui en sont dérivés.

$G' = (V', E')$ est un *graphe partiel* de G s'il contient exactement les mêmes sommets, mais que ses arêtes forment un sous-ensemble des arêtes de G. Autrement dit, $V' = V$ et $E' \subseteq E$. Le graphe 1.7(b) est un sous-graphe partiel du graphe 1.7(a).

Un *sous-graphe* est un graphe qui ne contient qu'une partie des noeuds d'un graphe donné, mais toutes les arêtes incidentes à deux noeuds conservés. Un graphe $G' = (V', E')$ est sous-graphe de G si $V' \subseteq V$ et si ses arêtes observent les deux propriétés suivantes :

- Une arête a de E' est dans E.
- Toute arêtes de E dont les sommets aux deux extrémités sont dans V' est dans E'.

La figure 1.7(c) montre un sous-graphe du graphe 1.7(a).

Un *sous-graphe partiel* est un graphe tel que l'ensemble de ses noeuds et l'ensemble de ses arêtes sont tous deux des sous-ensembles des noeuds et des arêtes du graphe dont il provient. Ainsi, si G' est un sous-graphe partiel de G on a $V' \subseteq V$ et $E' \subseteq E$. Un sous-graphe partiel du graphe 1.7(a) est illustré à la figure 1.7(d).

Lorsqu'un sous-graphe partiel ou un sous-graphe de G est un graphe connexe, il est désigné comme étant une *composante connexe* de G. On appelle *isthme* une arête e qui sépare un graphe connexe, ou une composante connexe d'un graphe, en deux composantes connexes. En d'autres mots, un isthme est une arête e telle que le graphe $(V, E - \{e\})$ comporte une

composante connexe de plus que le graphe G.

1.1.3 Fragmentation de graphes.

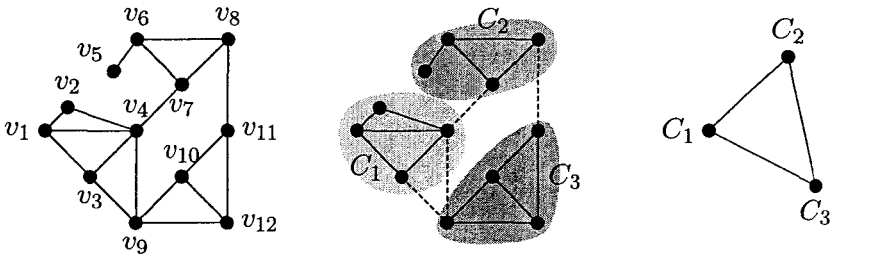
La fragmentation de graphes est le principal outil qui sera utilisé pour notre analyse des graphes. Faire la fragmentation d'un graphe, c'est répartir les sommets dans un certain nombre de sous-graphes de façon à ce que chaque noeud soit présent dans un et un seul des sous-graphes, ce qui forme une partition du graphe. Si l'ensemble des $C_i = (V_i, E_i)$, $1 \leq i \leq n$ est une partition, alors l'intersection de V_i et V_j est l'ensemble vide si $i \neq j$ et l'union des V_i donne V. En d'autres termes :

$$\bigcup_{1 \leq i \leq n} V_i = V \tag{1.13}$$

$$V_i \cap V_j \neq \emptyset \Leftrightarrow i = j \tag{1.14}$$

Étant donnée une partition, on dira des arêtes de G incidentes à deux sommets présents dans le même sous-graphe (dans le même ensemble V_k) que ce sont des arêtes *internes*, et elles vont se retrouver dans l'ensemble E_k. Les arêtes incidentes à des sommets de sous-graphes distincts sont des arêtes *externes*.

Il est possible d'exprimer une partition d'un graphe sous forme d'un autre graphe, le *graphe quotient*. Chaque sommet du graphe quotient représente un sous-graphe de la partition. On place une arête entre deux sommets s'il existe au moins une arête externe entre les sous-graphes respectifs. La figure 1.8 présente l'exemple d'un graphe, d'une partition de celui-ci et du graphe quotient correspondant. Considérons un graphe $G = (V, E)$ (figure 1.8(a)), où $V = \{v_1, v_2, ..., v_{12}\}$. Il est fragmenté en trois composantes connexes, $C_1 = \{v_1, v_2, v_3, v_4\}$, $C_2 = \{v_5, v_6, v_7, v_8\}$ et $C_3 = \{v_9, v_{10}, v_{11}, v_{12}\}$ (figure 1.8(b)). Le graphe quotient correspondant (figure 1.8(c)) compte trois sommets, un par sous-graphe, et trois arêtes. Une arête représente $\{v_8, v_{11}\}$, une autre représente $\{v_7, v_4\}$ et la dernière représente les deux arêtes entre C_1 et C_3, $\{v_3, v_9\}$ et $\{v_4, v_9\}$.



(a) Le graphe. (b) Une partition. (c) Le graphe quotient.

FIG. 1.8 – Un graphe, une partition de ce graphe et le graphe quotient.

L'espace des partitions.

La fragmentation ne produit qu'une partition particulière. L'espace des solutions possibles à la fragmentation d'un graphe est très vaste. Nous appellerons *k-partition* une partition en k sous-graphes (avec $1 \leq k \leq n$) d'un graphe G d'ordre n. Le nombre de k-partitions dans un graphe s'exprime à l'aide des nombres de Stirling de seconde espèce :

$$S_{n,k} = \begin{cases} 0 & \text{si } 1 \leq n < k \\ 0 & \text{si } k = 0 \text{ et } n > 0 \\ 1 & \text{si } k = 1 \text{ ou } k = n \\ S_{n-1,k-1} + kS_{n-1,k} & \text{sinon} \end{cases} \tag{1.15}$$

et le nombre total de partitions possibles pour un graphe s'exprime avec la formule suivante :

$$n_p = \sum_{k=1}^{n} S_{n,k} \tag{1.16}$$

Si l'on calcule quelques valeurs de n_p pour des graphes de différents ordres (le tableau 1.1 montre les graphes d'ordres 1 à 15), on remarque que l'ordre de croissance de la quantité de partitions est exponentielle. Pour des graphes de grande taille, comme ceux étudiés, l'espace des solutions est inexplorable dans son intégralité. L'algorithme recherché doit donc choisir une bonne solution dans cet espace, en espérant qu'elle se rapproche de la meilleure solution possible.

Ordre	Partitions
1	1
2	2
3	5
4	15
5	52
6	203
7	877
8	4 140
9	21 147
10	115 975
11	678 570
12	4 213 597
13	27 644 437
14	190 899 322
15	1 382 958 545

TAB. 1.1 – Partitions possibles selon l'ordre du graphe.

1.2 La visualisation de graphes.

Les graphes qui seront vus dans le cadre de cette étude sont potentiellement très gros et devront être affichés d'une manière qui les rend intelligibles à l'utilisateur. Afin d'obtenir un bon affichage, nous avons recours à des principes de la visualisation de l'information.

1.2.1 Visualisation de l'information.

De la manière la plus générique, on peut qualifier comme étant de l'information tout ce qui vient affecter les connaissances d'une ou plusieurs personnes [52]. L'informatisation, la médiatisation et l'apparition de l'Internet ont permis un accès plus facile à de vastes quantités d'informations produites par la recherche (résultats de simulations, mesure d'instruments...). Plus d'accès à l'information implique cependant des problèmes quand arrive le temps de chercher l'information pertinente dans l'ensemble des informations disponibles [70]. Il devient difficile de retracer ce qui est important parmi toute l'information superflue.

L'accès et le stockage de ces informations est possible par le recours à l'électronique numérique et à l'ordinateur, ce qui amène un second problème. L'information stockée

numériquement n'est pas directement déchiffrable par l'utilisateur. Elle doit être retransformée sous un format qui lui est accessible. L'utilisateur accède à l'information à travers ses sens. La vue est le sens le plus important chez l'utilisateur moyen. C'est d'ailleurs le sens principalement exploité dans la transmission d'informations de l'ordinateur vers l'utilisateur. La vue offre la meilleure bande passante et capte le plus d'informations. Elle est capable de percevoir des distances, des tailles, des couleurs, des formes, du mouvement, des structures, des regroupements... [20, 74]

La visualisation de l'information est une branche du domaine des interactions personne-machine qui s'occupe de développer des techniques pour communiquer de manière visuelle à l'aide de l'ordinateur. L'utilisation de cette approche nous permettra de maximiser notre utilisation du potentiel visuel. En combinant des éléments de plusieurs autres disciplines scientifiques (visualisation scientifique, interfaces utilisateur, extraction de données, graphisme, ergonomie...), elle permet à l'utilisateur d'exploiter ses capacités visuelles naturelles pour interagir rapidement avec l'information présentée. Il peut ensuite manipuler l'information, l'assimiler, la filtrer de ce qui n'est pas pertinent, la classifier et même y faire des découvertes [13, 31].

Dans notre cas, la quantité d'information est grande. Afficher de très gros graphes tend à causer une saturation du support visuel (l'écran). La densité d'éléments à afficher surpassant la résolution, le résultat global est inutilisable [36]. En plus de cette considération pratique, il faut se rappeler que l'être humain est sensible à l'esthétisme de l'information sur laquelle il travaille. L'information doit être claire, formatée et identifiable [27]. Ses constituants doivent être placés de façon à ce que l'utilisateur puisse en déduire la structure.

Un autre point à considérer au niveau de l'utilisateur, ce sont les limitations qu'impose sa capacité à se repérer lors de mouvements de l'information ou des vues sur cette information. En effet, l'utilisateur emploie une construction subconsciente, la carte mentale (« mental map » en anglais) [54], qui doit être entièrement reconstruite si les changements sont trop brusques. La carte mentale utilise les relations de positionnement relatif entre éléments, de proximité (rapprochement physique des éléments) et topologiques (divisions visuelles). Consi-

dérons par exemple les points de la figure 1.9. Le point c est au-dessus du d et à droite des points a et b (positionnement relatif). Ce même point c est proche du d et du e (proximité). Les points sont regroupés en deux régions (topologie), une à gauche du pointillé (sommets a et b), l'autre à droite (sommets c, d, e). Une transition trop brusque dans l'information affichée, c'est-à-dire un changement visuel qui ne tient pas compte de ces relations ou qui ne donne pas à l'utilisateur le temps de s'habituer, va détruire la carte mentale. L'utilisateur perd alors ses repères et doit se réaccoutumer, ce qui le ralentit. L'utilisateur a besoin de continuité dans la transformation de l'information visuelle qu'il étudie.

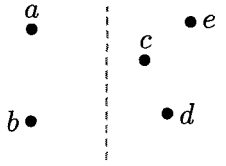


FIG. 1.9 – Quelques points.

L'information ne peut pas être placée arbitrairement sur l'écran. Les limitations du médium physique d'affichage sont à considérer dans la sélection de la représentation à donner à l'information, et de la manière de la mettre à l'écran. Pour afficher nos graphes, nous combinerons des méthodes basées sur le placement des sommets du graphe à partir d'une simulation physique de type "masses-ressorts". Cette approche est essentiellement construite autour de la physique des particules. Afin d'afficher un plus grand nombre de sommet, nous allons recourir à une représentation 3D.

1.2.2 L'affichage 3D des graphes.

Travailler sur de gros graphes nous confronte au problème de densité de l'information dans le support visuel. À notre avis, une représentation 3D du graphe semble désignée pour remédier à ce problème. C'est aussi la conclusion de *Ware et Franck* [77]. Voici un résumé de leur analyse.

La première impression est que la visualisation d'un graphe en 3D permet d'afficher nettement plus d'information à l'utilisateur. Soit I_{1d}, la quantité d'information assimilable (la

quantité d'information qu'une personne peut aisément distinguer et interpréter) dans un espace unidimensionnel (une ligne). Si un utilisateur travaille avec aise sur une ligne de I_{1d} éléments, il est logique de croire qu'il peut travailler sur I_{1d} de ces lignes dans un affichage 2D, donc que le cerveau est capable d'assimiler $(I_{1d})^2$ éléments. En poursuivant la même idée, l'ajout d'une troisième dimension donne un cube de $(I_{1d})^3$ éléments assimilables. En réalité, ce n'est pas le cas. Notre vision n'est pas vraiment tridimensionnelle dans le sens où nous ne percevons pas l'intérieur des solides, mais uniquement leur spatialité et leurs surfaces. De plus, certains solides peuvent en cacher d'autres ou avoir des faces obstruées.

Mécaniquement, la spatialité est obtenue par la vision stéréoscopique. Considérons que I_1 quantifie les éléments assimilables par un seul oeil (l'image est alors 2D) et que $I_2 = C(I_1)$, où C est une constante, quantifie les éléments assimilable par deux yeux. On peut penser que la valeur de C est alors au plus 2, puisqu'il y a deux yeux, et au minimum 1, puisqu'on a au moins un oeil. La stéréoscopie utilisant deux images légèrement différentes, la valeur de C devrait tendre vers 1. C'est d'ailleurs ce que prédisaient *Ware et Franck* avant leur expérimentation.

Lorsqu'ils ont testé cette hypothèse, ils ont eu la surprise de constater qu'une vision stéréoscopique, couplée à un bon mécanisme de contrôle de la rotation de l'image, permet d'obtenir une valeur de C d'environ 3. Donc, l'utilisateur peut travailler aisément avec trois fois plus d'éléments visuels dans une image stéréoscopique que dans une image bidimensionnelle. En ce qui nous concerne, nous n'avons pas accès à l'équipement pour offrir une vraie vision stéréoscopique. La profondeur est exprimée par des indices visuels comme des occlusions, des couleurs et des tailles d'éléments variables. De plus, nous la combinons avec une rotation contrôlée. Ce cas, qui a aussi été étudié par *Ware et Franck*, donne une nette amélioration de la compréhension du graphe par l'utilisateur.

1.2.3 L'algorithme de placement.

Maintenant que nous avons défini l'espace dans lequel s'effectuera l'affichage des noeuds, il nous faut un *algorithme de placement*, un algorithme qui va se charger de répartir

les noeuds dans l'espace. Celui que nous avons choisi est basé sur une simulation physique. Ce choix se justifie par le placement obtenu, qui met l'emphase sur la symétrie du graphe et met en relief les regroupements de sommets [15]. La simulation physique a été introduite par *Eades* [24]. Plusieurs modèles et algorithmes sont présentés dans [7] et [18]. La simulation dont il sera question est fondée sur la physique des particules.

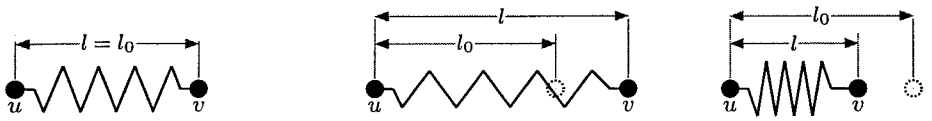
La simulation que nous effectuerons passe par la définition de systèmes "masses-ressorts" à partir de graphes. Étant donné un graphe, les noeuds sont associés à des *particules* sur lesquelles s'appliquent différentes forces. Une particule est en équilibre quand la somme des forces qu'elle subit est nulle. Pour simplifier, nos particules seront de masses unitaires. Les arêtes sont associées à des *ressorts*, que nous voulons de longueurs uniformes pour des raisons esthétiques. Cependant, il est aussi possible de définir la longueur des ressorts en fonction de certains paramètres. Nous allons désigner communément par la même notation le graphe et le système masses-ressorts qui lui est associé.

Une arête $e = \{u, v\}$ du graphe, représentée par le vecteur $\vec{uv}$, donnera lieu à deux force de cohésion, la première exercée sur la particule u par la particule v (f_{uv}) et la seconde exercée par la particule v sur la particule u (f_{vu}). La force f_{uv} est définie comme suit :

$$f_{uv} = -k(l_0 - l)\frac{\vec{uv}}{|\vec{uv}|} \tag{1.17}$$

où $l = |\vec{uv}|$, la longueur l_0 est la longueur au repos prédéfinie et le paramètre k détermine la « raideur » de la force de cohésion. Cette force équivaut à simuler un ressort régi par la loi de Hooke [8] entre u et v. Observons le comportement d'un tel ressort. Si $l = l_0$, il n'y a pas de force exercée ($f_{uv} = 0$) : le ressort est au repos (illustration 1.10(a)). Une valeur de l supérieure à l_0 signifie que le ressort est en extension (figure 1.10(b)). La force f_{uv} attire alors u vers v. Dans la situation contraire (l inférieure à l_0), il y a contraction du ressort (figure 1.10(c)) et la force f_{uv} cause la répulsion de u par v.

Le but de la simulation est d'obtenir la position d'équilibre du système, ce qui correspond à l'état où l'énergie cinétique des particules est très près de zéro. Au début de



(a) Au repos. (b) En extension. (c) En contraction.

FIG. 1.10 – Ressort entre deux particules.

la simulation, les particules sont distribuées arbitrairement dans l'espace euclidien 3D. C'est l'algorithme de simulation qui ramène itérativement le système en position d'équilibre en effectuant le calcul des forces exercées sur chaque particule et les nouvelles positions qui en découlent. Dans notre cas, la force de cohésion ne suffit pas à l'obtention de configurations d'équilibre visuellement intéressantes. Nous ajoutons donc une force de répulsion.

Pour notre modèle, la force de répulsion s'applique entre particules non-voisines partageant au moins un sommet commun. Si u est une particule, elle subit les forces de répulsion de l'ensemble de particules suivant :

$$R_u = N_2(u) - N(u) \tag{1.18}$$

Autrement dit, une particule subit une force de répulsion de toute particule qui est dans son 2-voisinage sans être dans son 1-voisinage. C'est une variation de la loi de Coulomb [39] qui est employée pour calculer la force de répulsion. Considérons trois particules t, u et v telles que t et v sont voisines de u, mais qu'il n'existe pas d'arête $\{v,t\}$ (figure 1.11). La formule pour calculer la répulsion entre v et t est la suivante :

$$F_{vt} = \beta \frac{\vec{vt}}{|\vec{vt}|^{\alpha}}; \beta > 0; \alpha > 1 \tag{1.19}$$

où $\vec{vt}$ est le vecteur $v - t$, où le paramètre β détermine l'amplitude de la force de répulsion et où α joue sur la distance à laquelle elle s'applique. Plus la distance α est élevée, plus les particules doivent être rapprochées pour qu'une force de même amplitude β fasse effet.

La force totale qui s'applique sur une particule u est la somme vectorielle de toutes

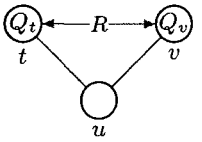


FIG. 1.11 – Répulsion entre deux particules.

les forces des ressorts et de toutes les forces de répulsion. C'est ce que représente la formule :

$$F(u) = \sum_{v \in N(u)} -k(l_0 - |\vec{uv}|)\frac{\vec{uv}}{|\vec{uv}|} + \sum_{v \in N_2(u) - N(u)} \frac{-\beta}{|\vec{uv}|^\alpha}\frac{\vec{uv}}{|\vec{uv}|} \tag{1.20}$$

Il est à noter qu'en changeant les différents paramètres (l, k, α et β), il devient possible d'influencer la forme de la représentation obtenue.

La simulation est effectuée en employant la deuxième loi de Newton, d'où l'on tire un système d'équations différentielles. Nous obtenons pour chaque particule une équation de la forme :

$$\ddot{u}(t) = \frac{1}{m}F(u(t);t) \tag{1.21}$$

où $\ddot{u}$ est la dérivée seconde et t le temps. Le système d'équations différentielles est résolu par une méthode numérique (la méthode d'Euler [7, 63] dans notre cas). Comme évoqué plus tôt, le processus de simulation se répète jusqu'à l'obtention d'un état d'équilibre. L'affichage continu de la simulation permet de préserver la carte mentale.

1.3 Exemples de graphes liés à des domaines d'application.

Voici quelques exemples de graphes modélisant des contextes afin d'illustrer l'utilité des graphes pour présenter de l'information structurée.

L'évolution des espèces est couramment représentée sous la forme d'un arbre. Chaque feuille est identifiée d'une espèce. Les noeuds internes de l'arbre sont des branchements de l'évolution, et comptent tous au moins deux enfants. Les arêtes représentent l'évolution elle-même [42, 44]. La figure 1.12 montre l'arbre de la vie terrestre. Des arbres similaires

sont construits pour étudier l'évolution génétique.

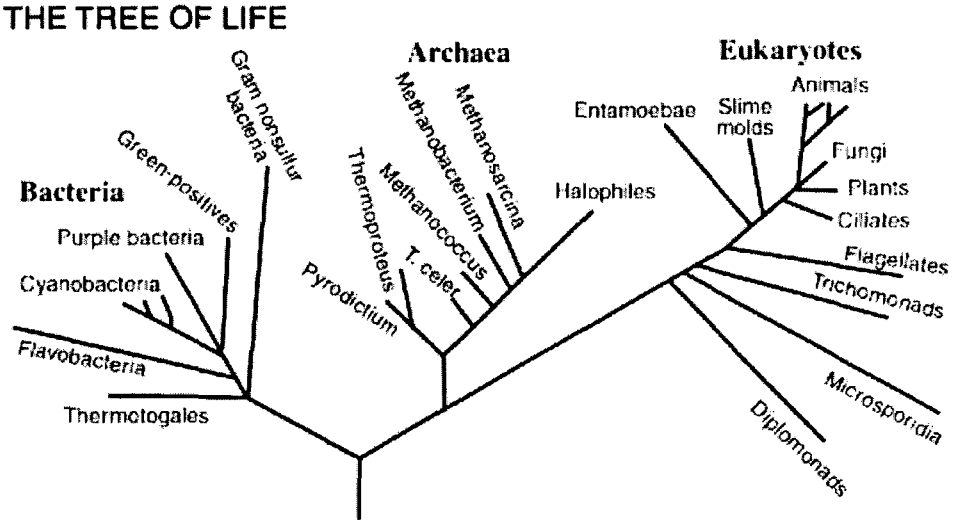
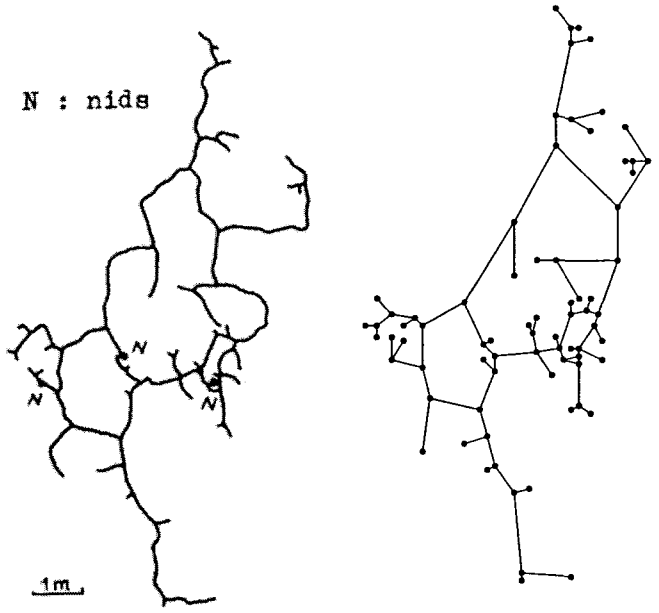


FIG. 1.12 – L'arbre de la vie terrestre (Image tirée de [79]).

Les graphes sont aussi employés en zoologie dans l'étude des différentes espèces. Prenons l'exemple du campagnol terrestre, un petit mammifère d'Europe et d'Asie. Dans sa variété fouisseuse, il construit des terriers complexes (figure 1.13(a)) en cherchant les racines, sa principale source de nourriture. *Airoldi* [2] a tenté de mieux comprendre le petit rongeur en considérant ses terriers comme des graphes (figure 1.13(b)). Il a transformé culs-de-sac et intersections en noeuds et tunnels en arêtes pour former un graphe.

En chimie, les molécules sont couramment illustrées à l'aide de graphes dont les noeuds remplacent les atomes et les arêtes, leurs liaisons. L'alcool isoamylique est présenté à la figure 1.14. Il combine 5 atomes de carbone (C), 12 atomes d'hydrogène (H) et 1 atome d'oxygène (O).

Il existe de nombreux exemples d'application des graphes en informatique. Parmi les applications les plus connues, plusieurs proviennent de la méthodologie de conception UML. Considérons par exemple les diagrammes de classes (figure 1.15) qui illustrent la structure de logiciels objets. Dans ces graphes, les classes sont représentées par les noeuds et les différentes relations entre elles (association, agrégation, héritage...) sont représentées par des arêtes.



(a) Le terrier (l'image originale est issue de l'article [2]).

(b) Le graphe.

FIG. 1.13 – Représentation d'un terrier de campagnol en graphe.

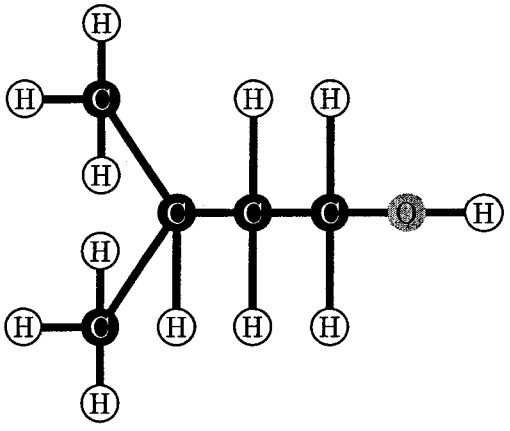


FIG. 1.14 – Molécule d'alcool isoamylique ($C_5H_{11}OH$).

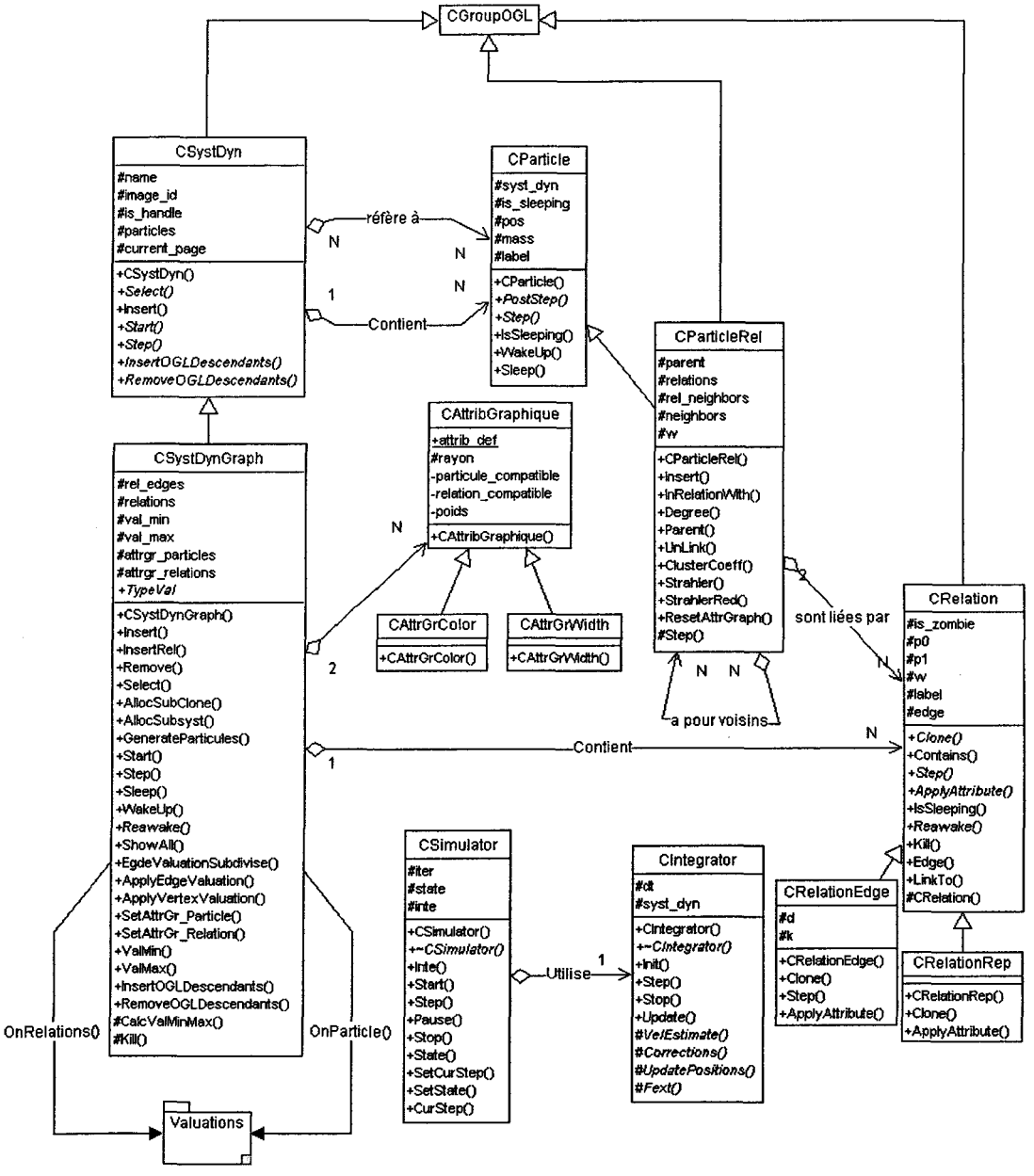


FIG. 1.15 – Exemple de diagramme de classes.

Les graphes servent aussi à la résolution de problèmes. On transcrit un problème de façon à le ramener à un aspect connu de la théorie des graphes. Trouver l'ordonnancement optimal de t tâches employant des ressources communes est un exemple illustrant cette manière de travailler. Chaque tâche t_i nécessite un sous-ensemble T_i des ressources communes (T) et une ressource ne peut être utilisée que par une tâche à la fois. Ce problème se représente par un graphe (figure 1.16). Une arête existe entre deux noeuds, un noeud i associé à la tâche t_i et un noeud j associé à t_j, s'il y a au moins une ressource commune à t_i et à t_j. En d'autres mots, l'arête $\{i, j\}$ existe si $T_i \cup T_j \neq \emptyset$. Pour obtenir la solution, on fait la coloration du graphe. On trouve le nombre minimal de couleurs pour qu'une couleur soit attribuée à chaque noeud sans que deux noeuds voisins n'aient la même [19]. Les colorations seront étudiées avec plus de détails en 3.1.1.

t	$T = \{A, B, C, D, E\}$
t_1	$T_1 = \{A, B\}$
t_2	$T_2 = \{C, D, E\}$
t_3	$T_3 = \{A, D, E\}$
t_4	$T_4 = \{B, C\}$
t_5	$T_5 = \{D\}$

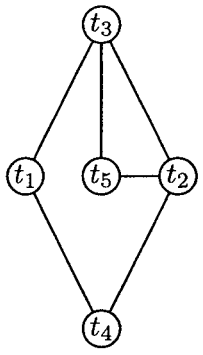


FIG. 1.16 – Des tâches et leur graphe de conflits de ressources.

CHAPITRE 2

GRAPHES ET MAINTENANCE DU LOGICIEL.

Les graphes permettent l'affichage d'informations dans une multitude de domaine. Nous avons vu parmi les nombreux exemples (voir 1.3) un usage des graphes dans le cadre de l'ingénierie à la conception de nouveaux logiciels. Les graphes peuvent aussi être employés face aux défis particuliers liés à une autre branche du génie logiciel, la maintenance de logiciels. C'est le sujet de ce chapitre. Nous débuterons par une introduction au génie logiciel et à la maintenance. Nous exposerons différents problèmes rencontrés en maintenance, et nous montrerons l'utilité des graphes dans la résolutions de ces problèmes.

2.1 L'ingénierie du logiciel.

Les premiers ordinateurs étaient programmés directement en « langage machine ». La grande complexité de ce langage rendait les manipulations de l'ordinateur frustrantes et s'avérait être une source d'erreurs. En plus, elle engendrait de l'incertitude sur la validité des résultats obtenus.

> [...]il devint clair que ce n'était pas la responsabilité principale du programmeur de contrôler la machine en définissant étape par étape un processus de calcul. Sa tâche était de formuler un algorithme qui produisait correctement et sans ambiguïtés la procédure mécanique permettant l'obtention de la solution à un problème donné [48] [1].

C'est ici qu'intervient l'*abstraction.* L'abstraction, c'est *le fait de considérer à part un élément d'une représentation ou d'une notion, en portant spécialement l'attention sur lui et*

[1] traduit de l'anglais

en négligeant les autres [66]. Ici, il s'agit de camoufler les éléments informatiques derrière des concepts plus généraux et plus proches de la pensée de l'utilisateur. L'abstraction a répondu au problème de la difficulté de la programmation machine en donnant naissance aux premiers langages d'assemblage et au premier langage de haut niveau, le Fortran (1954-57) [68]. Plus tard, l'avènement de Smalltalk a entraîné l'apparition de l'approche objet. Les outils que sont les langages de haut niveau et les langages orientés objet ont réduit l'effort nécessaire à la programmation de l'ordinateur. Ils ouvrent la porte à l'expression d'algorithmes avec un minimum de considérations pour les plateformes d'implémentation. Les types de données et les structures fondamentales les plus utiles (comme les boucles) deviennent des concepts abstraits qui sont transformés par la machine elle-même, à travers le compilateur, en données utilisables.

Influencées par l'évolution des langages, les méthodologies de conception ont progressé. De nouvelles manières de concevoir sont apparues, dont UML [61, 67]. Elles ajoutent leurs propres abstractions au-dessus de celles internes aux langages de haut niveau (classe, fichiers, fonctions...) et se rapprochent encore plus de la pensée humaine. Les nouvelles abstractions, qui servent à faciliter la conception et la communication entre programmeurs et clients, sont conservées dans des documents autres que le code source du programme.

La discipline qui étudie les outils, les méthodologies de travail et les principes régissant la conception, la création, l'évolution et l'entretien de systèmes logiciels se nomme l'*ingénierie du logiciel* [76]. Sa définition officielle selon les normes de la IEEE (« Institute of Electrical and Electronics Engineers ») est la suivante [37] :

> (1) L'application d'une approche systémique, disciplinée et quantifiable au développement, à l'opération et à la maintenance d'un logiciel ; C'est-à-dire, l'application de l'ingénierie à un logiciel. (2) L'étude des approches y correspondant [2].

Le processus d'ingénierie est constitué de trois phases importantes [64]. La première est la phase de définition, c'est-à-dire le choix de la méthode de travail, la détermination des besoins et des ressources, l'analyse du problème, la conception de l'architecture et la planification des tâches. La deuxième étape est la réalisation du produit, où l'on applique en

[2] traduit de l'anglais

pratique les décisions de la première phase, tout en testant ce qui en résulte. La troisième phase, celle qui nous intéresse le plus, est nommée *maintenance*. C'est durant cette phase que l'on se charge d'assurer et d'augmenter l'utilité du produit.

Pendant longtemps, l'ingénierie appliquée au logiciel s'est surtout concentrée sur les deux premières phases pour les rendre plus rapides, plus simples, plus efficaces [45]. Il s'est formé une association d'idées entre celles-ci et le terme ingénierie du logiciel [14]. Comme le processus d'ingénierie englobe aussi une troisième phase, nous allons éviter la confusion en désignant plus spécifiquement l'ensemble des opérations de définition et de fabrication du logiciel par le terme *ingénierie constructive* (« forward engineering » en anglais) [59].

2.1.1 Maintenance de logiciels.

Les logiciels ne requièrent pas la maintenance pour les mêmes raisons que des objets manufacturés comme un système mécanique ou un circuit électrique. En moyenne, les objets manufacturés connaissent leurs plus hauts taux de défectuosités au début de l'utilisation (défauts de constructions) et à la fin (défauts dus à l'usure). C'est ce qu'illustre la courbe de la figure 2.1. L'usure associée au temps, à l'usage ou aux conditions environnementales n'affecte pas les logiciels [48]. En théorie, ils ne sont sujets qu'aux défauts liés à la conception et à l'implémentation, après quoi le taux de défectuosités se stabilise [64] (figure 2.2).

Les logiciels étant immunisés aux effets des conditions ambiantes, tout changement, toute maintenance origine de facteurs humains. Une ou plusieurs personnes éprouvent de nouveaux besoins, qu'il faut combler, face au logiciel. Ces nouveaux besoins proviennent d'erreurs (mauvaise conception, erreurs d'écriture dans le code, erreurs de logique...), d'omissions (fonctionnalités manquantes, opérations incomplètes...) ou d'incompréhensions entre fabricants et opérateurs.

La stratégie adoptée dans le domaine du logiciel, avec son marché agressif et son évolution technologique très vive, privilégie le remplacement des logiciels plutôt que leur mise à jour. Beaucoup ne connaîtront jamais la phase de maintenance [45], car c'est un autre produit qui viendra répondre aux nouvelles conditions d'utilisation et corriger les erreurs. Remplacer

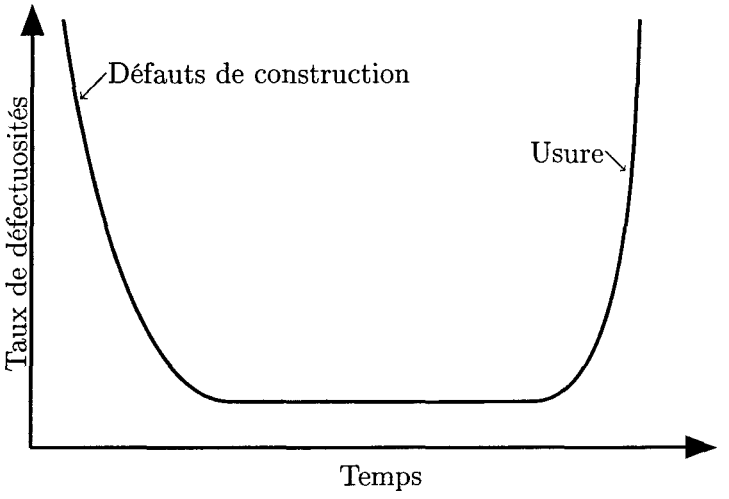


FIG. 2.1 – Courbe de défectuosités pour un objet manufacturé [64].

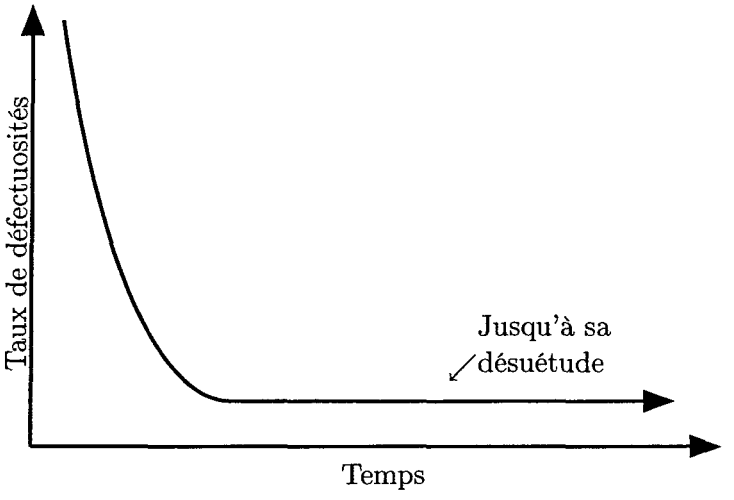


FIG. 2.2 – Courbe idéale de défectuosités pour un logiciel [64].

le système logiciel n'est pas toujours souhaitable. En effet, il est possible qu'il satisfasse les besoins à un haut niveau, qu'il contienne une somme substantielle de savoir corporatif [12] ou que les changements à lui apporter ne justifient pas les coûts d'une réécriture complète (en particulier s'il a une certaine ampleur). Lorsque la nécessité de faire des changements élimine l'option du statu quo et que le remplacement n'est pas acceptable, il faut modifier, d'où la maintenance dans le logiciel.

Les opérations de maintenance se classifient selon la nature des modifications voulues [49, 64]. Si la maintenance vient régler des non-conformités ou des problèmes de fonctionnement, elle est dite corrective. Si elle répond à un changement dans l'environnement d'utilisation, c'est la maintenance adaptative. Si le changement à effectuer touche le fonctionnement même du logiciel, on parle de maintenance pour l'amélioration. La maintenance préventive existe pour augmenter les performances ou la maintenabilité. Prenons pour exemple un logiciel de traitement de texte. Réparer un outil qui cause une sortie subite et imprévue du logiciel est de la maintenance corrective. Ajouter la langue japonaise afin de tenir compte d'un nouveau marché est de la maintenance adaptative. Remplacer le correcteur orthographique sur demande par un équivalent qui surligne les erreurs à même le texte est de la maintenance pour l'amélioration. Finalement, déplacer la gestion de fichiers dans une librairie dynamique séparée afin de faciliter les changements futurs est de la maintenance préventive.

Contrairement à la maintenance d'objets manufacturés, la maintenance d'un système logiciel n'a pas pour but premier de le ramener à un état proche de l'état d'origine, mais bien d'apporter des modifications. Il en résulte que chaque opération de maintenance peut avoir des conséquences inconnues sur la stabilité du système, ce qui invalide la courbe théorique du taux de défectuosités exposée précédemment, et donne une courbe réelle qui se rapproche de celle de la figure 2.3. Une modification apportée au logiciel cause un nouvel afflux d'erreurs, suivi d'une stabilisation, mais à un taux moyen de défectuosités plus élevé.

Ce phénomène d'augmentation du taux moyen d'erreurs est une conséquence directe de la maintenance sur la qualité du logiciel. En plus de venir avec leurs propres erreurs, les changements que l'on apporte au logiciel ont des effets secondaires sur celui-ci. L'architec-

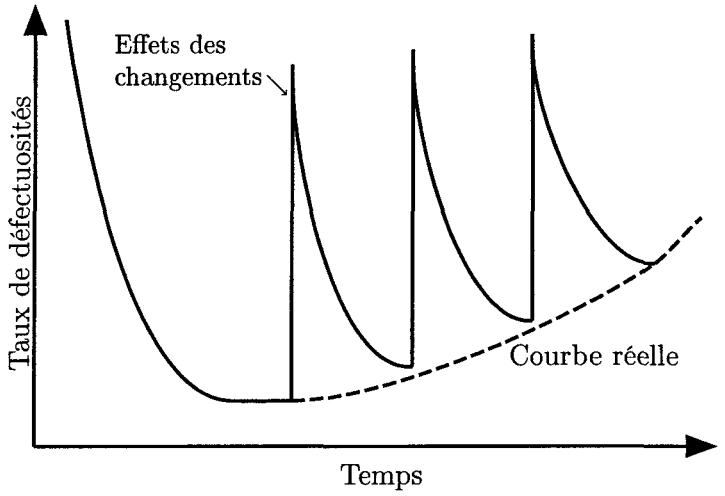


FIG. 2.3 – Courbe réelle de défectuosités du logiciel [64].

ture n'a pas nécessairement été prévue pour ces modifications, celles-ci peuvent s'appliquer à du code altéré lors de maintenances précédentes ou du nouveau code est ajouté au système en plus de ce qui était déjà présent. Fondamentalement, cela rend le système moins stable, le complexifie et par la même occasion complexifie les opérations de maintenance subséquentes [28, 43, 48].

La connaissance qu'ont les programmeurs d'un logiciel subit une dégradation similaire. À la longue, l'information connue sur un système logiciel s'écarte de celle nécessaire pour le maintenir [57]. Cet écart est produit par une multitude de causes. Les créateurs originaux risquent de s'atteler à d'autres tâches, d'oublier ou de ne plus être disponibles [6, 56]. Des déficiences dans la gestion de la documentation ou de mauvaises méthodes de travail ont laissé leurs marques, en particulier après plusieurs années de maintenance [45]. À la longue, la documentation peut manquer [38], s'accumuler au point d'être inassimilable ou ne plus être considérée comme sûre [12].

La dégradation qui se produit sur le fonctionnement, l'architecture et la documentation du logiciel font du code la seule source d'information fiable, et ce, même s'il ne contient pas tous les renseignements utiles (en particulier ceux relatifs à la conception) [57]. Cette fiabilité reste relative. Les noms de fonctions ou d'objets peuvent être peu informatifs, trompeurs

mêmes. Avec la dégradation de l'architecture, les structures abstraites et l'encapsulation des données sont compromises. Il n'y a pas de garanties que les algorithmes aient été ou soient restés lisibles [28].

Les actions de maintenance se classifient selon qu'elles s'attaquent à la tâche de corriger (modifier) ou aux problèmes de documentation ou de lisibilité.

2.1.2 Opérations d'ingénierie appliquées à la maintenance.

Les opérations d'ingénierie appliquées à la maintenance du logiciel se divisent en trois catégories : l'ingénierie reconstructrice, l'ingénierie réorganisatrice et l'ingénierie inverse [14].

Ingénierie reconstructrice L'ingénierie reconstructrice (« reengineering » en anglais) appliquée au logiciel, ou reconstruction du logiciel, est un mélange d'ingénierie constructive et d'ingénierie inverse. Il s'agit de réécrire une partie ou l'ensemble d'un système logiciel, souvent pour y implanter de nouvelles fonctionnalités. Si le système est rebâti dans son ensemble, la version précédente sert de documentation. Appliquer l'ingénierie reconstructrice à un logiciel de CAO (Conception Assistée par Ordinateur) consiste, par exemple, à le réécrire pour exploiter une nouvelle plateforme (comme passer de Windows à Linux) ou une nouvelle technologie (utiliser OpenGL en place de DirectX).

Ingénierie réorganisatrice L'ingénierie réorganisatrice (« restructuring » en anglais) du logiciel, ou restructuration du logiciel, consiste à transformer le logiciel d'une architecture d'un niveau d'abstraction donné à une autre architecture du même niveau d'abstraction, sans en modifier le comportement externe. Elle sert généralement à augmenter la maintenabilité ou à préparer le système logiciel pour des changements futurs. Un exemple d'ingénierie réorganisatrice d'un logiciel est de refaire la hiérarchie interne des objets (transformer d'une approche intégrée vers une approche client-serveur).

L'ingénierie inverse Le cadre des recherches exposées ici est l'ingénierie inverse (« reverse engineering » en anglais). Contrairement aux deux autres types d'opérations, elle ne modifie en rien le système existant. Elle sert à extraire de l'information et à construire

de la documentation. Une partie importante de notre travail va exposer et introduire des outils visuels d'analyse basés sur la visualisation de graphes. Ils serviront à extraire la structure de logiciels à partir de leurs codes sources. Combler les trous de documentation laissés par une perte accidentelle de fichiers est un exemple typique d'opération d'ingénierie inverse.

Si l'on considère la complexité des systèmes logiciels et les problèmes de dégradation des connaissances sur ceux-ci, il n'est pas surprenant que jusqu'à 50 à 90% du temps de maintenance (selon certaines études [12, 59]) soit consacré à l'ingénierie inverse plutôt qu'aux deux autres opérations.

2.2 Problématique.

Comme il a été évoqué dans notre aperçu de l'ingénierie du logiciel, les opérations de maintenance nécessitent que beaucoup de temps soit consacré à l'étude et à la compréhension du système logiciel existant. Optimiser le processus de compréhension du système logiciel a donc des effets directs sur les performances des programmeurs.

Même si un système logiciel ne s'est pas dégradé avec le temps, la compréhension du code source peut souvent être complexe. Saisir ce que fait exactement un groupe d'instructions ou un ensemble de fonctions risque de s'avérer ardu pour un programmeur qui n'en est pas l'auteur. Quand la quantité de lignes de code est importante (et certains systèmes en ont des millions [23]), cela devient carrément impossible pour une personne, ou une équipe de taille raisonnable, de comprendre le tout par la somme de ses parties [12].

L'abstraction est une aide très puissante à la compréhension du logiciel, mais à certaines conditions. Le but de l'abstraction est de camoufler la difficulté inhérente au code source, afin que l'utilisateur ne rencontre cette complexité que lorsque c'est absolument nécessaire. Elle lui permet de comprendre grossièrement le système avant de rencontrer les particularités. D'un autre côté, construire l'abstraction implique un travail d'analyse. Si ce travail d'analyse force l'utilisateur à se plonger dans la compréhension complète du code source, il n'y a aucun gain. Il faut être capable soit de générer l'abstraction d'une manière entièrement

automatique, soit de simplifier le processus de façon à diminuer l'interaction humaine. La première approche dépassant nos capacités informatiques actuelles (l'ordinateur est incapable d'abstractions « intelligentes »), il nous reste la deuxième.

Imaginons que lors de l'analyse qui mène à la construction d'une abstraction, il soit possible de diminuer l'ampleur de la tâche en effectuant des regroupements. Au lieu de produire l'abstraction du code à partir de chaque classe, fichier ou fonction individuel, la personne (ou l'équipe) responsable de la compréhension travaille sur des regroupements logiques de ces éléments. Ainsi, la formulation d'une bonne abstraction reste liée au programmeur, mais la quantité d'informations à traiter par celui-ci diminue. C'est sur ce principe que sont construits les algorithmes d'ingénierie inverse de *Mancoridis et al.* [51] et de *Chiricota et al.* [16]. Ce qui nous intéresse, c'est de suivre leurs traces en développant *une technique semi-automatisée pour assister le programmeur dans le processus d'abstraction d'un code source existant sans tenir compte de la documentation (si elle existe).*

Cette technique doit offrir au programmeur une vue d'ensemble du code, afin qu'il puisse faire les bons choix et les bonnes manipulations pour construire une abstraction. Le code source pouvant être constitué de plusieurs millions de lignes de codes, l'afficher dans son ensemble est impossible. Nous serions alors confronter au problème de densité de l'information évoqué précédemment (en 1.2.1). Il est préférable de représenter les éléments constitutifs du code (fonctions, fichiers, objets...) plutôt que chaque instruction individuellement. Nous avons vu déjà que l'un des principaux outils employés pour la visualisation de l'information est le graphe (voir aussi [4, 36]). Il s'avère que cet outil comble nos besoins. En substituant les éléments constitutifs d'un système logiciel et leurs relations à des structures du graphe, il devient possible de représenter ce système d'une manière très visuelle.

2.2.1 Les graphes utilisés.

Plusieurs types de graphes peuvent être produit avec aisance à partir du code source d'un système logiciel. Nous nous concentrerons plus particulièrement sur ceux-ci : le graphe d'inclusions pour les fichiers, le graphe d'appels pour les fonctions et le graphe d'associations

pour les objets.

Les graphes d'appels sont des graphes construits en utilisant les fonctions comme noeuds et en mettant une arête entre fonction appelante et fonction appelée [32]. Les figures 2.4 et 2.5(a) présentent du code C++ et le graphe d'appels qui en résulte.

Les graphes d'inclusions associent un noeud à chaque élément d'un ensemble de fichiers C ou C++. Les arêtes sont produites selon la présence dans les fichiers de la commande de précompilateur **#include**. Supposons que la commande **#include** "a" est présente dans un fichier b et que les fichiers a et b sont respectivement associés aux sommets v_a et v_b. Il y a alors une arête $\{v_a, v_b\}$ dans le graphe d'inclusions [3]. Il est possible de construire des graphes d'inclusions pour tous les langages où l'on retrouve ce genre de dépendances. Les figures 2.4 et 2.5(b) présentent du code C++ et le graphe d'inclusions qui en résulte.

Notre troisième graphe, désigné par graphe d'associations, est bâti sur le même principe que les deux autres. Chaque objet (class ou struct en C++) est un noeud. Il y a une arête entre deux objets s'il existe une association UML (héritage, agrégation...) entre eux. Les figures 2.4 et 2.5(c) contiennent du code C++ et le graphe d'associations correspondant.

Fichier 1.C

```
#include "2.C"
#include "3.C"

Class cA
{
public:
  un_a(cD& _un_d) :
    un_d(_un_d) {}
  void a() { un_d->d(); }
private:
  cD un_d;
};

void main(void)
{
  cA un_a(un_d);
  cB un_b;
  un_a->a();
  un_b->b();
}
```

Fichier 2.C

```
class cB
{
public:
  void b() { c(); }
};

class cC : public cB
{
public:
  void c() { }
};
```

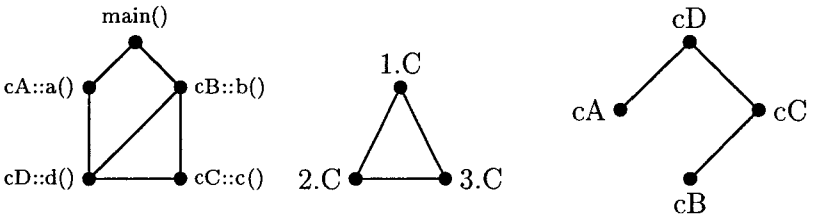
Fichier 3.C

```
#include "2.C"

class cD
{
public:
  cD(cC _un_c = cC()) :
    un_c(_un_c) {}

  void d(void)
  {
    if ( 1 )
      un_c->c();
    else
      un_c->b();
  }
private:
  cC un_c;
};
```

FIG. 2.4 – Exemple de code source C.



(a) Appels. (b) Inclusions. (c) Associations.

FIG. 2.5 – Graphes extraits du code source de la figure 2.4.

Ces trois types de graphes sont obtenus par un processus d'extraction similaire à la compilation. Le code source est déchiffré par un analyseur syntaxique. Alors que durant la compilation l'analyseur syntaxique produit des instructions pour l'ordinateur, ici il va identifier les éléments étudiés (fonctions, fichiers, objets...) et leurs relations. Un logiciel que nous utilisons pour effectuer cette tâche est Doxygen [75]. Celui-ci génère plusieurs graphes en format .dot (de Dotty [60]) pour l'un des trois types voulus, qui sont ensuite assemblés.

2.2.2 L'abstraction à partir des graphes.

C'est par l'opération de fragmentation expliquée en 1.1.3 que nous allons générer nos abstractions des graphes analysés et visualisés. Cette opération nous permettra d'obtenir des simplifications des graphes par le regroupement de noeuds en sous-ensembles. Bien que nous allons l'utiliser sur les graphes du génie logiciel, elle s'applique à d'autres familles de graphes.

La partition générée doit aider les opérations d'ingénierie inverse sur le système logiciel, donc répondre aux objectifs-clés de l'ingénierie inverse. Ces objectifs sont au nombre de six [14]. Les voici, en ordre décroissant d'importance par rapport aux buts de nos recherches.

Synthétiser des abstractions.

Le but ultime de l'ingénierie inverse est de faciliter la compréhension, et l'abstraction est un outil puissant dans cette direction. Dans notre cas, c'est le programmeur qui synthétise une abstraction du système logiciel à travers les partitions du graphe. Une bonne partition, par rapport à cet objectif, regroupe les éléments constitutifs étudiés (objets, fichiers,

fonctions) selon une logique évidente pour le programmeur qui analysera les sous-graphes obtenus. Prenons par exemple le cas d'un texteur (traitement de texte) multiplateforme. Une bonne partition consiste, par exemple, à le diviser de façon à obtenir un sous-graphe pour la gestion du texte, un deuxième sous-graphe pour la gestion des fichiers, un troisième pour le correcteur orthographique et enfin un dernier sous-graphe pour l'interface utilisateur. D'un autre côté, elle peut aussi le diviser autrement et obtenir un sous-graphe pour chaque plateforme supportée et un autre pour les parties communes. Une bonne partition est donc en adéquation avec un découpage logique du logiciel. Une mauvaise partition n'assiste en rien le programmeur. Elle risque même de lui faire perdre du temps si celui-ci s'attarde à l'analyser. Obtenir une bonne partition, afin que l'analyse du programmeur donne une bonne abstraction et que sa productivité s'en trouve augmentée, est l'objectif principal des recherches abordées ici.

Récupérer de l'information perdue.

Souvent, généralement même, un système logiciel a connu une époque durant laquelle il était bien documenté, et où il était facile d'en avoir une vue globale simplifiée. Accidents, manques de rigueur, temps, absences de politiques de gestions... divers événements entraînent des pertes d'informations. C'est un but de l'ingénierie inverse de reconstruire la documentation manquante ou de combler les besoins de compréhension du programmeur en créant une nouvelle documentation. Déjà, la création de nos trois types de graphes (associations, inclusions, appels) représente un pas dans la bonne direction pour remplir cet objectif. Cependant, une partition peut mettre en évidence des relations importantes entre les éléments visualisés du code. Avec la manière dont sont répartis les éléments du code (les noeuds du graphe) dans les différents sous-graphes, ce sont deux informations potentiellement utiles au programmeur.

Prendre en charge la complexité.

Les intervenants appelés à travailler sur le code d'un système logiciel sont fréquemment devant un vaste volume de données. Pour compenser à cette quantité d'informations, il

faut qu'au moins une partie de la tâche soit prise en charge par des mécanismes automatisés. Dans notre cas, la meilleure partition possible n'est pas un résultat intéressant si le programmeur doit regarder les noeuds du graphe un par un, surtout si l'on considère le temps nécessaire. Une bonne partition du graphe doit donc être générée rapidement, par un mécanisme automatisé ou semi-automatisé simple à contrôler.

Fournir des vues alternatives.

Graphiques ou textuelles, de nouvelles vues sur le système logiciel facilitent non seulement la compréhension, mais aussi les étapes de révision et de vérification du système. Il peut s'agir de générer une toute nouvelle façon d'organiser l'information comme de simplement permettre l'animation ou la rotation d'une vue existante. Dans notre cas, la nouvelle vue offerte à l'utilisateur, c'est notre partition du graphe et son graphe quotient. Toute partition du graphe remplit cet objectif. Une bonne partition, selon les normes établies par les objectifs précédents, a toutefois l'avantage d'une vue mieux structurée et plus intelligente, comblant mieux l'objectif de fournir des vues alternatives.

Faciliter la réutilisation et trouver des effets de bord.

Les deux derniers objectifs ne concernent pas notre recherche. Faciliter la réutilisation consiste à identifier les parties d'un système logiciel propices à être réutilisées ailleurs dans le même système ou dans un autre système. Trouver des effets de bord (effets secondaires), c'est mettre en relief des erreurs dans le design ou l'intégration de modifications qui peuvent avoir des conséquences néfastes cachées.

2.2.3 La partition recherchée.

Les objectifs-clés mettent en évidence de nombreuses caractéristiques auxquelles doit correspondre la partition produite par l'algorithme dont nous avons besoin. Certaines sont basées sur des caractéristiques topologiques du graphe qui s'évaluent de manière purement mécanique (comme la connectivité des sous-graphes), mais d'autres s'appuient sur des critères humains (regrouper les sommets selon la sémantique). Afin de permettre l'obtention

de la meilleure partition possible du point de vue de l'ingénierie inverse, nous préconisons que l'algorithme recherché soit paramétrable. Cela permettra à l'utilisateur de contrôler ce qu'il voit et d'explorer l'espace des partitions. Ainsi il pourra en sélectionner une qui lui semble meilleure selon les critères humains.

CHAPITRE 3

ALGORITHMES DE FRAGMENTATION.

Ce chapitre est consacré à l'étude de différentes approches pour fragmenter un graphe. Nous débuterons par des techniques dérivées de la théorie des graphes. Nous verrons la relation entre coloration et fragmentation dans un graphe. Nous étudierons les cliques et la dominance et nous construirons des partitions en combinant ces nouveaux concepts aux distances mesurées dans le graphe.

La deuxième partie du chapitre est vouée aux techniques de routage. Tout réseau peut être représenté avec un graphe où les sommets incarnent les appareils et les arêtes leurs liens. Afin d'obtenir une transmission des messages, les algorithmes de routage effectuent des regroupements de noeuds. Ces regroupements forment souvent des partitions du graphe. Nous étudierons deux algorithmes de fragmentation inspirés de stratégies de routage.

Deux outils de fragmentation de graphes du génie logiciel seront aussi étudiés. Le premier, Bunch, explore l'espace des partitions du graphe pour trouver une solution optimale selon un critère mathématique précis. Le second, la filtration d'arêtes, identifie les arêtes ayant le plus de chances d'être des isthmes. Ces arêtes sont éliminées et les composantes connexes du graphe partiel obtenu déterminent la partition.

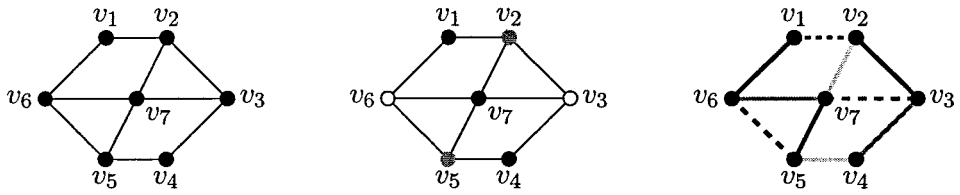
Le chapitre se terminera par une analyse de ces différents algorithmes relativement à notre problématique.

3.1 Méthodes de la théorie des graphes.

Voici diverses méthodes de fragmentation d'un graphe inspirées ou dérivées d'éléments étudiés en théorie des graphes.

3.1.1 La coloration de graphes.

Les problèmes de coloration sont parmi les plus étudiés dans la théorie des graphes. La coloration d'un graphe consiste à assigner une couleur à chacun des sommets de façon à ce que deux sommets adjacents soient de couleurs différentes [9, 10, 19]. Une k-coloration d'un graphe est une coloration qui utilise k couleurs. Si la valeur de k est le nombre minimal de couleurs nécessaires pour colorer les sommets d'un graphe G, la coloration est dite minimale et k est le nombre chromatique de G, ou $\chi(G)$. Si ce sont les arêtes du graphe qui sont coloriées avec q couleurs différentes, il s'agit d'une q-coloration d'arêtes du graphe. Dans une coloration d'arêtes, des arêtes incidentes à un sommet commun sont toutes de couleurs différentes. Le nombre minimal de couleurs nécessaires pour une coloration d'arêtes d'un graphe G est désigné par indice chromatique $q(G)$ de G et est égal au degré maximal parmi tous les sommets du graphe. Le graphe 3.1(b) illustre une coloration valide pour le graphe 3.1(a). Afin que deux sommets adjacents n'aient pas la même, trois couleurs différentes doivent être utilisées. Une coloration d'arêtes pour le même graphe est illustrée à la figure 3.1(c). Quatre couleurs y sont nécessaires. Le nombre chromatique du graphe 3.1(a) est 3 et son indice chromatique est 4.



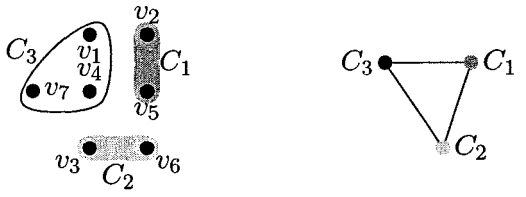
(a) Le graphe. (b) Sommets. (c) Arêtes.

FIG. 3.1 – Un graphe et des colorations de celui-ci.

La majorité des colorations ne favorisent pas une couleur par rapport aux autres. Deux couleurs distinctes vont colorer un nombre semblabe d'éléments. On dit de ces colorations qu'elles sont régulières [19]. Les colorations 3.1(b) et 3.1(c) sont justement des colorations régulières. Chacune des trois couleurs de la coloration 3.1(b) ou des quatres couleurs de la coloration 3.1(c) est associée à 2 ou 3 éléments du graphe.

Fragmentation par k-coloration.

Toute k-coloration d'un graphe $G = (V, E)$ est une partition $P = \{C_1, C_2, ..., C_k\}$ de G où chaque sous-graphe C_i contient l'ensemble des sommets d'une couleur de la coloration. Deux sommets connexes sont nécessairement de couleurs différentes et appartiennent à des sous-graphes différents de la partition. Il n'y a donc aucune connectivité entre les sommets d'un sous-graphe C_i, mais il y a une forte connectivité entre les sous-graphes distincts de la partition. La coloration 3.1(b) donne les trois sous-graphes suivants : $C_1 = \{V_2, V_5\}$, $C_2 = \{V_3, V_6\}$ et $C_3 = \{V_1, V_4, V_7\}$, illustrés en 3.2(a), et le graphe quotient 3.2(b). On constate l'absence d'arêtes dans le sous-graphe, ce qui marque l'absence de connectivité interne, et le graphe quotient complet, signe d'une très forte connectivité externe.



(a) Composants. (b) Graphe quotient.

FIG. 3.2 – Partition à partir de la coloration usuelle 3.1(b).

3.1.2 Cliques k-distantes.

En étudiant le dessin de graphes de réseaux, *Edachery et Al.* [25] ont élaboré l'idée similaire à la nôtre que regrouper les sommets permet de diminuer l'encombrement et la complexité visuelle. Les détails inutiles ou mineurs sont éliminés pour laisser place à la hiérarchie principale du réseau.

Les auteurs ont d'abord déterminé ce qu'était pour eux un bon regroupement, qu'ils désignent par supernoeud ou métanoeud (« supernode » ou « metanode » en anglais). Il s'agit d'un regroupement de sommets tel que ses sommets constitutifs ont de plus fortes relations entre eux qu'avec les autres sommets du graphe. L'algorithme étudié ici considère que la force de relation entre deux sommets est inversement proportionnelle à la distance qui

sépare les sommets. Plus deux sommets sont proches, plus la relation entre eux est forte.

Une clique est un regroupement de sommets tel que tous les sommets sont voisins les uns des autres. Le sous-graphe qui résulte d'une clique est isomorphe à un graphe complet. Une clique k-distante est un regroupement de sommets qui sont tous k-voisins les uns des autres. Autrement dit, la distance maximale entre deux sommets quelconques du regroupement est au plus k. Une clique est donc aussi une clique 1-distante. Le diamètre du sous-graphe qui résulte d'une clique k-distante est, par définition, d'au plus k.

L'algorithme de Edachery.

La définition que font *Edachery et al.* d'une bonne partition basée sur la distance correspond à subdiviser un graphe dans un nombre minimal de cliques k-distantes. Trouver le nombre minimal de cliques est un problème de complexité NP [25] (voir [72] pour les détails sur la complexité). Comme une clique est en fait une clique 1-distante, il faut en déduire que trouver le nombre minimal de cliques k-distantes est aussi un problème de complexité NP. Pour conserver une complexité temporelle plus raisonnable, on emploie un algorithme qui trouve un nombre relativement petit de cliques k-distantes, mais pas nécessairement le minimum.

L'algorithme applique récursivement des fusions dans les sous-graphes d'une partition d'un graphe $G = \{V, E\}$ jusqu'à ce qu'aucune fusion ne soit plus possible. Une fusion consiste à réunir les sommets de plusieurs sous-graphe de G pour former un nouveau sous-graphe de G. La fusion n'est possible que lorsque le sous-graphe qui en résulte est de diamètre maximal k, pour $k \geq 2$. Le calcul du diamètre étant une opération relativement coûteuse (elle est $O(n^2)$ [80]), on utilise plutôt l'approximation suivante : pour un sous-graphe $C_{i,j}$, le diamètre estimé $D'(C_{i,j})$ est le plus grand diamètre possible pour ce sous-graphe.

L'algorithme construit itérativement des partitions :

$$P_i = \{C_{i,1} = \{V_{i,1}, E_{i,1}\}, C_{i,2} = \{V_{i,2}, E_{i,2}\}, ..., C_{i,n_i} = \{V_{i,n_i}, E_{i,n_i}\}\}, \tag{3.1}$$

où les sous-graphes de P_i résultent de fusions dans les sous-graphes de P_{i-1}. L'algorithme termine à la partition P_n (où $0 \leq n \leq |V|$) telle qu'il n'y ait plus de fusions possibles.

Le point de départ de la récursion est une partition initiale P_0 de G construite de la manière suivante : Le sous-graphe $C_{0,1}$ est constitué de l'ensemble $N[v_0]$, où v_0 est un des sommets de plus haut degré du graphe. Après avoir choisi $C_{0,1}$, les graphes $C_{0,j}$, pour $j > 1$, sont générés. Considérons l'ensemble :

$$V'_{0,j} = V - \bigcup_{h=1}^{j-1} V_{0,h} \tag{3.2}$$

l'ensemble des sommets qui ne sont pas encore dans un sous-graphe de P_0. On sélectionne v_j, un sommet de $V'_{0,j}$ dont le degré est maximal dans G. L'ensemble $V_{0,j}$ contient tous les sommets de $N[v_j]$ qui sont dans $V'_{0,j}$. On continue jusqu'à ce que chaque sommet de G soit dans un sous-graphe de P_0. Le diamètre d'un sous-graphe $C_{0,j} \in P_0$ (pour $j \geq 1$) est nécessairement dans l'intervalle $[0, 2]$, la valeur de 2 étant obtenue dans le cas suivant : deux sommets a et b appartiennent à $N[v_j] \cap V'_{0,j}$, soit l'ensemble des sommets d'un sous-graphe $C_{0,j}$. Les sommets a et b ne sont pas voisins entre eux. La chaîne (a, v_j, b) de longueur 2 est, par la définition du 1-voisinage clos, l'une des plus courtes chaînes entre a et b. De plus, a et b représentent un cas de plus longue distance entre deux sommets de cet ensemble, ce qui donne un diamètre de 2. Le diamètre estimé est la borne supérieure de l'intervalle $[0, 2]$, donc $D'(C_{0,j}) = 2$.

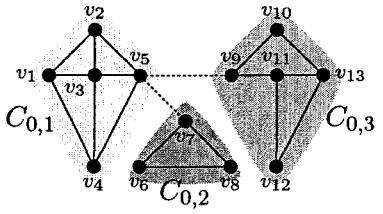


FIG. 3.3 – Noeuds-ponts.

Par la suite, pour $i \geq 0$, on construit la partition P_{i+1}. Tout d'abord, on trouve l'ensemble des noeuds-ponts W_i de P_i. Un noeud-pont dans W_i est un sommet incident à au moins une arête externe de P_i. Par exemple, les arêtes illustrées avec des pointillés dans le graphe de la figure 3.3 sont externes et les noeuds-ponts sont v_5, v_7 et v_9. Soit $\Gamma(v)$, le sous-

graphe de P_i auquel le sommet v appartient. On définit l'ensemble suivant pour chacun des noeuds-ponts :

$$L(v) = \{\Gamma(u) | u \in N[v] \text{ et } D'(\Gamma(u)) < k\} \tag{3.3}$$

Dans le cas des noeuds-ponts du graphe 3.3, on obtient, avec $k \geq 2$, les ensembles $L(v_5) = \{C_{0,1}, C_{0,2}, C_{0,3}\}$, $L(v_7) = \{C_{0,1}, C_{0,2}\}$ et $L(v_9) = \{C_{0,1}, C_{0,3}\}$. Soit $w_{i,1}$, un noeud-pont de P_i tel que $|L(w_{i,1})|$ est maximal. On effectue la fusion suivante : $C_{i+1,1}$ est le sous-graphe de G engendré par l'ensemble de sommets $V_{i+1,1}$ défini comme suit :

$$V_{i+1,1} = \bigcup_{C_{i,j} \in L(w_{i,1})} V_{i,j} \tag{3.4}$$

Le diamètre estimé $D'(C_{i+1,1})$ est la somme des deux plus grands diamètres estimés des sous-graphes de $L(w_{i,1})$, plus 1. Deux cas sont possibles :

1. Le diamètre estimé $D'(C_{i+1,1})$ est inférieur ou égal à k. La fusion fonctionne. La partition P_{i+1} contient le sous-graphe $C_{i+1,1}$ et les sous-graphes de P_i qui n'ont pas servi à engendrer $C_{i+1,1}$. On incrémente i pour une nouvelle itération de l'algorithme.
2. Le diamètre estimé $D'(C_{i+1,1})$ est supérieur à k. Le sous-graphe $C_{i+1,1}$ peut être trop gros pour constituer une k-clique. On enlève un sous-graphe quelconque de $L(w_{i,1})$ et on tente de nouveau la fusion. On recommence la procédure jusqu'à ce que $D'(C_{i+1,1})$ soit inférieur ou égal à k (cas précédent). Si $L(w_{i,1})$ ne contient plus que $\Gamma(w_{i,1})$, on le retire de W_i. On essaie alors avec $w_{i,2}$, un noeud-pont de W_i tel que la cardinalité $|L(w_{i,2})|$ est maximale, et ainsi de suite. Si $w_{i,j-1}$ (pour $j \geq 2$) ne fait pas, on retire $w_{i,j-1}$ de W_i et on recommence avec un noeud-pont $w_{i,j}$ de W_i tel que $|L(w_{i,j})|$ est maximale. Si la fusion échoue pour tous les noeuds-ponts de P_i, l'algorithme se termine. L'algorithme se termine aussi si $|L(w_{i,x})| \leq 1$ pour tous les noeuds ponts $w_{i,x} \in W_i$.

Prenons pour exemple le graphe $G = (V, E)$ de la figure 3.4(a). Nous voulons le fragmenter en cliques 5-distantes. La partition initiale P_0 est illustrée à la figure 3.4(b) avec ses noeuds-ponts (sommets pleins). On identifie un noeud-pont $w_{0,1}$ tel que $|L(w_{0,1})|$ est

maximal. C'est le noeud carré de la figure 3.4(b). L'ensemble $L(w_{0,1})$ contient $\{C_{0,3}, C_{0,5}, C_{0,6}\}$. Le diamètre estimé du sous-graphe $C_{1,1}$ issu de la fusion des éléments de $L(w_{0,1})$ est :

$$D'(C_{1,1}) = D'(C_{0,3}) + D'(C_{0,5}) + 1 = 2 + 2 + 1 = 5 \tag{3.5}$$

La fusion est donc possible. On obtient alors la partition P_1 illustrée à la figure 3.4(c). On identifie les noeuds-ponts de cette partition (ceux pleins dans l'illustration) ainsi qu'un noeud-pont $w_{1,1}$ tel que $|L(w_{1,1})|$ est maximal. Le sommet choisi est représenté par un noeud carré. La liste $L(w_{1,1})$ qui lui est associée contient $\{C_{1,2}, C_{1,3}\}$. Le diamètre estimé du sous-graphe $C_{2,1}$ issu de la fusion des éléments de $L(w_{1,1})$ est :

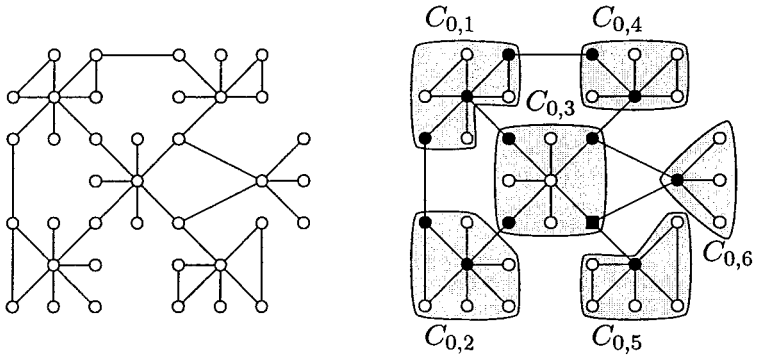
$$D'(C_{2,1}) = D'(C_{1,2}) + D'(C_{1,3}) + 1 = 2 + 2 + 1 = 5 \tag{3.6}$$

ce qui correspond à une fusion valide. On obtient la partition P_2 de la figure 3.4(c). On identifie les noeuds-ponts. Ils sont pleins dans l'illustration. Tous les ensembles L des différents noeuds-ponts de P_2 ont une cardinalité de 1, ce qui signifie qu'il n'y a plus de fusions possibles. L'algorithme a terminé et P_2 est notre résultat final.

Edachery et Al. ont testé d'autres variantes de cet algorithme en jouant sur la sélection des sous-graphes à fusionner et le calcul du diamètre, mais le fonctionnement et les résultats restent similaires.

3.1.3 Dominance.

Un autre problème de complexité NP très connu lié aux graphes est celui des ensembles dominants minimaux [46], que l'on retrouve surtout dans les problèmes de réseautage et de hiérarchisation. Un ensemble dominant d'un graphe $G = (V, E)$ est un sous-ensemble V' de V tel que tout sommet de $V - V'$ est voisin d'au moins un sommet de V'. Un ensemble V' est dit dominant minimal si, en plus d'être dominant, on a $|V'| \leq |W|$ pour tout autre ensemble dominant W. Pour réduire la complexité temporelle dans le calcul des ensembles dominants, on emploie parfois des heuristiques. Ce sont des algorithmes qui trouvent des



(a) G. (b) P_0.

(c) P_1. (d) P_2.

FIG. 3.4 – Fragmentation avec l'algorithme d'*Edachery et Al.*

ensembles dominants quasi-minimaux, c'est-à-dire des ensembles de cardinalité relativement petite sans être dominants minimaux. Comme les graphes du génie logiciel sont grands, des heuristiques devront être employées. Considérons le graphe illustré par la figure 3.5 : l'ensemble $D_1 = \{a, e, j\}$ est un exemple d'ensemble dominant dans ce graphe tandis que l'ensemble $D_2 = \{c, h\}$ y est un exemple d'ensemble dominant minimal.

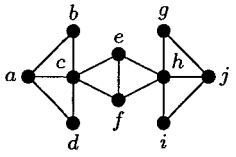


FIG. 3.5 – Un graphe.

Obtenir une partition à l'aide d'un ensemble dominant.

De nombreux algorithmes pour calculer des ensembles dominants minimaux [33, 34] ou quasi-minimaux [46, 73] sont cités dans la littérature. À partir des ensembles dominants qu'ils produisent, il est possible de fragmenter le graphe. L'algorithme 3.1 est un exemple d'algorithme de fragmentation qui prend un ensemble dominant en entrée. Les sous-graphes sont produits en regroupant les sommets selon leur proximité aux sommets de l'ensemble dominant. À l'aide d'un ensemble dominant $D = \{d_1, d_2, .., d_n\}$ sur l'ensemble des sommets V d'un graphe $G = (V, E)$, l'algorithme produit une partition P de taille n. On commence par produire les graphes $C_1 = \{V_1, E_1\}, C_2 = \{V_2, E_2\}, .., C_n = \{V_n, E_n\}$, tous vides. Ce sont nos futurs sous-graphes. Chaque sommet $d_j \in D$ est placé dans V_j. On observe le voisinage de chaque sommet v de $V - D$. Si $|N(v) \cap D|$ ne contient qu'un seul sommet d_j, alors v est placé dans V_j. Si $|N(v) \cap D|$ contient plus d'un sommet, v est placé dans l'ensemble U. L'ensemble U contient donc les sommets incidents à plusieurs sommets de l'ensemble dominant D. L'étape suivante consiste à répartir chaque sommet $u \in U$ à l'ensemble V_j qui correspond aux règles suivantes :

- Le sommet d_j est incident à u.

– Il n'existe pas de sommet d_i incident à u tel que $|V_i| < |V_j|$.

Répartir ainsi les sommets de U permet d'uniformiser la taille des sous-graphes obtenus. Une fois que tous les sommets de U sont répartis dans un ensemble V_i, on construit les ensembles d'arêtes de nos graphes en mettant dans l'ensemble E_i toute arête de E incidente à deux sommets de V_i. Après cela, chaque graphe C_i est un sous-graphe valide de G et l'ensemble des sous-graphes C_1 à C_n forme notre partition P de G.

Algorithme 3.1 FragmentationDominance($G = (V, E), D = \{d_1..d_n\}, P = \{C_1..C_n\}$)

```
V' ← V − D
U ← ∅
n ← |D|
pour i ← 1 à n faire
        C_i ← {V_i, E_i}
        V_i ← {d_i}
        E_i ← ∅
pour tout v ∈ V' faire
        j ← 0
        pour i ← 1 à n faire
                si v ∈ N(d_i) alors
                        si j ≠ 0 alors
                                j ← i
                        sinon
                                U ← U ∪ {v}
                                j ← n + 1
        si j < n + 1 alors
                V_j ← V_j ∪ {v}
pour tout u ∈ U faire
        j ← 0
        pour i ← 1 à n faire
                si u ∈ N(d_i) alors
                        si j = 0 ou |d_i| < |d_j| alors
                                j ← i
        V_j ← V_j ∪ {v}
pour tout e = {v_x, v_y} ∈ E faire
        pour i ← 1 à n faire
                si v_x ∈ V_i et v_y ∈ V_i alors
                        E_i ← E_i ∪ {e}
```

3.1.4 Dominance par distances.

La combinaison des concepts de k-voisinage (voir 1.1.1) et de dominance produit une nouvelle forme de dominance, la dominance par distances [35]. Un ensemble de sommets $D \subset V$ dans un graphe $G = (V, E)$ est k-dominant si pour chaque sommet $v \in (V - D)$,

la distance $d(v, D)$ est inférieure ou égale à k. On remarquera que si et seulement si D est k-dominant, alors $N_k[D] = V$.

Algorithme de fragmentation par dominance par distances.

Quelques ajustements sur l'algorithme 3.1 permettent d'obtenir un algorithme calculant une partition à partir d'un ensemble k-dominant (algorithme 3.2). Un sous-graphe est associé à chaque sommet dominant. Si un sommet v est k-voisin d'un seul sommet dominant d_i, v sera ajouté au sous-graphe de d_i. Si v est k-voisin de plusieurs sommets dominants, il sera ajouté au sous-graphe du sommet dominant le plus proche. Si plusieurs sommets dominants répondent à cette dernière condition, l'un de ceux dont le sous-graphe associé a le plus petit ordre reçoit v.

Cet algorithme nécessite en entrée un ensemble k-dominant. Plus l'ensemble est petit, moins il y a de sous-graphes dans la partition obtenue. Trouver un ensemble k-dominant minimal est un problème NP-complexe. Heureusement, *Henning* [35] a introduit un algorithme permettant de trouver un ensemble k-dominant de cardinalité maximale $n/(k+1)$, où n est l'ordre du graphe à fragmenter. Cet algorithme a été utilisé avec succès par *Auillans et Al.* [4] pour faire la fragmentation de graphes bipartis.

L'algorithme 3.3 tiré de *Henning* construit un ensemble k-dominant D_k sur un graphe connexe $G = (V, E)$.

Ce que fait l'algorithme, c'est calculer une suite d'arbres T_i (pour $i = (0, 1, ...)$). On débute par calculer un arbre T_0 qui recouvre le graphe G. Un arbre recouvrant (« spanning tree » en anglais) de G est un graphe partiel de G qui est aussi un arbre. On choisit une chaîne $U_0 = \{u_0, u_1, .., u_n\}$ dont la longueur est égale au diamètre de T_0. Ensuite, on retire le $k^{\text{ième}}$ sommet de U_0 que l'on place dans notre futur ensemble dominant D_k. Cette opération sépare l'arbre en deux sous-graphes. On pose T_1 comme étant le sous-graphe qui contient le sommet u_{k+1} de U_0. On réapplique itérativement la procédure : on calcule une chaîne U_i dans T_i de longueur égale au diamètre de T_i. On retire de l'arbre T_i le $k^{\text{ième}}$ sommet de U_i. Ce sommet rejoint notre ensemble D_k. Le sous-graphe avec le $(k+1)^{\text{ième}}$ sommet de U_i devient T_{i+1}. Ce

Algorithme 3.2 FragDomDistance($G = (V, E), k, D_k = \{d_1..d_n\}, P = \{C_1..C_n\}$)

$V' \leftarrow V - D$
$U \leftarrow \emptyset$
$n \leftarrow |D_k|$
pour $i \leftarrow 1$ à n **faire**
    $C_i \leftarrow \{d_i\}$
pour tout $v \in V'$ **faire**
    $j \leftarrow 0$
    $d \leftarrow \infty$
    pour $i \leftarrow 1$ à n **faire**
        si $v \in N_k(d_i)$ **alors**
            si $d > d(v, d_i)$ **alors**
                $j \leftarrow i$
                $d \leftarrow d(v, d_i)$
            sinon si $d = d(v, d_i)$ **alors**
                $j \leftarrow n + 1$
    si $j < n + 1$ **alors**
        $C_j \leftarrow C_j \cup \{v\}$
    sinon si $j = n + 1$ **alors**
        $U \leftarrow U \cup \{v\}$
pour tout $v \in U$ **faire**
    $j \leftarrow 0$
    $d \leftarrow \infty$
    pour $i \leftarrow 1$ à n **faire**
        si $v \in N_k(d_i)$ **alors**
            si $d > d(v, d_i)$ **alors**
                $j \leftarrow i$
                $d \leftarrow d(v, d_i)$
            sinon si $d = d(v, d_i)$ **alors**
                si $|C_i| \leq |C_j|$ **alors**
                    $j \leftarrow i$
    $C_j \leftarrow C_j \cup \{v\}$
Construire les ensembles E_i de chaque sous-graphe C_i

Algorithme 3.3 EnsembleKDominant($G = (V, E), k, D_k = \{d_1..d_n\}$)

$D_k \leftarrow \emptyset$
ConstruireArbreRecouvrant(G, T)
tant que Rayon de $T > k$ } **faire**
    $d \leftarrow$ diamètre de T
    $U \leftarrow$ une chaîne $\{u_1, u_2, .., u_d\}$ de T de longueur d.
    $D_k \leftarrow D_k \cup \{u_k\}$
    $T \leftarrow$ la composante avec u_{k+1} si on enlève l'arête $\{u_k, u_{k+1}\}$ de T
$v \leftarrow$ un centre de T
$D_k \leftarrow D_k \cup \{v\}$

processus se termine avec l'arbre T_j de rayon inférieur à k. Un centre de T_j est ajouté à D_k, qui est alors un ensemble k-dominant de G. Le tableau 3.1 donne une trace de l'algorithme appliqué sur le graphe 3.6(a) pour $k = 3$. Les différentes itérations sont illustrées par les figures 3.6(b) à 3.6(g). Les arêtes de la chaîne U_1 sont représentées en gras et le $k^{\text{ième}}$ sommet est plein.

Iter.	Arbre	Rayon	d	U_i	u_k	D_k
1	T_0	7	15	$\{h, c, d, e, a, b, f, g, l, p, o, n, s, r, m\}$	d	$\{d\}$
2	T_1	6	12	$\{i, j, k, f, g, l, p, o, n, s, r, m\}$	k	$\{d, k\}$
3	T_2	6	12	$\{e, a, b, f, g, l, p, o, n, s, r, m\}$	b	$\{d, k, b\}$
4	T_3	4	9	$\{w, x, u, p, o, n, s, r, m\}$	u	$\{d, k, b, u\}$
5	T_4	4	9	$\{m, r, s, n, o, p, l, g, f\}$	s	$\{d, k, b, u, s\}$
6	T_5	3	–	$\emptyset$	$\emptyset$	$\{d, k, b, u, s, p\}$

TAB. 3.1 – Fragmentation par 3-domination du graphe 3.6(a).

Il est possible d'obtenir une partition à partir de l'algorithme 3.3 sans utiliser l'algorithme 3.2 sur l'ensemble k-dominant généré. Il suffit de modifier l'algorithme 3.3 pour récupérer les composantes éliminées lors du retrait des arêtes de l'arbre recouvrant. En ajoutant à ces composantes l'arbre final de rayon inférieur ou égal à k, on a une partition du graphe. Pour notre exemple, nous aurions obtenu la partition P illustrée par la figure 3.6(h).

3.2 Stratégies de routage.

Dans un réseau d'ordinateurs comme dans un réseau de processeurs, la communication est une activité primordiale [5, 62]. Une bonne communication a un coût (distance parcourue) minimal pour tous les messages. Assurer une bonne communication par rapport à l'architecture et aux ressources disponibles est le rôle d'une stratégie de routage.

L'architecture d'un réseau est généralement exprimée sous la forme d'un graphe $G = (V, E)$, où les noeuds sont des éléments du réseau (processeurs, ordinateurs) et où les arêtes représentent des liens directs de communication. Dans certains cas, il est nécessaire de représenter les coûts de communication de chacun des liens (en temps, en bande-passante, etc...). On utilise alors un graphe valué (voir 1.1.2).

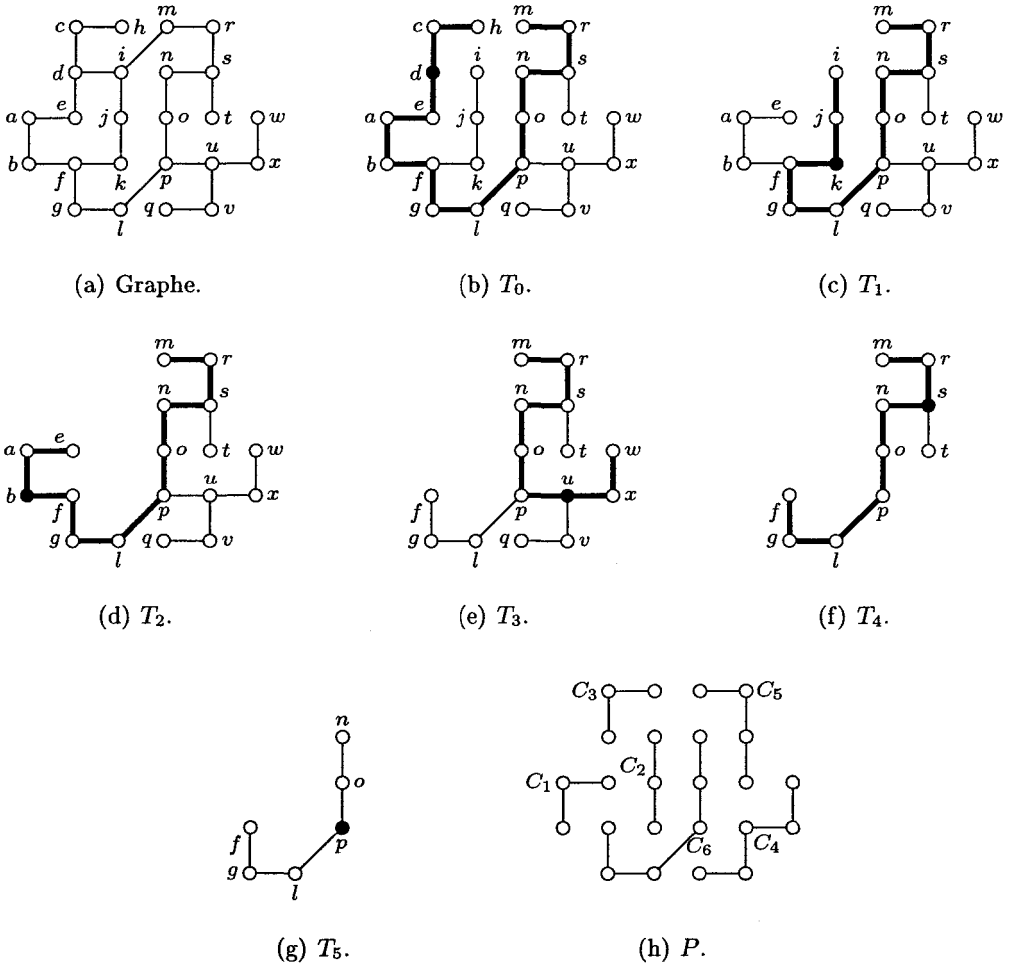
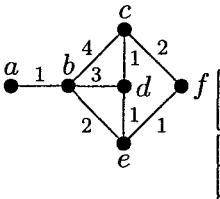


FIG. 3.6 – Algorithme de k-dominance.

Lorsqu'on veut envoyer un message d'un sommet u à un autre v sans le diffuser sur l'ensemble du réseau, il faut employer une stratégie de routage. Elle consiste à choisir les arêtes d'une chaîne U entre u et v de façon à ce que le coût total des arêtes soit minimal. Si le nombre de sommets n'est pas trop grand, il est pensable d'utiliser un tableau M_G qui donne pour chaque sommet $u \in V$ le sommet suivant dans une chaîne de coût minimal vers chacun des autres sommets du graphe. Prenons pour exemple la figure 3.7, qui illustre un graphe $G' = \{V', E'\}$ et son tableau de routage $M_{G'}$. On veut envoyer un message du sommet e vers le sommet d. La chaîne de coût minimal entre e et d est $\{e, d, c\}$. L'entrée $M_{G'}(e, c)$ est alors d, puisque c'est le sommet qui suit e dans la chaîne de coût minimal.



(a) Graphe.

Source	Destination					
	a	b	c	d	e	f
a	-	b	b	b	b	b
b	a	-	c	d	e	e
c	b	b	-	d	d	f
d	b	b	c	-	e	e
e	b	b	d	d	-	f
f	e	e	c	e	e	-

(b) Informations.

FIG. 3.7 – Informations pour une stratégie de routage optimale.

La taille d'une telle table de routage est dans $O(|V|^2)$ pour tout un réseau. Le tableau est ditribué dans le réseau, de manière à associer une sous-table M_{G_u} à chaque sommet $u \in V$. La sous-table M_{G_u} contient toutes les entrées $M_G(u, v)$ pour $v \in V$, et est donc de $O(|V|)$ éléments. On constate que si le nombre de sommets augmente, la quantité de mémoire requise dans chaque sommet augmente aussi. Les sommets représentent des structures physiques du réseau, et chacun dispose d'une quantité de mémoire limitée. Dépasser cette limite implique des coûts d'infrastructure, car il faut changer/augmenter les structures impliquées. Si l'on veut un réseau qui croît avec le temps, il faut que la consommation de mémoire soit à peu près stable, ou s'accroisse très faiblement, avec l'augmentation du nombre d'éléments connectés. La stratégie des sous-tableaux de $O(|V|)$ éléments n'est pas une bonne stratégie de

routage à cet égard.

Pour mesurer la qualité d'une stratégie de routage qui n'est pas nécessairement optimale, on observe deux facteurs. Le premier est évidemment sa consommation de mémoire. Le second est la distance moyenne parcourue par un message entre deux sommets par rapport à la distance optimale. Exprimé sous forme d'un ratio, ce rapport se nomme facteur d'étirement (« stretch factor » en anglais). Considérons une stratégie de routage RS dont le coût d'une communication entre deux sommets u et v est $C(RS, u, v)$. Le facteur d'étirement se calcule comme ceci :

$$STRETCH(RS) = \max_{u,v \in V} \left\{ \frac{|C(RS, u, v)|}{d_\gamma(u, v)} \right\} \tag{3.7}$$

Une solution optimale donne un rapport de 1. Une bonne stratégie de routage combine un facteur d'étirement le plus petit possible avec une consommation de mémoire par sommet non seulement réduite, mais indépendante de la taille du réseau.

Les stratégies de routage sont généralement constituées de deux algorithmes. Le premier prépare les informations nécessaires à chaque sommet pour assurer le routage. C'est l'algorithme de prétraitement (« preprocessing protocol » en anglais). Le second transfère des messages. C'est l'algorithme de livraison (« delivery protocol » en anglais). Dans notre cas, uniquement le prétraitement nous intéresse. Faire le prétraitement dans un graphe de réseau consiste souvent à répartir les sommets en groupes, donc à fragmenter le graphe. Les prochaines pages révisent des algorithmes de fragmentation dans le contexte du routage.

3.2.1 Stratégie de routage paramétrée pour réseaux génériques.

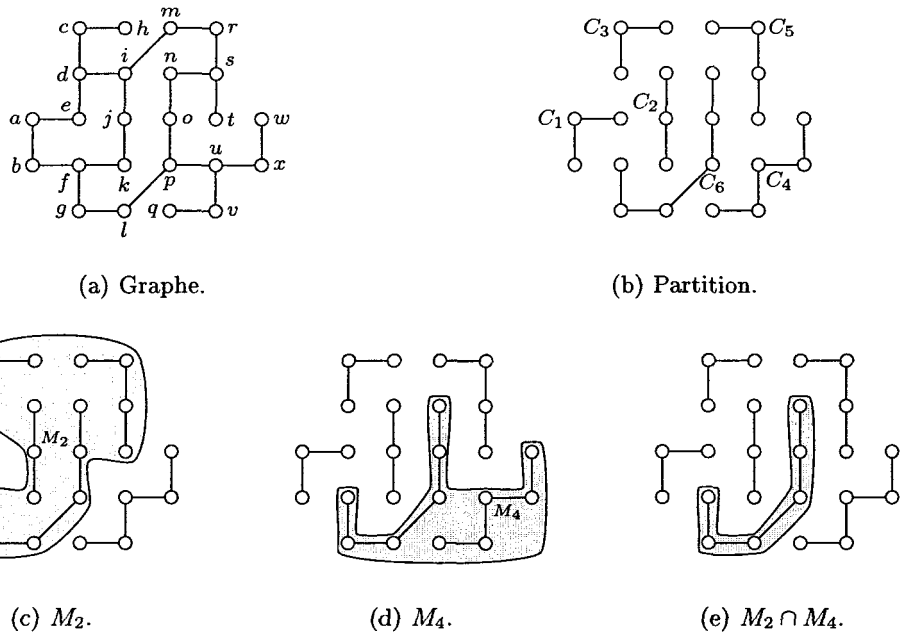
Nous ne présenterons qu'un aperçu de cet algorithme de *Peleg et Upfal.* Le lecteur trouvera la description complète dans [62]. C'est un algorithme paramétrable qui génère une stratégie de routage sur un graphe sans orientation ni valuation en permettant le choix du facteur d'étirement selon la quantité de mémoire disponible dans chaque sommet.

Tout d'abord, ajoutons une définition de distance supplémentaire à celles déjà vues en 1.1.1. La distance entre deux ensembles de sommets $M \subseteq V$ et $N \subseteq V$ d'un graphe

$G = (V, E)$ est la distance minimale entre toutes les paires de sommets $v \in M$ et $u \in N$:

$$D(M, N) = \min_{v \in M, u \in N} d(u, v) \tag{3.8}$$

L'algorithme établit une première stratégie de routage partielle, qui couvre une « région » du graphe. Deux paramètres sont utilisés : $k, m \geq 1$. Le graphe G est fragmenté par k-dominance en $P = \{C_1, C_2, .., C_n\}$. Le paramètre m détermine la taille d'une région. Chaque sous-graphe $C_i = (V_i, E_i)$ contient un sommet c_i, son *chef* (« leader »), tel que la distance entre un sommet $v \in V_i$ et c_i est au plus k. L'algorithme relève d'une définition plus souple de partition que la nôtre, les sommets pouvant appartenir à plus d'un sous-graphe. Pour chaque sous-graphe C_i, l'algorithme construit une région M_i constituée de tous les sous-graphes de P distants d'au plus m de C_i. Les figures 3.8(c) et 3.8(d) illustrent des régions du graphe 3.8(a) avec $m = 1$ et $k = 3$.



(a) Graphe. (b) Partition.

(c) M_2. (d) M_4. (e) $M_2 \cap M_4$.

FIG. 3.8 – Exemples de régions pour $m = 1$.

Le routage est direct entre le chef d'un sous-graphe C_i et les chefs des sous-graphes

de sa région M_i. Le routage à l'intérieur d'un sous-graphe C_i se fait par arbre d'intervalle (« Interval Tree Routing » en anglais). Il s'agit en gros d'un arbre recouvrant de C_i ayant pour racine c_i et où chaque sommet dispose d'informations pour atteindre les autres à partir de ses arêtes. En plus des informations de l'arbre d'intervalle, chaque sommet $v \in V_i$ connaît son chef c_i. Une communication entre un sommet $u \in C_i$ et $v \in C_j$ $(i \neq j)$ se fait ainsi : si C_j est dans M_i, le message part de u, se rend à c_i par l'arbre d'intervalle de C_i, passe de c_i à c_j par un lien direct et finalement rejoint v par l'arbre d'intervalle de C_j. Si C_j n'est pas dans M_i, la communication est alors bloquée.

Il faut assurer la communication entre chaque paire de sommets du réseau, et non pas uniquement entre paires d'une région. C'est pourquoi cette stratégie de routage est hiérarchique. Elle est appliquée plusieurs fois pour des valeurs de m croissantes. Si la taille de m ne permet pas d'établir la communication, on incrémente m, la plus grande valeur permettant une communication à travers tout le graphe.

L'algorithme de fragmentation de cette stratégie de routage construit des sous-graphes de rayon k, ce qui correspond à faire la k-dominance. Les régions se chevauchent : si le sous-graphe C_i est dans la région M_j, alors le sous-graphe C_j est nécessairement dans M_i. Les régions ne forment donc pas une partition du graphe. La figure 3.8(e) illustre le chevauchement de deux régions. Ce sont les idées à la base de cette stratégie de routage qui sont intéressantes, dont deux particulièrement. Premièrement, il y a le concept de régions, à savoir de regrouper des sous-graphes avoisinants de la partition selon des critères choisis. Deuxièmement, il y a la hiérachisation, qui est à la base de l'algorithme suivant.

3.2.2 Fragmentation par voisinages volumétriques.

Voici un algorithme de fragmentation construit à partir de deux algorithmes de routage expliqués dans [5, 26] qui héritent des idées de hiérachisation de l'algorithme de *Peleg et Upfal.* Il divise le graphe selon une hiérarchie basée sur une nouvelle définition de voisinage et sur le concept d'ensembles couvrants.

Les définitions de voisinage vues jusqu'ici, et employées dans les algorithmes de

fragmentation par cliques et par dominance, ainsi que dans la stratégie de routage précédente, étaient toutes construites à partir de rayons. La fragmentation par voisinages volumétriques utilise une définition de voisinage dont le but est d'obtenir une quantité spécifique de sommets. Supposons un graphe connexe $G = (V, U)$ étiqueté par $\lambda : V \rightarrow L$, une fonction injective et supposons que l'ensemble d'étiquettes L est muni d'un ordre total. La fonction λ étant injective, elle induit un ordre total sur V. On définit une nouvelle relation d'ordre total $\prec_v$ sur un sommet v quelconque de V. La relation va comme suit :

$$x \prec_v y \Rightarrow \begin{cases} d(x,v) < d(y,v) \\ \text{ou} \\ d(x,v) = d(y,v) \text{ et } \lambda(x) < \lambda(y) \end{cases} \tag{3.9}$$

Si le graphe est valué par une fonction $\gamma : E \rightarrow \Re$ en plus d'être étiqueté, la distance pondérée est utilisée :

$$x \prec_v y \Rightarrow \begin{cases} d_\gamma(x,v) < d_\gamma(y,v) \\ \text{ou} \\ d_\gamma(x,v) = d_\gamma(y,v) \text{ et } \lambda(x) < \lambda(y) \end{cases} \tag{3.10}$$

Le voisinage volumétrique de taille i de v s'écrit $N(v,i)$ et contient les i premiers sommets de V selon l'ordre total $\prec_v$. Le voisinage volumétrique de v de taille i par rapport au sous-ensemble $S \subseteq V$, ou $N(v,i,S)$, contient les i-premiers sommets de S selon l'ordre total $\prec_v$. Le rayon d'un voisinage volumétrique, $\mathrm{r}(v,i,S)$, c'est le rayon du sous-graphe induit par les sommets de ce voisinage. Les sommets pleins de la figure 3.9 forment le voisinage $N(v_1, 10)$ du graphe, avec $r(v_1, 10, V) = 2$.

Considérons un ensemble B et une collection $\boldsymbol{H}$ de sous-ensembles $H_i \subset B$ quelconques non-vides telle que :

$$\bigcup_{i=1}^{|\boldsymbol{H}|} H_i = B \tag{3.11}$$

Nous dirons qu'un sous-ensemble $M \subseteq B$ couvre $\boldsymbol{H}$, ou forme une couverture de $\boldsymbol{H}$, si M a

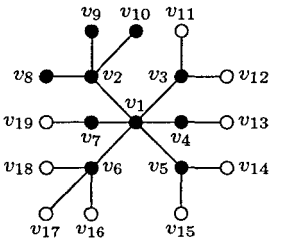


FIG. 3.9 – $N(v_1, 10)$.

au moins un élément commun avec chacun des sous-ensembles contenus dans $\boldsymbol{H}$:

$$\forall i, H_i \cap M \neq \emptyset \tag{3.12}$$

Dans [17], on introduit un algorithme vorace pour calculer un ensemble couvrant M pour une collection de sous-ensembles $\boldsymbol{H}$ de B. Cet algorithme (algorithme 3.4) est basé sur l'ajout itératif de sommets jusqu'à obtenir un ensemble couvrant. La taille de cet ensemble est en proportion à croissance logarithmique par rapport à la taille de l'ensemble couvrant minimale. On commence en posant $M = \emptyset$. Ensuite, à chaque itération, on identifie un sommet m de $B - M$ qui couvre le plus d'ensembles de la collection H. On ajoute m à M. Si M est une couverture, l'algorithme termine, sinon on procède à une nouvelle itération. La couverture ainsi obtenue est désignée couverture vorace de H.

Le but de l'algorithme de fragmentation est de trouver des points centraux de communication pour le routage, ou *pivots*. Les pivots sont regroupés en niveaux selon un paramètre k. Il y a $k + 1$ niveaux. Chaque niveau R_i (pour $0 \leq i \leq k$) contient un certain nombre de pivots. Au niveau R_0, chacun des sommets du graphe est un pivot. Les pivots des niveaux suivants sont trouvés itérativement. L'ensemble des pivots R_{i+1} est une couverture sur la collection E_{R_i} constituée des voisinages volumétriques de taille m des sommets-pivots R_i :

$$E_{R_i} = \bigcup_{r \in R_i} \{N(r, m, R_i)\} \tag{3.13}$$

Algorithme 3.4 CouvertureVorace(B, H, M)

```
M ← ∅
m ← ∅
couvre ← 0
couverture ← FALSE
tant que couverture = FALSE } faire
        B' ← B − M
        pour tout b ∈ B' faire
                cover ← 0
                pour tout H_i ∈ (H − H') faire
                        si b ∈ H_i alors
                                cover ← cover + 1
                si  cover > currcover  alors
                        currcover ← cover
                        m ← b
        M ← M ∪ {m}
        H' ← ∅
        pour tout H_i ∈ H faire
                si H_i ∩ M ≠ ∅ alors
                        H' ← H' ∪ {H_i}
        si H' = H alors
                couverture ← TRUE
```

La valeur de m est fixée à $n^{1/k}$ par les concepteurs. Cette valeur a été choisie pour obtenir les caractéristiques de routage (consommation de mémoire, facteur d'étirement) recherchées. Considérons la figure 3.10. Avec $k = 3$, on obtient $m \approx 2.15$, arrondi à 2. Le tableau 3.2 montre le calcul des pivots des différents niveaux.

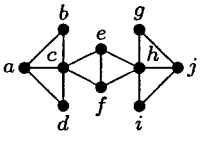


FIG. 3.10 – Un graphe.

Les sous-graphes de la partition P_i sont constitués des sommets des voisinages volumétriques de la collection $E_{R_{i-1}}$ qui partagent le même pivot de R_i. On s'assure par la suite qu'un sommet n'est présent que dans un seul sous-graphe, les pivots étant chacun dans le leur. Par exemple, pour les pivots R_1 de 3.2, on obtient $\{\{a, b, c, d\}, \{a, c, f\}, \{e, g, h, i\}, \{g, j\}\}$.

It	Résultats pour $m = 2$	
0	R_0	$= \{a, b, c, d, e, f, g, h, i, j\}$
	E_{R_0}	$= \{\{a, b\}, \{b, a\}, \{c, a\}, \{d, a\}, \{e, c\}, \{f, c\}, \{g, h\}, \{h, e\}, \{i, h\}, \{j, g\}\}$
1	R_1	$= \{a, h, c, j\}$
	E_{R_1}	$= \{\{a, c\}, \{c, a\}, \{h, j\}, \{j, h\}\}$
2	R_2	$= \{a, h\}$
	E_{R_2}	$= \{\{a, h\}, \{h, a\}\}$
3	R_3	$= \{a, b, c, d, e, f, g, h, i, j\}$

TAB. 3.2 – Pivots du graphe de la figure 3.10.

Si l'on s'assure que chaque sommet n'est présent qu'une fois, il en résulte la partition $P_i = \{\{a, b, d\}, \{c, f\}, \{e, h, i\}, \{g, j\}\}$.

3.3 Méthodes de l'ingénierie inverse du logiciel.

Voici deux algorithmes de fragmentation utilisés pour des opérations d'ingénierie inverse. Le premier, Bunch, est un algorithme d'optimisation. Le second, la fragmentation par filtration d'arêtes, utilise une valuation pour identifier les arêtes à enlever.

3.3.1 Bunch.

Une autre approche pour fragmenter un graphe consiste à réduire le problème à un problème d'optimisation. Pour cela, il faut déterminer un critère quantifiable de ce qu'est une bonne partition. Ensuite, il suffit d'explorer l'espace des partitions possibles pour repérer la partition qui donne le meilleur résultat selon ce critère quantifiable. Cette approche est utilisée par *Mancoridis et al.* [21, 50, 51, 55, 56].

Dans le contexte du génie logiciel, la qualité d'une partition du graphe dépend de la connectivité interne et de la connectivité externe des sous-graphes. Plus la connectivité interne d'un sous-graphe de la partition est forte, plus il y a dépendance mutuelle entre les éléments logiciels représentés par ses sommets. D'un autre côté, une faible connectivité externe de ce sous-graphe implique que les interactions de ses éléments logiciels avec le reste du système étudié sont limitées. Étant donné un graphe servant à modéliser un logiciel, une partition optimale aura une très forte connectivité interne et une très faible connectivité externe. Considérons

que chaque sous-graphe correspond à un sous-système de notre système logiciel. La partition optimale offre de nombreux avantages pour l'analyse d'un logiciel et sa maintenance. Une faible connectivité implique que les communications et les interactions entre sous-systèmes sont limitées. Maintenance et réutilisation sont facilitées, puisqu'il est plus simple d'isoler un sous-système et de contrôler ou tester ses interactions. La simplicité des communications entre sous-systèmes aide à trouver leurs rôles respectifs, donc à synthétiser une abstraction du système. *Mancoridis et al.* ont introduit une mesure de la qualité de modularisation (ou MQ pour « Modularization Quality ») d'une partition d'un graphe, dont nous donnerons la définition. Pour les définitions suivantes, nous considérerons une k-partition $P = \{C_1, C_2, .., C_k\}$ d'un graphe $G = (V, E)$.

La mesure de la connectivité interne A_i d'un sous-graphe C_i constitué de n_i sommets et μ_i arêtes internes se calcule en faisant le rapport entre le nombre d'arêtes du sous-graphe C_i et le nombre d'arêtes du graphe complet d'ordre n_i :

$$A_i = \frac{\mu_i}{\binom{n_i}{2}} \tag{3.14}$$

Une valeur de 0 implique qu'il n'y a aucune connectivité interne. Une valeur de 1 exprime une connectivité complète, toutes les paires de sommets étant liées par une arête (C_i est isomorphe au graphe complet). La connectivité interne moyenne d'une partition est la moyenne de connectivité interne de ses sous-graphes :

$$A_P = \frac{1}{k} \sum_{i=1}^{k} A_i \tag{3.15}$$

La mesure de la connectivité externe $E_{i,j}$ entre deux sous-graphes C_i et C_j se calcule en comparant le nombre de relations présentes par rapport au maximum possible :

$$E_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \frac{\varepsilon_{i,j}}{2n_i n_j} & \text{si } i \neq j \end{cases} \tag{3.16}$$

où $\varepsilon_{i,j}$ est le nombre d'arêtes entre les sous-graphes C_i et C_j, n_i le nombre de sommets de C_i et n_j le nombre de sommets de C_j. La valeur de $E_{i,j}$ varie entre 0 et 1, 0 indiquant qu'il n'y a pas de liens entre les sous-graphes et 1 que chaque élément de C_i est lié à tous ceux de C_j et inversément. La connectivité externe moyenne de la partition est :

$$E_P = \frac{1}{\frac{k(k-1)}{2}} \sum_{i,j=1}^{k} E_{i,j} \tag{3.17}$$

La MQ pénalise la connectivité externe tout en récompensant la connectivité interne. La qualité de modularisation d'une partition se calcule en soustrayant sa connectivité externe moyenne à sa connectivité interne moyenne :

$$MQ_P = A_P - E_P \tag{3.18}$$

Chiricota et al. [16] ont effectué une étude statistique du comportement de la MQ. À partir d'un ensemble de partitions générées aléatoirement dans un graphe, ils ont obtenu les caractéristiques suivantes : une moyenne d'environ -0.2 et un écart-type de 0.2. Ces informations permettent de penser qu'une MQ supérieure ou égale à 0.05 est signe d'une bonne partition du graphe.

Le logiciel Bunch.

Bunch est une collection de quatre algorithmes qui visent à calculer une partition avec la valeur de MQ optimale (la plus haute) ou encore une partition avec une MQ considérée « quasi-optimale » (une MQ dont la valeur est jugée relativement haute). Le premier algorithme est optimal et donne une partition avec une MQ maximale, les autres algorithmes sont des heuristiques connues du domaine des problèmes d'optimisation, modifiées pour trouver une bonne partition. Ils donnent une solution en explorant une partie de l'espace des partitions et en retournant ce qu'ils y ont trouvé de meilleur.

Deux des algorithmes heuristiques utilisent un concept de voisinage d'une partition P d'un graphe G défini comme suit : Une partition $P' = \{C'_1, C'_2, ..., C'_n\}$ de G est voisine de

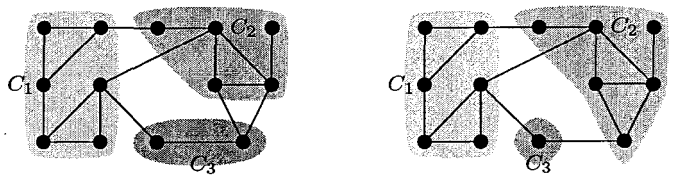
$P = \{C_1, C_2, ..., C_m\}$ s'il existe deux sous-graphes C_i de P et C'_j de P' tels que :

$$C_i - C'_i = \{v\} \text{ et } C'_j - C_j = \{v\} \tag{3.19}$$

pour $i \neq j$. De plus, les autres sous-graphes de P et P' respectent la propriété suivante :

$$C_x = C'_x \tag{3.20}$$

pour toute valeur de x différente de i et de j. La figure 3.11 illustre une partition P et une voisine de P.



(a) Une partition P. (b) Voisine de P.

FIG. 3.11 – Exemple d'une partition voisine d'une partition P.

Voici les quatre algorithmes.

Optimal C'est l'algorithme du modèle théorique. Il explore tout l'espace des partitions. Considérant ce qui a été dit sur la taille de cet espace, il n'est pas surprenant que l'algorithme optimal n'est employé que pour les graphes de taille réduite.

SAHC (*Steepest Ascent Hill Climbing*) Cet algorithme génère aléatoirement une première partition P. Par la suite, les opérations suivantes sont appliquées itérativement :

- On obtient la collection $\boldsymbol{E}$ de toutes les partitions voisines de P.
- On explore $\boldsymbol{E}$ d'où on extrait une partition P' de MQ maximale.
- Si la MQ de P' est supérieure à la MQ de P, P' remplace P et on itère de nouveau.
- Sinon, P est considérée comme étant une solution locale optimale.

Le maximum local P est retourné par l'algorithme comme étant une solution quasi-

optimale.

NAHC (*Next Ascent hill Climbing*) Cet algorithme diffère de SAHC dans sa manière de sélectionner P'. Au lieu de prendre une des meilleures partitions voisines dans $\boldsymbol{E}$ à chaque itération, l'algorithme sélectionne aléatoirement une partition P' de $\boldsymbol{E}$ telle que $MQ(P') > MQ(P)$. Si cette partition n'existe pas, P est retournée par l'algorithme comme étant une solution quasi-optimale.

GA (*Genetic Algorithm*)Un premier ensemble de partitions (la population) est généré aléatoirement. Pendant un nombre d'itérations fixé, des sélections et des croisements (mélanges de partitions), accompagnés de mutations (changements aléatoires) sont effectués pour obtenir de nouvelles populations. Une partition avec la MQ la plus élevée dans la dernière population est retournée. Voir [21] pour plus de détails sur cet algorithme.

Les premières versions de Bunch ne contenaient que ces algorithmes. En discutant avec des concepteurs et d'autres intervenants de l'industrie, *Mancoridis et al.* se sont aperçus d'un manque de souplesse de leur outil. Certains cas d'exception, de même que les informations disponibles sur le système logiciel étudié, devaient être considérés dans le processus de recherche d'une bonne partition [50]. C'est pourquoi les caractéristiques suivantes ont été ajoutées à Bunch :

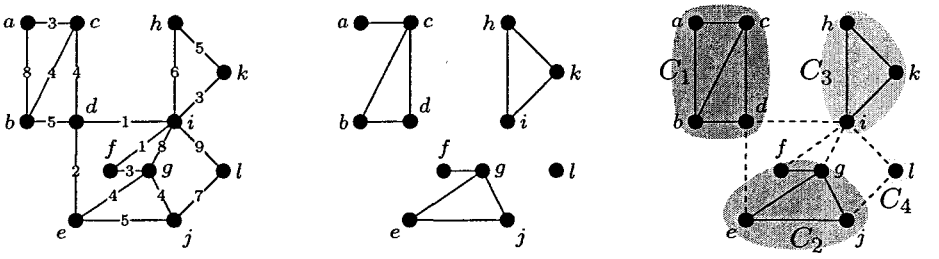
Directions de l'utilisateur L'utilisateur peut fournir de l'information déjà connue sur le système que Bunch doit considérer dans sa recherche automatisée d'une partition.

Isolation de modules fortement connexes Certains éléments du système logiciel sont utilisés partout. Leur présence complexifie la recherche d'une bonne partition. L'utilisateur conscient de ce problème peut identifier ces éléments manuellement ou à l'aide d'outils pour qu'ils soient isolés.

Maintenance incrémentale Avec les premières versions de Bunch, l'ajout d'un seul élément à un système logiciel pouvait causer d'importants changements à une partition existante. Les nouvelles versions de Bunch ont été modifiées afin de permettre l'ajout d'éléments sans pour autant obtenir des partitions radicalement différentes.

3.3.2 Fragmentation de graphes par filtration d'arêtes.

Il est possible de calculer une partition d'un graphe $G = (V, E)$ valué par une fonction de coût $\gamma : E \rightarrow \Re$ en filtrant les arêtes. Il suffit de construire le graphe partiel $G' = (V, E')$, tel qu'une arête $e \in E$ est présente dans E' seulement si son poids entre dans un intervalle de valeurs choisies. Le graphe G' ainsi créé sert à définir une partition P. Chaque composante connexe $C'_i = (V_i, E'_i)$ de G' forme un sous-graphe $C_i = (V_i, E_i)$ de P. La figure 3.12(a) donne l'exemple d'une partition obtenue en filtrant les arêtes dont les valeurs sont dans l'intervalle $[3, 6]$.



(a) G. (b) G'. (c) P.

FIG. 3.12 – Filtration d'arêtes.

Chiricota et al. [16] ont employé cette technique pour fragmenter des graphes issus du génie logiciel. Ils ont introduit une valuation permettant une filtration adéquate de tels graphes. La valuation en question, nommée « strength », est basée sur l'indice de fragmentation. L'indice de fragmentation du sommet v d'un graphe $G = (V, E)$ est une valuation de sommets $c(v)$ qui mesure le rapport entre le nombre d'arêtes des voisins de v et le nombre maximal d'arêtes possibles entre ceux-ci. Il est défini par la formule suivante :

$$c(v) = \frac{e(N(v))}{\binom{|N(v)|}{2}} \tag{3.21}$$

où $e(N(v))$ est le nombre d'arêtes dans le voisinage de v et où $\binom{|N(v)|}{2}$ est le coefficient binomial représentant le nombre maximal de combinaisons possibles de deux éléments parmi $|N(v)|$,

c'est-à-dire la taille (en arêtes) d'un graphe complet de $|N(v)|$ sommets.

La valuation force de cohésion, ou strength, applique le même principe que l'indice de fragmentation : trouver la quantité d'arêtes dans le voisinage par rapport au maximum. La pondération se fait sur les arêtes plutôt que sur les sommets. La valuation force de cohésion a pour but de donner une idée de l'importance de l'arête dans la connectivité du graphe, autrement dit les chances qu'enlever cette arête sépare le graphe en deux composantes connexes. Cette valuation permet d'extraire les « clusters » (agrégats de sommets) inhérents à la structure du graphe.

Définissons $S(U, V)$ comme étant la proportion des arêtes présentes entre deux ensembles distincts de sommets U et V par rapport au maximum possible, et $S(U)$ comme étant le nombre d'arêtes entre sommets de U par rapport au nombre d'arêtes du graphe complet d'ordre $|U|$. La valeur $S(U, V)$ se calcule ainsi :

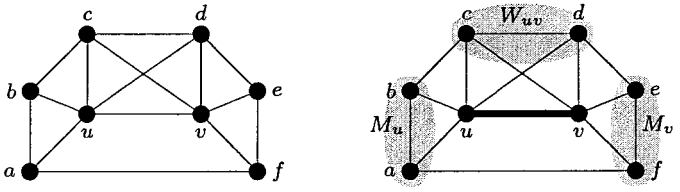
$$S(U,V) = \begin{cases} 0 & \text{si } U = \emptyset \text{ ou } V = \emptyset \\ \frac{e(U,V)}{|U||V|} & \text{sinon} \end{cases} \tag{3.22}$$

et la valeur $S(U)$ avec la formule suivante :

$$S(U) = \begin{cases} 0 & \text{si } U = \emptyset \\ \frac{e(U)}{\binom{|U|}{2}} & \text{sinon} \end{cases} \tag{3.23}$$

Dans ces formules, $e(U, V)$ représente le nombre d'arêtes entre U et V et $e(U)$ le nombre d'arêtes dont les deux sommets incidents sont dans U.

Considérons une arête $e = \{u, v\}$ d'un graphe $G = (V, E)$. Chaque sommet des voisinages de u et v appartient à l'un des ensembles suivants : M_u, l'ensemble des sommets uniquement dans $N(u)$; M_v, l'ensemble des sommets uniquement dans $N(v)$; W_{uv}, l'ensemble des sommets dans $N(u)$ et $N(v)$ (i.e. $N(u) \cap N(v)$). Les ensembles obtenus forment une partition du sous-graphe formé par les sommets $N(u) \cup N(v)$. C'est ce qu'illustre la figure 3.13(b) pour l'arête $\{u, v\}$ du graphe 3.13(a).

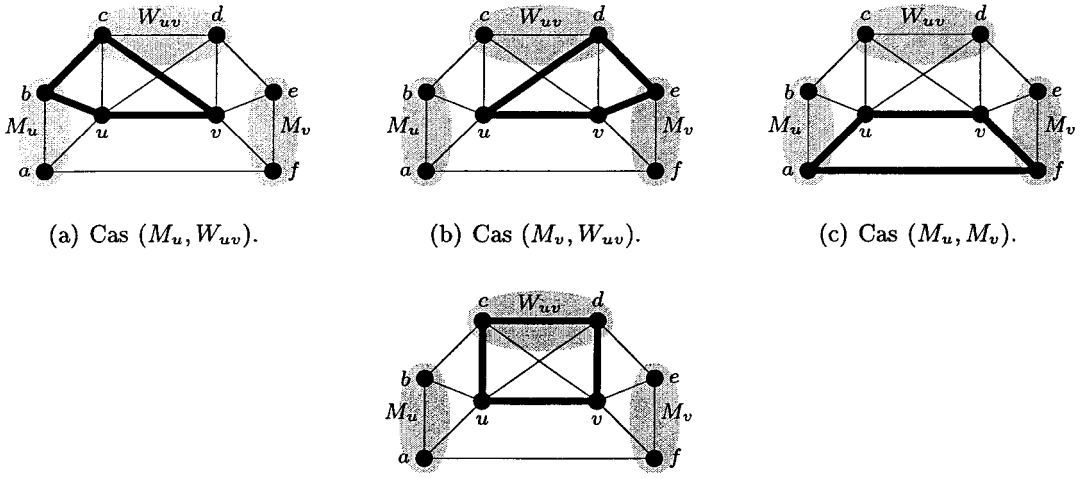


(a) (b)

FIG. 3.13 – Fragmentation des voisinages dans Strength.

On classifie les 4-cycles (cycles de longueur 4) qui passent par l'arête e à l'aide de la partition P_{uv} de $N(u) \cup N(v)$ en M_u, M_v et W_{uv}. Il y a quatre cas possibles, illustrés par la figure 3.14 :

1. Un sommet du 4-cycle est dans M_u et un autre est dans W_{uv}
2. Un sommet du 4-cycle est dans M_v et un autre est dans W_{uv}
3. Un sommet du 4-cycle est dans M_u et un autre est dans M_v
4. Deux sommets du 4-cycle sont dans W_{uv}



(a) Cas (M_u, W_{uv}). (b) Cas (M_v, W_{uv}). (c) Cas (M_u, M_v).

(d) Cas W_{uv}.

FIG. 3.14 – Types de 4-cycles passant par $\{u, v\}$.

La densité de 4-cycles passant par $e = \{u, v\}$, ou $\gamma_4(e)$, est calculée en additionnant

la densité d'arêtes pour chacun des quatre cas :

$$\gamma_4(e) = S(M_u, W_{uv}) + S(M_v, W_{uv}) + S(M_u, M_v) + S(W_{uv}) \tag{3.24}$$

Les 3-cycles (cycles de longueur 3) passant par l'arête e ne correspondent qu'à un seul cas. Si (u, c, v) est un 3-cycle, le sommet c ne peut qu'être voisin à la fois de u et de v, donc appartenir à W_{uv}. La figure 3.15 illustre un 3-cycle.

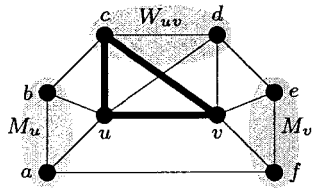


FIG. 3.15 – 3-Cycle passant par $\{u, v\}$.

La densité de 3-cycles ($\gamma_3(e)$) se mesure en comparant la quantité de sommets qui sont voisins des deux extrémités de l'arête e avec le nombre de sommets qui sont voisins de l'une extrémité ou de l'autre :

$$\gamma_3(e) = \frac{|W_{uv}|}{|M_u| + |M_v| + |W_{uv}|} \tag{3.25}$$

La force de cohésion de e, ou $\Sigma(e)$, est la somme de $\gamma_3(e)$ et de $\gamma_4(e)$:

$$\Sigma(e) = \gamma_3(e) + \gamma_4(e) \tag{3.26}$$

Plus la force de cohésion est forte, plus l'arête e a des chances d'être une arête interne à l'un des sous-graphes d'une partition établie selon la connectivité. À l'opposé, plus elle est petite, plus l'arête a des chances d'être une relation externe. Un isthme $f = \{x, y\}$ a une valeur de 0 puisque, par définition, $W_{xy} = \emptyset$ et $e(M_x, M_y) = 0$. On obtient alors $\gamma_3(f) = 0$ et $\gamma_4(f) = 0$, donc une valeur $\Sigma(f)$ nulle.

Fragmentation par force de cohésion.

La force de cohésion vient compléter la filtration des arêtes pour produire des partitions de graphes similaires à Bunch, qui favorisent une forte connectivité intra-composantes et un faible nombre d'arêtes inter-composantes. La figure 3.16 illustre la création de partitions à l'aide de cette technique. Le graphe 3.16(a) est valué par la force de cohésion (figure 3.16(b)). La grosseur des arêtes est proportionnelle à leur poids. Une filtration est utilisée pour obtenir un graphe partiel (figure 3.16(c)) dont les composantes connexes (figure 3.16(d)) vont déterminer les sous-graphes de la partition. Les résultats obtenus par cette approche sont comparables à ceux de Bunch, mais à un coût temporel nettement moindre et sans les éléments aléatoires que l'on retrouve dans les heuristiques de Bunch.

3.4 Discussion.

Les algorithmes que nous venons d'étudier se classifient en trois catégories, selon les critères qui déterminent la répartition des sommets dans les sous-graphes. Il y a les algorithmes basés sur la connectivité dans le graphe, les algorithmes basés sur les distances dans le graphe et les algorithmes basés sur la cardinalité des sous-graphes. Le tableau 3.3 présente la classification obtenue.

Méthodes					
cardinalité		distances		connectivité	
Voisinage volumétrique	3.2.2	Cliques k-distantes	3.1.2	Bunch	3.3.1
		Dominance	3.1.3	Filtration d'arêtes pondérées par force de cohésion	3.3.2
		k-dominance	3.1.4	Coloration	3.1.1
		Routage paramétré pour réseaux génériques	3.2.1		

TAB. 3.3 – Classification des méthodes de fragmentation.

L'algorithme de fragmentation qui convient le moins pour remplir nos besoins est l'algorithme par k-coloration. La partition obtenue, avec ses sous-graphes sans connectivité

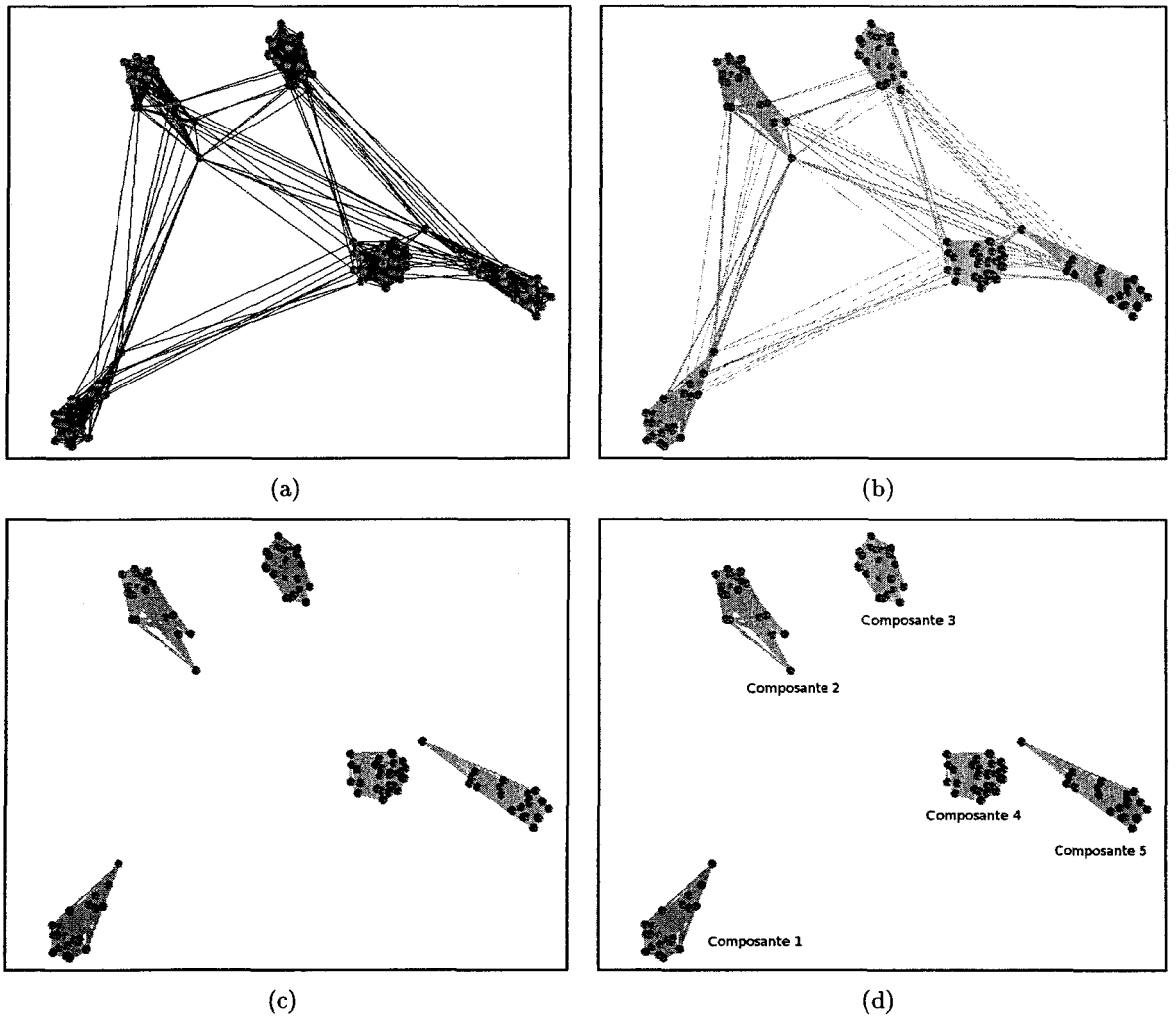
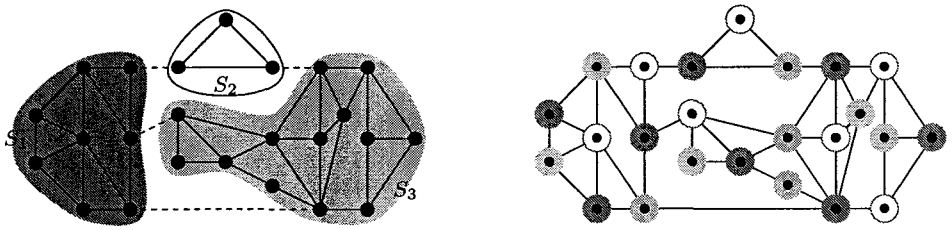


FIG. 3.16 – Filtration des arêtes à l'aide de la force de cohésion.

interne, va à l'encontre de la pensée logicielle. Idéalement, un système logiciel est composé de sous-systèmes dont les sommets sont connexes. Or, la coloration sépare les sommets adjacents, ce qui a pour effet de répartir les sous-systèmes à travers les sous-graphes de la partition. C'est ce que montre la figure 3.17. Le graphe 3.17(a) représente un logiciel composé de trois sous-systèmes $\{S_1, S_2, S_3\}$. La figure 3.17(b) illustre la répartition des sommets en couleurs selon une 3-coloration. Il est évident que la répartition obtenue n'a rien à voir avec la structure du système logiciel.



(a) Graphe logiciel G.

(b) Coloration de G.

FIG. 3.17 – Sous-systèmes versus sous-graphes d'une coloration.

À première vue, les cliques k-distantes semblent convenir pour générer de bonnes partitions de graphes selon les critères exposés (voir 2.2.2 et 2.2.3). Les informations utilisées sont purement topologiques et la fragmentation est paramétrée à travers k. Ce paramètre est une source de problèmes par rapport à notre but d'obtenir une partition qui reconnaît les composantes d'un système logiciel. Le paramètre k vient limiter les sous-graphes de notre partition. Aucun sous-graphe ne peut dépasser un diamètre de k. Généralement, les sous-systèmes d'un système logiciel sont de tailles et de formes hautement variables. Avec un diamètre restreint, les gros sous-systèmes vont être séparés en plusieurs sous-graphes plus petits alors que les petits sous-systèmes risquent d'être regroupés. Le diamètre doit servir de référence, et non pas constituer un absolu.

L'algorithme de fragmentation par dominance par distances souffre du même problème, et pour une raison très simple. Considérons un sous-graphe C quelconque produit à partir de la dominance par distance. Le sommet k-dominant de C est w. Comme on l'a vu

précédemment, tout sommet de C est à une distance d'au plus k de w. Considérons aussi deux sommets u et v de C, chacun à distance k de w. Dans le pire cas, le chemin simple le plus court allant de u et v est de longueur $2k$, et passe par w. La figure 3.18 est un exemple du pire cas pour $k = 2$. Le sous-graphe C est alors une clique de diamètre $2k$. Trouver une partition à partir de la k-dominance est donc équivalent à trouver une partition composée de cliques d-distantes où $d = 2k$. Utiliser la dominance pour fragmenter le graphe revient à construire une partition par k-dominance avec $k = 1$, donc à construire une partition de cliques 2-distantes. De même, la stratégie de routage paramétrée fragmente avec un algorithme de k-dominance.

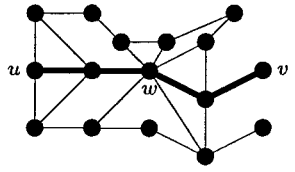


FIG. 3.18 – Pire cas de diamètre pour $k = 2$.

La fragmentation par voisinages volumétriques aborde le problème autrement et tente d'obtenir des sous-graphes d'un même ordre. Cependant, cette méthode est peu paramétrable et ses paramètres donnent des résultats que l'utilisateur ne peut pas aisément prévoir. En effet, il n'y a que deux paramètres : le nombre k de niveaux hiérarchiques dans le processus de fragmentation et le niveau i (pour $1 \leq i \leq k$) qui sera étudié. L'utilisateur n'est pas en mesure de prévoir l'effet qu'aura un changement de la valeur de k, puisque changer le nombre de niveaux modifie la grosseur des voisinages et par le fait même le choix des pivots, donc la fragmentation du graphe en entier. Trouver le résultat tient alors plus de la chance que de la réflexion de l'utilisateur. La valeur i, permet uniquement de raffiner la complexité de la partition obtenue. Plus i est petit, plus la partition contient de sous-graphes. Cependant, les différentes partitions sont très ressemblantes. Un autre problème de l'approche volumétrique est qu'elle tend à produire des sous-graphes de taille uniforme. C'est une limitation importante pour le domaine de l'ingénierie du logiciel, puisque les sous-systèmes d'un logiciel peuvent être

de tailles très variées.

Bunch est un outil conçu pour être entièrement automatisé. L'entrée d'informations par l'utilisateur n'y est considérée qu'à cause de commentaires de l'industrie et vient même à l'encontre du but déclaré de Bunch (fragmentation par MQ). Dans une situation où l'espace des solutions possibles est entièrement explorable dans un temps raisonnable, l'utilisateur n'est d'aucune nécessité puisque l'algorithme est conçu pour produire la meilleure solution selon son critère de MQ. Lorsque l'espace de solution n'est plus explorable, l'utilisateur ne sert qu'a spécifier l'heuristique et, optionnellement, fournir un point de départ à celle-ci. Il n'a aucun contrôle de la solution obtenue. En soi, ce ne serait pas une mauvaise chose si ce n'était des limitations du critère utilisé. Le critère d'une bonne partition selon Bunch est idéaliste. Ce ne sont pas tous les systèmes logiciels qui vont avoir une structure propice à une bonne MQ, avec des sous-systèmes à forte connectivité interne et faible connectivité externe. Les systèmes logiciels sont créés par des humains, qui ne les construisent pas toujours selon la théorie. Même bien construit, un système qui a subi les effets secondaires de nombreuses années de maintenance n'a pas nécessairement conservé sa structure (voir 2.1.1). Alors, rien ne garantit que la partition optimale ou quasi-optimale selon la MQ est une partition valide selon nos critères du génie logiciel. L'utilisateur n'ayant presque aucun contrôle sur le résultat, il n'est pas en mesure de corriger les erreurs de l'algorithme.

Une autre faiblesse de Bunch est dans l'emploi d'heuristiques pour obtenir une solution quasi-optimale. Les heuristiques fonctionnent à partir d'éléments aléatoires. Il en résulte qu'un même graphe ne donnera pas la même partition d'une exécution de Bunch à l'autre, sauf si l'algorithme optimal est sélectionné. Il est impossible pour l'utilisateur de prévoir la solution. Corriger le manque de prévisibilité en employant uniquement l'algorithme optimal de Bunch n'est pas réaliste. La croissance exponentielle de l'espace des partitions possibles d'un graphe selon son ordre (voir 1.1.3) implique des temps de calculs beaucoup trop longs, sauf pour un graphe très petit.

La filtration des arêtes valuées par force de cohésion souffre d'une certaine myopie sur l'espace des partitions. Pour un graphe donné, la valuation ne produit qu'un seul résultat.

Le filtre appliqué par la suite ne fait que changer la quantité de composantes, comme le choix du niveau change la complexité de la partition dans l'algorithme par voisinages volumétriques. L'existence d'un paramètre donne un meilleur contrôle à l'utilisateur que dans le cas de Bunch, mais ce contrôle reste quand même trop limité pour nos besoins.

Nous avons besoin d'une solution qui combine les qualités des différentes approches vues jusqu'ici, mais qui évite les défauts d'imprévisibilité, de taille fixée ou de manque de paramètres. Nous allons élaborer une telle solution dans le prochain chapitre.

CHAPITRE 4

FRAGMENTATION PAR NOYAUX DE RAYON k.

Nous avons vu dans le chapitre précédent différents algorithmes de fragmentation de graphes qui ne rencontrent pas entièrement les objectifs décrits en 2.2.2 et 2.2.3. Certains n'offrent pas assez de contrôle à l'utilisateur, d'autres produisent des partitions qui ne respectent pas nos critères d'ingénierie inverse. C'est pourquoi nous introduisons un nouvel algorithme de fragmentation rencontrant nos objectifs. La première partie du chapitre portera sur la description de cet algorithme. Par la suite, nous présenterons son implémentation de manière détaillée. Nous terminerons le chapitre par un exemple d'application.

4.1 Description de l'algorithme.

L'algorithme que nous introduisons combine la fragmentation par distances à la fragmentation par connectivité. Les algorithmes par distances vus au chapitre 3 effectuent généralement deux opérations lors de la fragmentation d'un graphe : ils sélectionnent des sommets de référence et répartissent en sous-graphes de la partition les sommets du graphe. Un sommet de référence est un sommet à partir duquel des distances sont calculées dans le graphe. Ces distances vont servir pour la répartition des différents sommets en sous-graphes. Prenons comme exemple l'algorithme de k-dominance vu en 3.1.4, et appliquons-le à un graphe $G = (V, E)$. Dans ce cas, les sommets de référence sont les éléments de l'ensemble k-dominant D. La répartition d'un sommet $v \in V$ dans le sous-graphe approprié se fait selon le sommet de D le plus proche. L'algorithme que nous proposons procède avec les deux mêmes étapes, mais ajustées à nos besoins. La sélection des sommets de référence se fera en évaluant leur

connectivité dans le graphe. La répartition des sommets en sous-graphes sera conçue de manière à ce que la partition obtenue respecte à la fois notre définition d'une partition valide et nos critères d'ingénierie inverse.

4.1.1 Sélection des sommets de référence.

Les sommets de référence sont à la base des calculs de distances, donc de la répartition des sommets. Pour obtenir une partition acceptable, l'ensemble des sommets de référence doit être choisi avec soin. Dans notre cas, les sommets appropriés sont centraux aux sous-systèmes du logiciel représenté par le graphe. Nous préconisons l'emploi de la valuation force de cohésion évoquée en 3.3.2 pour identifier ces sommets centraux. Elle a été utilisée avec succès pour identifier les arêtes externes d'une partition du graphe mais permet aussi de trouver les arêtes internes. Les sommets de référence seront donc les sommets incidents aux arêtes identifiées comme « centrales » à des sous-graphes par la valuation force de cohésion. Pour que le choix des sommets de référence soit paramétrable, l'utilisateur déterminera le niveau à partir duquel une arête est jugée suffisamment centrale pour que ses sommets incidents soient sommets de référence. Considérons un graphe $G = (V, E)$ pondéré par la valuation de force de cohésion $\Sigma : E \rightarrow \Re$. Les arêtes sélectionnées sont dans l'intervalle :

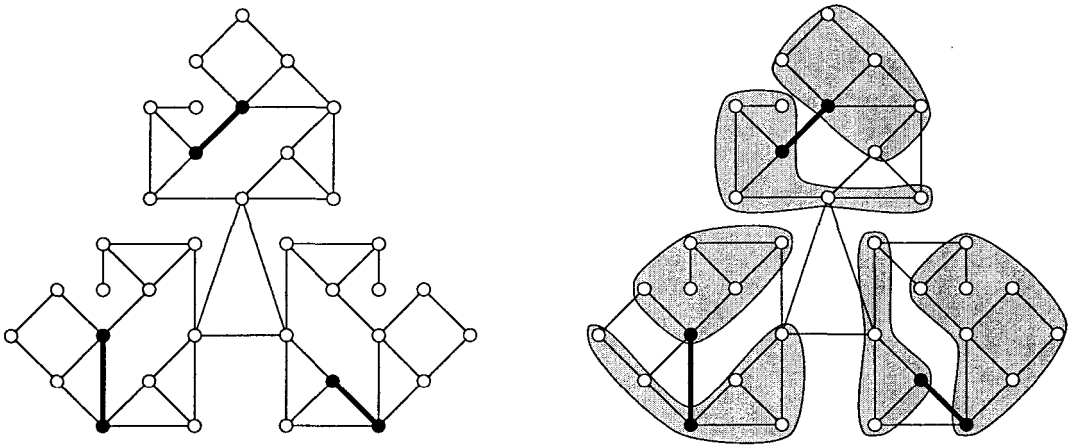
$$\left[f, \max_{e \in E} \Sigma(e) \right] \tag{4.1}$$

où f est le seuil minimal d'acceptation d'une arête tel que :

$$\min_{e \in E} \Sigma(e) \leq f \leq \max_{e \in E} \Sigma(e) \tag{4.2}$$

L'utilisation des arêtes cause un problème relativement à notre objectif d'identifier des sous-systèmes : il est possible d'avoir deux sommets de référence adjacents en plein milieu d'un sous-système potentiel. Couplé à une mauvaise répartition, cela revient à scinder le sous-système en deux sous-graphes distincts. Considérons le graphe $G = (V, E)$ de la figure 4.1(a) où trois arêtes ont été sélectionnées (les arêtes en gras), pour un total de six sommets de

référence (les sommets pleins de l'image). Considérons aussi une partition P de G dont chaque sous-graphe contient un sommet de référence v_{ref} et l'ensemble des sommets de V qui sont plus près de v_{ref} que de tout autre sommet de référence, les sommets équidistants à plusieurs sommets de référence étant répartis arbitraitement. Une partition possible de G est illustrée à la figure 4.1(b). Chacun des sous-systèmes identifiables à la figure 4.1(a) est divisé en deux dans P. La répartition des sommets doit éviter cette division.



(a) Un graphe G.

(b) Une partition P de G.

FIG. 4.1 – Exemple de mauvaise partition.

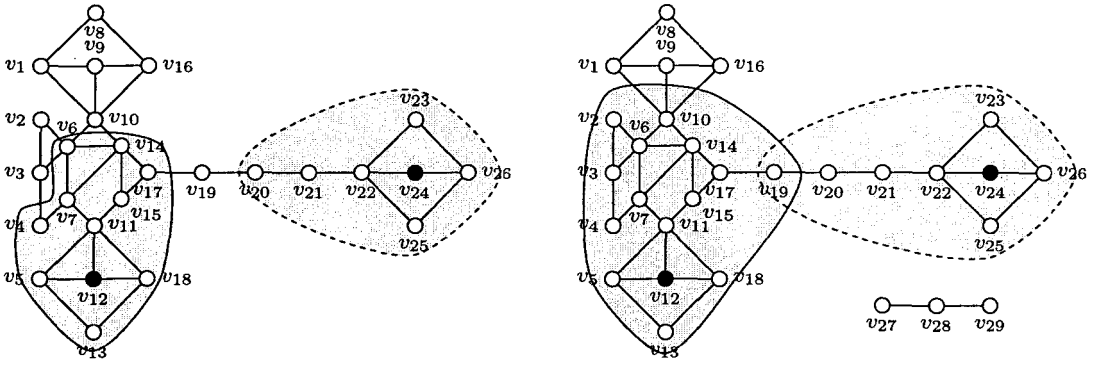
4.1.2 Répartition des sommets en sous-graphes.

La répartition des sommets de l'algorithme de fragmentation ne doit pas uniquement générer une partition valide (qui respecte la définition vue en 1.1.3), mais doit en produire une qui respecte nos idées de ce qu'est une bonne partition par rapport à l'ingénierie inverse : forte connectivité interne, faible connectivité externe, sous-graphes de tailles variées, un sous-graphe par sous-système. De plus, la répartition doit être influencée par un paramètre. Un paramètre qui revient souvent dans les algorithmes du chapitre 3 est la distance. Considérons un graphe $G = (V, E)$, un paramètre $1 \leq k \leq D(G)$ et un ensemble de sommets de référence $V_{ref} = \{v_1, v_2, .., v_n\} \subseteq V$. Si le paramètre k exprime la distance, chaque sommet du graphe

G appartient à l'un des cas suivants :

1. La distance entre un sommet v et un seul sommet de référence est au plus k ($v \in N_k[v_i]$ et $v \notin N_k[v_j]$ pour tout j différent de i).
2. La distance entre un sommet v et plusieurs sommet de référence est au plus k ($v \in N_k[v_i]$ et $v \in N_k[v_j]$ pour j différent de i).
3. La distance entre un sommet v et les sommets de référence est supérieure à k ($v \notin N_k[V_ref]$). De plus :
   (a) il existe un sommet de référence particulier qui est plus proche de v que les autres sommets de référence ($d(v, v_i) < d(v, v_j)$ pour tout j différent de i).
   (b) il existe quelques sommets de référence équidistants à v et plus proches que les autres sommets de référence ($d(v, v_i) = d(v, v_j)$ pour j différent de i).
   (c) le sommet v n'est pas connexe à un sommet de référence ($d(v, V_{ref}) = \infty$).

Ces cas sont illustrés à l'aide de la figure 4.2 et du tableau 4.1. Les sommets de V_{ref} sont ceux pleins dans les illustrations.



(a) $N_3[V_{ref}]$. (b) $N_4[V_{ref}]$.

FIG. 4.2 – Répartition des sommets pour $V_{ref} = \{v_{12}, v_{24}\}$.

Figure 4.2(a).	
Cas	Sommets
$v_i \in N_3[v_{12}]$ et $v_i \notin N_3[V_{24}])$	$v_4, v_5, v_6, v_7, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{17}, v_{18}$
$v_i \in N_3[v_{24}]$ et $v_i \notin N_3[V_{12}])$	$v_{20}, v_{21}, v_{22}, v_{23}, v_{24}, v_{25}, v_{26}$
$v_i \in N_3[v_{12}] \cap N_3[V_{24}]$	$\emptyset$
$3 < d(v_i, v_{12}) < d(v_i, v_{24})$	$v_1, v_2, v_3, v_8, v_9, v_{10}, v_{16}$
$3 < d(v_i, v_{24}) < d(v_i, v_{12})$	$\emptyset$
$3 < d(v_i, v_{24}) = d(v_i, v_{12})$	v_{19}
$d(v_i, V_{ref}) = \infty$	$\emptyset$
Figure 4.2(b).	
Cas	Sommets
$v_i \in N_4[v_{12}]$ et $v_i \notin N_4[V_{24}])$	$v_2, v_3, v_4, v_5, v_6, v_7, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{17}, v_{18}$
$v_i \in N_4[v_{24}]$ et $v_i \notin N_4[V_{12}])$	$v_{20}, v_{21}, v_{22}, v_{23}, v_{24}, v_{25}, v_{26}$
$v_i \in N_4[v_{12}] \cap N_4[V_{24}]$	v_{19}
$4 < d(v_i, v_{12}) < d(v_i, v_{24})$	v_1, v_8, v_9, v_{16}
$4 < d(v_i, v_{24}) < d(v_i, v_{12})$	$\emptyset$
$4 < d(v_i, v_{24}) = d(v_i, v_{12})$	$\emptyset$
$d(v_i, v_{ref}) = \infty$	v_{27}, v_{28}, v_{29}

TAB. 4.1 – Répartition des sommets des figures 4.2(a) et 4.2(b).

Supposons une nouvelle définition du diamètre. Soit H l'ensemble des composantes connexes d'un graphe G. On définit le diamètre maximal D_C comme :

$$D_C(G) = \max_{C \in H} D(C) \tag{4.3}$$

Le paramètre k est alors borné comme suit :

$$1 \leq k \leq D_C G \tag{4.4}$$

Il a été évoqué plus tôt que les sommets de référence viennent par paires. Les sommets d'une paire étant incidents à la même arête, ils appartiennent nécessairement au cas des sommets distants à une distance inférieure ou égale à k de plusieurs sommets de référence.

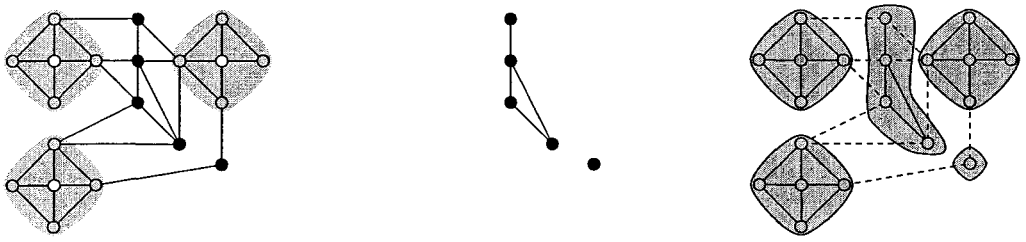
Pour obtenir une partition valide, il faut que l'algorithme de répartition identifie auquel des cas évoqués plus haut appartient chacun des sommets du graphe pour associer ce sommet à l'un des sous-graphes de la partition. Nous procéderons ainsi : un noyau est formé autour de chacun des sommets de référence d'un ensemble V_{ref}. Le noyau M_x est constitué de tous les sommets qui sont distants d'au plus k du sommet $x \in V_{ref}$ (le k-voisinage clos de x). Si un sommet v est attribuable selon cette règle à plus d'un noyau, c'est signe que ces noyaux sont relativement proches et nous procéderons par une fusion. Tous les noyaux qui contiennent v vont former un seul noyau plus gros. Remarquons que les noyaux qui ne sont pas produits par fusion ont, par définition, un rayon k et un diamètre d'au plus $2k$. La fusion permet toutefois de dépasser cette limite.

Les sommets à une distance de k ou plus de V_{ref} sont traités de la façon suivante. À chaque noyau M_x est associé un sous-graphe C_x composé des sommets de M_x ainsi que des sommets v de

$$V' = V - \bigcup_{x \in V_{ref}} M_x \tag{4.5}$$

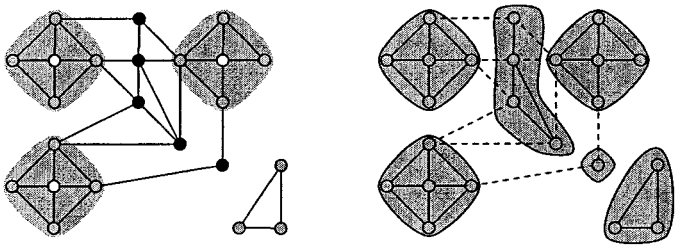
tels que $d(v, M_x) < d(v, M_y)$ pour tout $x \neq y$. Autrement dit, C_x est constitué des sommets de V' dont le noyau le plus proche est M_x. Un sommet v tel que plusieurs noyaux sont les plus

proches est qualifié de *conflictuel.* Chaque composante connexe de l'ensemble des sommets conflictuels formera un sous-graphe de la partition. Une composante connexe sans sommets de référence est transférée comme sous-graphe de la partition. Prenons pour exemple le graphe de la figure 4.3(a) qui présente plusieurs noyaux. Les seuls sommets qui ne sont pas dans un noyau (ceux illustrés par des points noirs) sont conflictuels. Le sous-graphe S des sommets conflictuels est illustré à la figure 4.3(b) et la partition qui en résulte à la figure 4.3(c). Les sommets conflictuels formant deux composantes connexes dans S, ils ont deux sous-graphes dans la partition. Prenons comme second exemple le graphe précédent auquel nous ajoutons une composante connexe, ce qui donne le graphe 4.4(a). La partition issue de ce graphe non-connexe, si l'on conserve les mêmes noyaux, est illustrée par la figure 4.4(b).



(a) Noyaux. (b) Sous-graphe S. (c) Partition.

FIG. 4.3 – Traitement des sommets conflictuels.



(a) Noyaux. (b) Partition.

FIG. 4.4 – Fragmentation d'un graphe non-connexe.

4.2 Implémentation de l'algorithme.

Voici de manière plus détaillée l'algorithme de fragmentation par noyaux. Nous allons considérer un graphe $G = (V, E)$ constitué d'un ensemble de sommet $V = \{v_1, v_2, .., v_n\}$. Chacune des fonctions qui constituent l'algorithme sera expliquée individuellement. L'algorithme de fragmentation (présenté par l'algorithme 4.1) prend en entrée un graphe $G = (V, E)$ qui sera pondéré selon la valuation force de cohésion $\Sigma : E \rightarrow \Re$ (d'autres valuations qui donnent des centres d'agrégats peuvent être utilisées). Cet algorithme tient compte de deux paramètres : le seuil minimal d'acceptation d'une arête f et le rayon k. L'algorithme commence par identifier les sommets isolés (sans incidences). Ces sommets servent à construire un sous-graphe C_{isol}. C'est ce que fait la fonction FiltrationIsolés(G,C_{isol}) (algorithme 4.2). On définit G', le graphe obtenu en retirant les sommets de C_{isol} du graphe G. Les sommets de C_{isol} n'ayant pas d'influence sur les résultats, ils sont regroupés et retirés afin d'augmenter les performances et d'assurer une meilleure lisibilité au graphe quotient. Par la suite, il y a création des noyaux de la collection M en deux opérations :

Identification des ensembles de référence On y fait la sélection des sommets de référence que l'on regroupe en ensembles de référence (ensembles de sommets de référence connexes).

Création Les sommets distants de k ou moins d'un ensemble de référence sont répartis dans le sous-graphe approprié. On fusionne les sous-graphes lorsque nécessaire.

Les autres sommets du graphe sont ensuite répartis selon qu'ils sont proches d'un noyau particulier, non-connexes à un noyau ou équidistants à plusieurs noyaux. Un graphe quotient G_Q est créé pour accompagner P.

Voici le détail de chacune des fonctions.

4.2.1 Identification des ensembles de référence.

La phase d'identification fait la sélection des sommets de référence qu'elle regroupe en ensembles de référence. Les ensembles de référence sont constitués de sommets de référence connexes. Étant connexes, ces sommets de référence vont être fusionnés lors de la création des

Algorithme 4.1 FragmentationNoyauxRayonK($G = (V, E), f, k, G_q = (V_q, E_q), P$)

$V_q \leftarrow \emptyset$
$E_q \leftarrow \emptyset$
$P \leftarrow \emptyset$
FiltrationIsolés(G, C_{isol})
$G' \leftarrow (V - C_{isol}, E)$
Identification($G', f, \boldsymbol{M}$)
CréationNoyaux($G', k, \boldsymbol{M}$)
RépartitionSommets($G', \boldsymbol{M}, P$)
$P \leftarrow P \cup \{C_{isol}\}$
CréationQuotient(G, P, G_q)

Algorithme 4.2 FiltrationIsolés($G = (V, E), C_{isol}$)

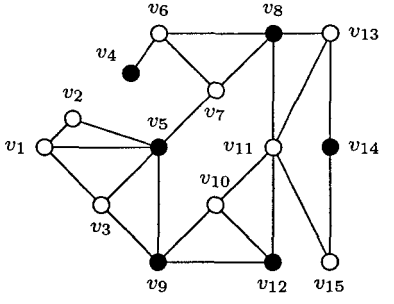
$C_{isol} \leftarrow \emptyset$
pour tout $v \in V$ **faire**
    si $|N(v)| = 0$ **alors**
        $C_{isol} \leftarrow C_{isol} \cup \{v\}$

noyaux. En effectuant cette fusion tout de suite, on diminue le nombre de noyaux lors des opérations subséquentes, ce qui a un impact bénéfique sur les performances de l'algorithme. L'algorithme 4.3 détaille la fonction d'identification « Identification(G,f,$\boldsymbol{M}$) ». Elle prend en paramètres une valeur f et un graphe G pour retourner une collection $\boldsymbol{M}$ d'ensembles de sommets de référence. Elle fonctionne comme suit. On applique tout d'abord la valuation. Le poids de chacune des arêtes e du graphe est comparé au seuil minimal d'acceptation f. Si le poids $\Sigma(e)$ est supérieur ou égal à f, les deux sommets incidents à e sont des sommets de référence et rejoignent l'ensemble V_{ref}. On construit ensuite les ensembles de référence. On définit G' comme étant le sous-graphe de G engendré par V_{ref}. Chaque ensemble de référence C, qui sera placé dans la collection $\boldsymbol{M}$, est constitué des sommets d'une composante connexe de G'. C'est ce que fait la deuxième boucle de l'algorithme. Elle emploie à répétition la fonction « ExtraireComposantConnexe » (décrite ci-dessous) pour aller chercher une à une les composantes connexes. Le tableau 4.5(b) donne un exemple de trace du fonctionnement de cette boucle pour un ensemble $V_{ref} = \{v_4, v_5, v_8, v_9, v_{12}, v_{14}\}$. La colonne « it » indique l'itération de la boucle et la colonne « l » le numéro de la ligne étudiée.

La fonction « ExtraireComposantConnexe(G, V', v, C) » procède comme suit : elle

Algorithme 4.3 Identification$(G = (V, E), f, \boldsymbol{M})$

$\gamma \leftarrow$ Force de cohésion.
$V_{ref} \leftarrow \emptyset$
pour tout $e = (u, v) \in E$ **faire**
    si $\gamma(e) \geq f$ **alors**
        $V_{ref} \leftarrow V_{ref} \cup \{u, v\}$
tant que $V_{ref} \neq \emptyset$ } **faire**
    $C \leftarrow \emptyset$
    ExtraireComposantConnexe$(G, V_{ref}, V_{ref}[1], C)$
    $\boldsymbol{M} \leftarrow \boldsymbol{M} \cup \{C\}$
    $V_{ref} \leftarrow V_{ref} - C$



(a) Graphe.

It	l	V_{ref}	C	$\boldsymbol{M}$
1	1	$\{v_4, v_5, v_8, v_9, v_{12}, v_{14}\}$	$\emptyset$	$\emptyset$
1	2	$\{v_4, v_5, v_8, v_9, v_{12}, v_{14}\}$	$\{v_4\}$	$\emptyset$
2	1	$\{v_5, v_8, v_9, v_{12}, v_{14}\}$	$\emptyset$	$\{\{v_4\}\}$
2	2	$\{v_5, v_8, v_9, v_{12}, v_{14}\}$	$\{v_5, v_9, v_{12}\}$	$\{\{v_4\}\}$
3	2	$\{v_8, v_{14}\}$	$\{v_8\}$	$\{\{v_4\}, \{v_5, v_9, v_{12}\}\}$
4	2	$\{v_{14}\}$	$\{v_{14}\}$	$\{\{v_4\}, \{v_5, v_9, v_{12}\}, \{v_8\}\}$
4	3	$\emptyset$	$\{v_{14}\}$	$\{\{v_4\}, \{v_5, v_9, v_{12}\}, \{v_8\}, \{v_{14}\}\}$

(b) Trace.

FIG. 4.5 – Trace de la seconde boucle de l'algorithme 4.3.

prend en paramètre un graphe $G = (V, E)$, un ensemble $V' \subseteq V$, un sommet v et retourne sa réponse, une composante connexe, dans l'ensemble C. L'identification d'une composante connexe est faite par un parcours en profondeur à partir de $v \in V$ du sous-graphe de G généré par les sommets V'. La figure 4.6 illustre un tel sous-graphe pour le graphe 4.5(a) avec $V' = \{v_4, v_5, v_8, v_9, v_{12}, v_{14}\}$. Les indices des sommets visités durant le parcours sont placés dans C. Si le sous-graphe n'est plus connexe, la visite se limite aux sommets connexes à v.

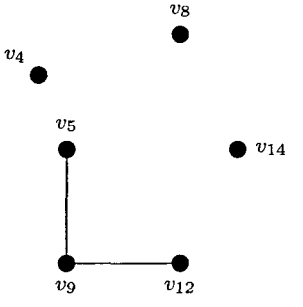


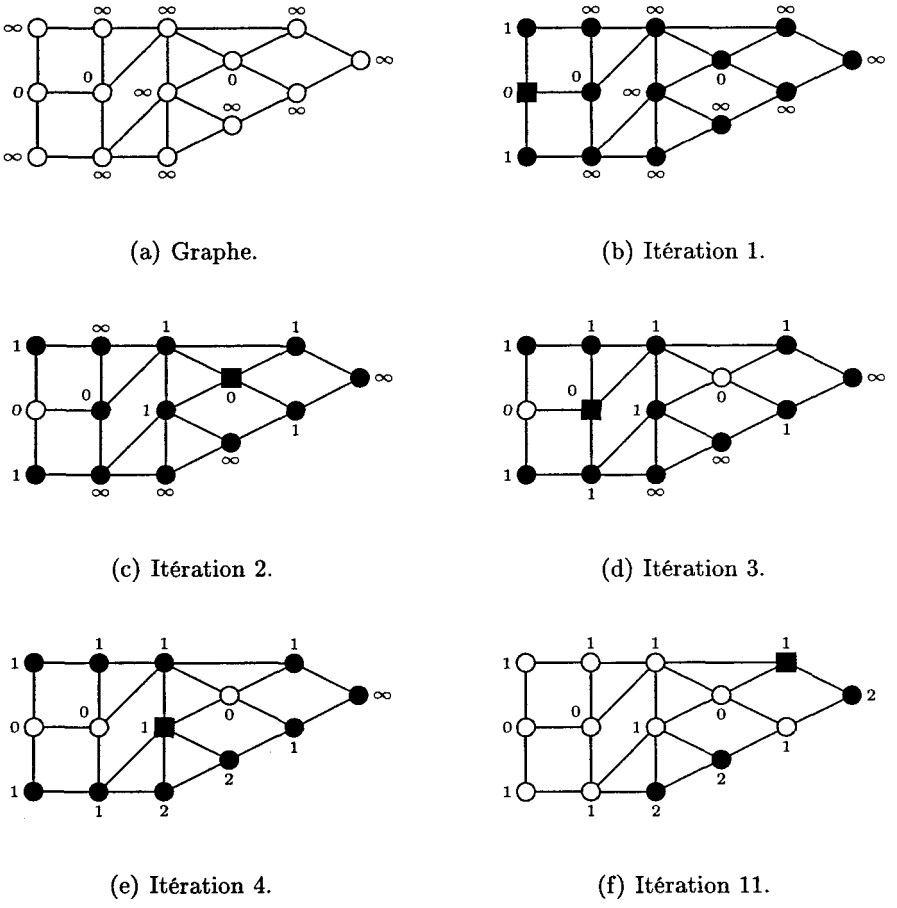
FIG. 4.6 – Sous-graphe du graphe 4.5(a) avec $V' = \{v_4, v_5, v_8, v_9, v_{12}, v_{14}\}$.

4.2.2 Création des noyaux.

La création des noyaux est la première partie de l'opération de répartition des sommets. Les ensembles de référence identifiés servent à générer des noyaux. Un sommet appartient à un noyau si sa distance avec un sommet de référence est d'au plus k. Un noyau M_j fusionne avec un noyau M_i (pour $1 \leq i < j$) si l'intersection des deux noyaux est non-vide.

La création des noyaux utilise le calcul des distances entre les ensembles de référence et les sommets du graphe. C'est ce que fait la fonction CalculerDistances($G = (V, E), V', W$), qui prend en paramètres un graphe $G = (V, E)$ et un sous-ensemble $V' \subseteq V$. Les résultats sont placés dans un tableau W tel que w_v de W contiendra $d(v, V')$ pour un sommet quelconque $v \in V$. La fonction utilise l'algorithme de Dijkstra (expliqué dans [17]). Si l'on remplace le sommet de départ de l'algorithme de Djikstra par l'ensemble de sommets V', l'algorithme va calculer pour chaque sommet $v \in V$ la distance $d(v, V')$. La figure 4.7 illustre cette application de Dijkstra sur un groupe de sommets. Les sommets non-visités sont pleins, le sommet courant

est carré et les sommets visités sont vides. Les valeurs sont indiquées pour chaque sommet.



(a) Graphe. (b) Itération 1.

(c) Itération 2. (d) Itération 3.

(e) Itération 4. (f) Itération 11.

FIG. 4.7 – Dijkstra sur un groupe de sommets.

La fonction « CréationNoyaux(G,k,M) », décrite par l'algorithme 4.4, prend en charge la création des noyaux eux-même. Cette fonction a pour paramètres un graphe $G = (V, E)$, un rayon k et aussi une collection M d'ensembles de référence ordonnés. Ainsi, M_i désigne le i-ème ensemble de la collection M. L'algorithme de création des noyaux transforme chacun des ensembles M_i en un ensemble de sommets qui est aussi un noyau. Pour arriver à ce résultat, la fonction emploie un dictionnaire D tel que d_v de D contient soit M_{d_v}, c'est-à-dire l'indice du noyau associé pour le moment au sommet v, soit 0 si v n'est pas encore associé à un noyau. Ce dictionnaire nous permet d'identifier les noyaux qui partagent des sommets pour

en faire la fusion. Il est complété par un ensemble des fusions qui contient les noyaux déjà créés à fusionner avec le noyau en cours de création. L'algorithme fonctionne comme suit : il calcule pour chaque sommet $v \in V$ la distance $d(v, M_1)$. Si $d(v, M_1) \leq k$, on donne à d_v la valeur 1. Par la suite, on itère pour $1 < j < |M|$. Chaque itération débute en réinitialisant l'ensemble de fusions F tel qu'il ne contient plus que l'indice j du noyau courant. Ensuite, on calcule $d(v, M_j)$ pour chaque sommet $v \in V$ à l'aide de la fonction « CalculerDistances ». Les distances sont conservées dans un tableau W tel que $w_v = d(v, M_j)$. Si w_v est inférieure ou égale à k, on considère la valeur de d_v. Il y a deux possibilités :

- La valeur d_v égale 0. Le sommet v n'est pas encore associé à un noyau. On associe v au noyau courant j en mettant d_v à j et on passe au sommet suivant.
- On a $1 \leq d_v \leq j - 1$. Le sommet v est associé à un noyau créé précédemment, ce qui indique une fusion entre M_j et M_{d_v}. La valeur d_v est ajoutée à F.

Lorsque tous les sommets ont été visités, on effectue les fusions de F. Chaque noyau M_i indicé dans F est fusionné à M_j. Le noyau M_j contient donc :

$$M_j \cup \left(\bigcup_{i \in F} M_i \right) \tag{4.6}$$

Le dictionnaire est ajusté pour tout sommet v impliqué dans la fusion de façon à ce que $d_v = j$. La fusion terminée, on passe à l'itération suivante.

4.2.3 Répartition des particules hors-noyaux.

Cette partie de l'algorithme continue la répartition des sommets. Précédemment, nous avons traité les cas suivants pour un sommet $v \in V$: le sommet v est un sommet de référence, il est dans un noyau ou il n'est pas connexe à un sommet de référence et n'a pas de voisins. Nous allons considérer les cas suivants pour un sommet v dont la distance est au moins $k + 1$ relativement à V_{ref} : le sommet v n'est pas connexe à un sommet de référence, il est plus proche d'un noyau particulier que de tous les autres noyaux, ou il est équidistant à plusieurs noyaux, qui sont aussi les noyaux les plus proches.

La fonction « RépartitionSommets(G,M,P) » (algorithme 4.5) prend en para-

Algorithme 4.4 CréationNoyaux($G = (V = \{v_1, .., v_n\}, E), k, \boldsymbol{M} = \{M_1, .., M_m\}$)

$D \leftarrow \{d_1, .., d_n\}$
pour tout $v \in V$ **faire**
    $d_v \leftarrow 0$
pour $j \leftarrow 1$ to m **faire**
    $F \leftarrow \{j\}$
    $W \leftarrow \{w_1, .., w_n\}$
    CalculerDistances(G, M_j, W)
    pour tout $v \in V$ **faire**
        si $w_v \leq k$ **alors**
            $M_j \leftarrow M_j \cup \{v\}$
    pour tout $v \in M_j$ **faire**
        si $d_v = 0$ **alors**
            $d_v \leftarrow j$
        sinon si $d_v \notin F$ **alors**
            $F \leftarrow F \cup \{d_v\}$
    $M_0 \leftarrow \emptyset$
    pour tout $f \in F$ **faire**
        $M_0 \leftarrow M_0 \cup M_f$
        $M_f \leftarrow \emptyset$
        pour tout $v \in M_f$ **faire**
            $d_v \leftarrow j$
    $M_j \leftarrow M_0$

Algorithme 4.5 RepartitionSommets($G = (V = \{v_1, .., v_n\}, E), \boldsymbol{M} = \{M_1, .., M_m\}, P$)

$V' \leftarrow V$
$P_c \leftarrow \emptyset$
$P_{nc} \leftarrow \emptyset$
pour $j \leftarrow 1$ to m **faire**
    $V' \leftarrow V' - M_j$
si $|V'| \neq 0$ **alors**
    $D \leftarrow \{\{d_1, \delta_1\}, .., \{d_n, \delta_n\}\}$
    RemplirDictionnaire($G, V', \boldsymbol{M}, D$)
    pour tout $v \in V'$ **faire**
        si $d_v \neq 0$ **alors**
            $M_{d_v} \leftarrow M_{d_v} \cup \{v\}$
        sinon si $\delta_v \neq \infty$ **alors**
            $P_{nc} \leftarrow P_{nc} \cup \{v\}$
        sinon
            $P_c \leftarrow P_c \cup \{v\}$
FairePartition($G, \boldsymbol{M}, P_c, P_{nc}, P$)

mètres un graphe $G = (V, E)$ et un ensemble $\boldsymbol{M}$ de noyaux de G. Elle retourne une partition P. L'algorithme est appliqué sur l'ensemble des sommets de V qui ne sont pas dans un noyau de $\boldsymbol{M}$, ou :

$$V' = V - \bigcup_{M_j \in M} M_j \tag{4.7}$$

Nous allons définir $M(v)$ pour un sommet $v \in V'$ comme étant l'ensemble des noyaux M_j de $\boldsymbol{M}$ tels que la distance $d(v, M_j)$ est minimale et différente de ∞. Le sommet v va être réparti dans l'un des ensembles suivants selon la valeur de $M(v)$:

- Le sommet v va dans le noyau M_j si $M_j \in M(v)$ et $|M(v)| = 1$.
- Le sommet v va dans l'ensemble de sommets conflictuels P_c si $|M(v)| > 1$.
- Le sommet v va dans l'ensemble de sommets non-connexes P_{nc} si $|M(v)| = 0$.

Pour ce faire, on a encore recours à un dictionnaire. Le dictionnaire D contient une paire de valeurs pour chaque sommet v : d_v, un indice de noyau et δ_v, une distance.

La fonction RemplirDictionnaire($G = (V, E), V', \boldsymbol{M}, D$) (algorithme 4.6) initialise et remplit le dictionnaire. Elle prend en paramètres G : un graphe ; V' : les sommets qui ne sont pas attribués à un noyau ; $\boldsymbol{M}$: une collection de noyaux de G. L'initialisation du dictionnaire se fait comme suit. Si un sommet v de V est dans V', alors le sommet n'est pas encore attribué à un noyau et les valeurs d_v et δ_v sont respectivement de 0 et de ∞. Si v n'est pas dans V', il est dans un des ensembles contenus dans $\boldsymbol{M}$. Cet ensemble M_j est identifié et les valeurs d_v et δ_v sont respectivement de j et de 0. L'initialisation terminée, on remplit le dictionnaire. Pour chacun des noyaux M_j de $\boldsymbol{M}$, on calcule les distances dans le graphe, que l'on place dans l'ensemble W tel que w_v contient $d(v, M_j)$. Ensuite, on compare w_v de chaque $v \in V'$ à la distance δ_v du dictionnaire. La valeur δ_v correspond à :

$$\delta_v = d(v, M_{d_v}) \tag{4.8}$$

pour $d_v \neq 0$, une valeur de d_v nulle indiquant un conflit ou une non-connectivité avec les noyaux M_g (pour $1 \leq g \leq J$). Si w_v est inférieure à δ_v, alors v appartient pour le moment

à M_j. On change δ_v à la distance w_v et d_v à la valeur j. Une égalité entre δ_v et w_v indique un sommet potentiellement conflictuel. La distance w_v est placée dans δ_v mais d_v devient nul pour indiquer le conflit.

Algorithme 4.6 RemplirDictionnaire($G, V', M = \{M_1, .., M_m\}, D = \{\{d_1, \delta_1\}, .., \{d_n, \delta_n\}\}$)

```
pour tout v ∈ V faire
    si v ∉ V' alors
        j ← 1
        tant que v ∉ M_j } faire
            j ← j + 1
        d_v ← j
        δ_v ← 0
    sinon
        d_v ← 0
        δ_v ← ∞
pour j ← 1 to m faire
    si M_j ≠ ∅ alors
        W ← {w_1, .., w_n}
        CalculerDistances(G, M_j, W)
        pour tout v ∈ V' faire
            si δ_v > w_v alors
                δ_v ← w_v
                d_v ← j
            sinon si δ_v = w_v et d_v ≠ 0 alors
                δ_v ← w_v
                d_v ← 0
```

L'algorithme 4.5 utilise les données du dictionnaire pour identifier à quel ensemble appartient chaque sommet de V'. Il y a quatre cas possibles, qui sont résumés dans le tableau 4.2.

Valeurs		Cas
d	δ	
j	0	Appartient au noyau M_j
j	$1 \leq d < \infty$	Le noyau M_j, à distance d, est le plus près.
0	$1 \leq d < \infty$	Sommet conflictuel (à distance d de plusieurs noyaux.)
0	∞	Ce sommet n'est pas connexe à un noyau.

TAB. 4.2 – Signification des valeurs du dictionnaire.

Lorsque tous les sommets sont attribués à un ensemble M_j, à l'ensemble P_c ou à

l'ensemble P_{nc}, la fonction « FairePartition($G, \boldsymbol{M}, P_c, P_{nc}, P$) » transforme les ensembles d'indices en sous-graphes de G. L'algorithme 4.7 décrit cette fonction. Chacun des ensembles M_j de $\boldsymbol{M}$ devient un sous-graphe $C = \{V_c, E_c\}$ de P. Les arêtes communes à deux sommets de V_c sont placées dans E_c. C'est ce que fait la première des trois boucles de FairePartition. La deuxième boucle convertit chacune des composantes connexes du sous-graphe de G généré par les sommets indicés dans P_c en sous-graphe de P. La troisième boucle répète cette dernière procédure pour les sommets de P_{nc}. Les deuxième et troisième boucles extraient les composantes connexes avec la procédure expliquée lors de l'identification des ensembles de référence (voir 4.2.1).

4.2.4 Création du graphe quotient.

La fonction « CréationQuotient(G, P, G_q) » construit le graphe quotient de la partition P du graphe G. Elle produit un sommet $v_i \in V_q$ pour chaque sous-graphe C_i de la partition $P = \{C_1, C_2, .., C_n\}$. Ensuite, elle place une arête $g = \{v_i, v_j\}$ (pour $i \neq j$) dans E_q s'il existe dans E au moins une arête $e = \{v_x, v_y\}$ telle que v_x est dans C_i et v_y dans C_j.

4.3 Exemple d'application de l'algorithme.

Considérons un graphe $G = (V, E)$ tel qu'illustré à la figure 4.8. Ce graphe est pondéré par une valuation quelconque $\Sigma : E \rightarrow \Re$. Nous n'utilisons pas la valuation force de cohésion de manière à conserver un meilleur contrôle sur les résultats présentés par l'exemple, mais les principes de fonctionnement de l'algorithme restent les mêmes. Supposons une valuation et un paramètre f tels que les arêtes en gras de la figure 4.8 sont dans l'intervalle :

$$\left[f, \max_{e \in E} \Sigma(e) \right] \tag{4.9}$$

et les sommets incidents à ces arêtes sont les sommets de référence. De plus, on pose $k = 2$.

La première opération appliquée par l'algorithme consiste à filtrer les sommets isolés. Le graphe G est divisé en deux sous-graphes : G' qui va servir pour les opérations subséquentes et C_{isol} qui contient les sommets isolés. Ensuite, on identifie les ensembles de

Algorithme 4.7 FairePartition($G = (V, E), \boldsymbol{M} = \{M_1, .., M_m\}, P_c, P_{nc}, P$)

$C_0 = (V_0, E_0)$
pour $j \leftarrow 1$ to m **faire**
    si $|M_j| \neq 0$ **alors**
        $V_0 \leftarrow \emptyset$
        $E_0 \leftarrow \emptyset$
        pour tout $v \in M_j$ **faire**
            $V_0 \leftarrow V_0 \cup \{v\}$
        pour tout $e = \{u, v\} \in E$ **faire**
            si $u \in V_0$ et $v \in V_0$ **alors**
                $E_0 \leftarrow E_0 \cup \{e\}$
        $P \leftarrow P \cup \{C_0\}$
tant que $P_c \neq \emptyset$ } **faire**
    $C \leftarrow \emptyset$
    $V_0 \leftarrow \emptyset$
    $E_0 \leftarrow \emptyset$
    ExtraireComposantConnexe($G, P_c, P_c[1], C$)
    pour tout $v \in C$ **faire**
        $V_0 \leftarrow V_0 \cup \{v\}$
    pour tout $e = \{u, v\} \in E$ **faire**
        si $u \in V_0$ et $v \in V_o$ **alors**
            $E_0 \leftarrow E_0 \cup \{e\}$
    $P \leftarrow P \cup \{C_0\}$
    $P_c \leftarrow P_c - C$
tant que $P_{nc} \neq \emptyset$ } **faire**
    $C \leftarrow \emptyset$
    $V_0 \leftarrow \emptyset$
    $E_0 \leftarrow \emptyset$
    ExtraireComposantConnexe($G, P_{nc}, P_{nc}[1], C$)
    pour tout $v \in C$ **faire**
        $V_0 \leftarrow V_0 \cup \{v\}$
    pour tout $e = \{u, v\} \in E$ **faire**
        si $u \in V_0$ et $v \in V_o$ **alors**
            $E_0 \leftarrow E_0 \cup \{e\}$
    $P \leftarrow P \cup \{C_0\}$
    $P_{nc} \leftarrow P_{nc} - C$

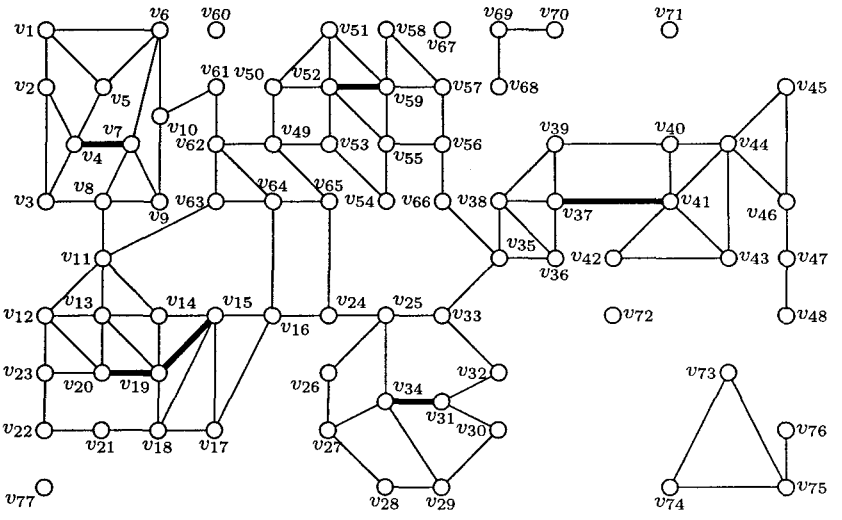


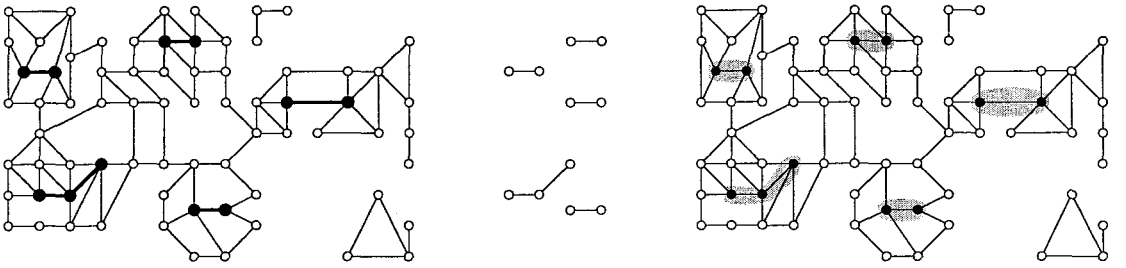
FIG. 4.8 – Un graphe.

référence. Les sommets de référence (incidents aux arêtes dans l'intervalle) sont identifiés. Ce sont les sommets pleins de la figure 4.9(a) :

$$V_{ref} = \{v_4, v_7, v_{15}, v_{19}, v_{20}, v_{31}, v_{34}, v_{37}, v_{41}, v_{52}, v_{59}\} \tag{4.10}$$

Le sous-graphe G'' généré à partir de G' par V_{ref} est illustré à la figure 4.9(b). On extrait les composantes connexes de G''. Chaque composante connexe forme un ensemble de référence M_i de $\boldsymbol{M}$ pour le calcul des noyaux. Ici, on a $M_1 = \{v_4, v_7\}$, $M_2 = \{v_{15}, v_{19}, v_{20}\}$, $M_3 = \{v_{31}, v_{34}\}$, $M_4 = \{v_{37}, v_{41}\}$ et $M_5 = \{v_{52}, v_{59}\}$. Ils sont illustrés à la figure 4.9(c).

La création des noyaux transforme les ensembles de sommets M_i de $\boldsymbol{M}$ en noyaux. Commençons par M_1. On mesure les distances dans le graphe. Tous les sommet v dont la distance avec M_1 est de 2 ou moins est dans le nouveau noyau. Le noyau M_1 est illustré à la figure 4.10(a). Après, on itère pour $2 \leq j \leq 5$. L'ensemble M_j sera converti en noyau. Le nouveau noyau va contenir le voisinage $N_k[M_J]$ plus tout noyau M_i déjà calculé tel que : $M_j \cap M_i \neq \emptyset$ pour $0 < i < j$. Donc, le noyau M_2 va contenir le voisinage $N_2[M_2]$ illustré à la figure 4.10(b). Comme il existe un sommet de M_1 qui est aussi dans $N_2[M_2]$, il y a fusion. Le



(a) Sommets de référence. (b) G''. (c) M.

FIG. 4.9 – Identification des ensembles de référence.

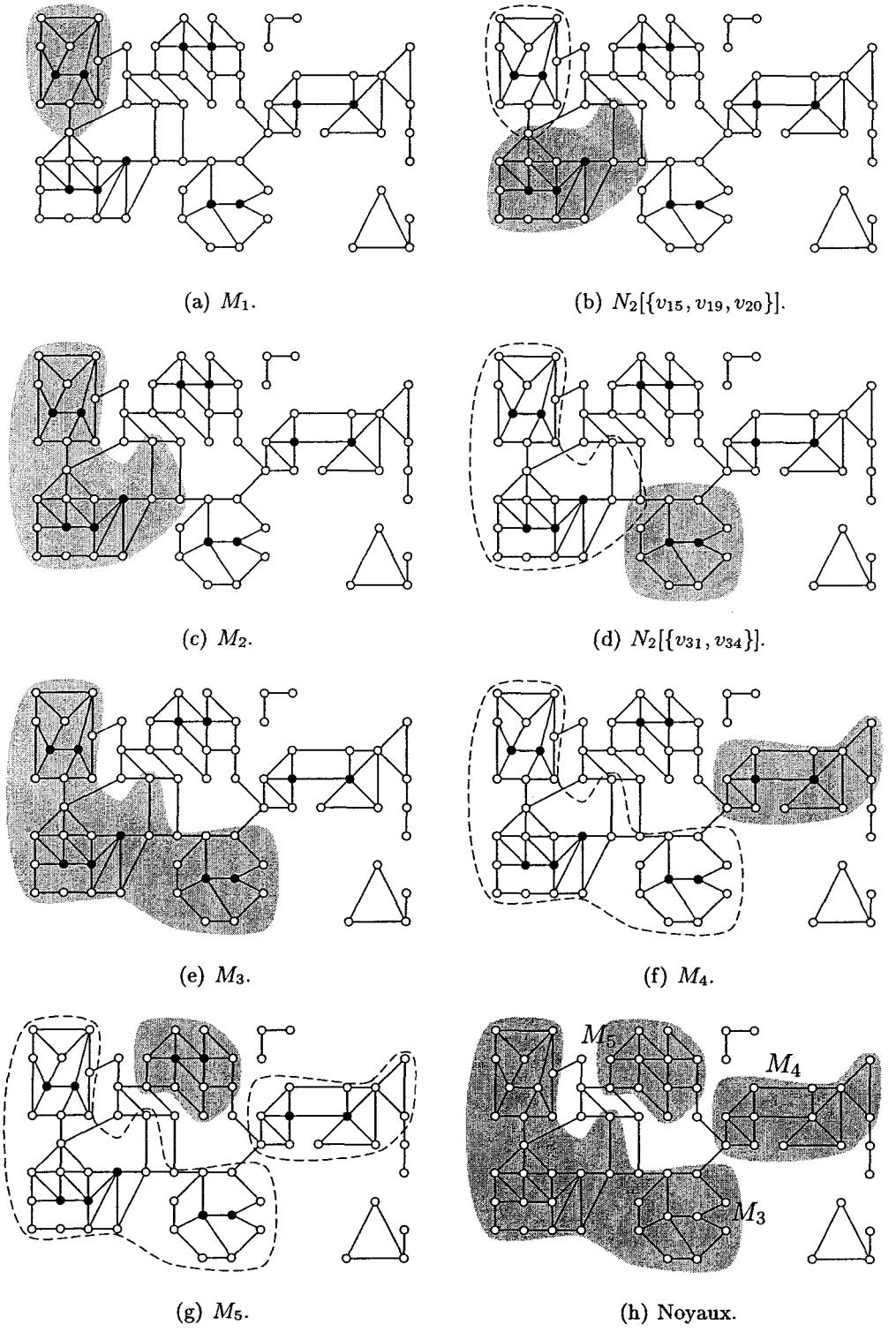
Noyau	Contenu
M_1	$\emptyset$
M_2	$\emptyset$
M_3	$v_{64}, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}$, $v_{13}, v_{14}, v_{15}, v_{16}, v_{17}, v_{18}, v_{19}, v_{20}, v_{21}, v_{22}, v_{23}$, $v_{24}, v_{25}, v_{26}, v_{27}, v_{28}, v_{29}, v_{30}, v_{31}, v_{32}, v_{33}, v_{34}$
M_4	$v_{35}, v_{36}, v_{37}, v_{38}, v_{39}, v_{40}, v_{41}, v_{42}, v_{43}, v_{44}, v_{45}, v_{46}$
M_5	$v_{49}, v_{50}, v_{51}, v_{52}, v_{53}, v_{54}, v_{55}, v_{56}, v_{57}, v_{58}, v_{59}$

TAB. 4.3 – Résultats de la création des noyaux.

voisinage et le noyau M_1 sont assemblés pour former le noyau M_2 (figure 4.10(c)). Après la fusion : $M_1 = \emptyset$. Le voisinage $N_2[M_3]$ illustré par la figure 4.10(d) a un sommet commun avec M_2, d'où un nouveau recours à la fusion. Le noyau M_3 obtenu est représenté à la figure 4.10(d). Les deux ensembles M_4 et M_5 restants sont transformés sans que la fusion soit nécessaire, et produisent respectivement les noyaux des figures 4.10(f) et 4.10(g). Le tableau 4.3 résume les noyaux obtenus, qui sont illustrés tous ensemble dans la figure 4.10(h).

On constate à la figure 4.10(h) qu'il reste des sommets qui ne sont pas attribués à l'un ou à l'autre des noyaux. On procède à l'étape de répartition. La fonction utilise un dictionnaire pour déterminer auquel des ensembles d'indices suivants un sommet v appartient :

- Le noyau indiqué par d_i
- L'ensemble P_c
- L'ensemble P_{nc}



(a) M_1.

(b) $N_2[\{v_{15}, v_{19}, v_{20}\}]$.

(c) M_2.

(d) $N_2[\{v_{31}, v_{34}\}]$.

(e) M_3.

(f) M_4.

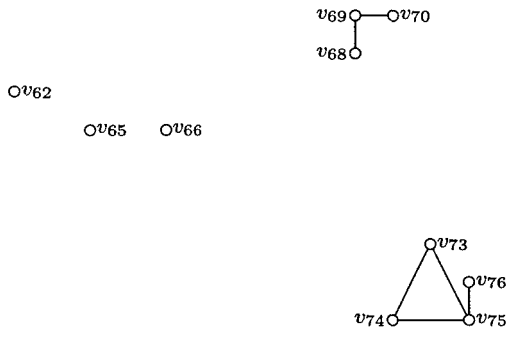
(g) M_5.

(h) Noyaux.

FIG. 4.10 – Exemple de création des noyaux.

selon les règles exposées au tableau 4.2) pour chacun des sommets v de G'.

Une fois les sommets répartis dans les différents ensembles, il faut transformer ces ensembles en sous-graphes de la partition. La fonction « FairePartition » produit une partition P de G' à partir de la collection M et des ensembles P_c et P_{nc}. Chaque ensemble $M_i \in M$ est transformé en un sous-graphe $C_i = \{V_i, E_i\}$. Ensuite, les sommets de P_c sont utilisés pour générer le sous-graphe G_c de G' (figure 4.11(a)). Chaque composante connexe de G_c est un sous-graphe de P. Lorsque toutes les composantes connexes de G_c sont dans P, on recommence pour P_{nc}. On crée un sous-graphe G_{nc} de G' (figure 4.11(b)) dont chaque composante connexe va devenir un sous-graphe de P.



(a) G_c. (b) G_{nc}.

FIG. 4.11 – Sous-graphes générés par P_c et P_{nc}.

On a maintenant une partition P de G'. Pour que P soit une partition valide de G, il faut lui ajouter les sommets isolés que l'on a filtré précédemment. La partition finale est illustrée à la figure 4.12. Les arêtes externes sont des pointillés.

Regardons les effets des paramètres sur le résultat. La figure 4.13(a) illustre la partition obtenue avec l'algorithme pour $k = 1$ et pour les même sommets de référence. Une partition engendrée avec une valeur de f plus élevée (intervalle plus petit) et avec $k = 2$ est dessinée à la figure 4.13(b). Les sommets de référence sont pleins, les arêtes dans l'intervalle plus larges et les arêtes externes pointillées. Les différents sous-graphes de ces deux partitions sont coloriés à l'aide de deux couleurs, pour la lisibilité.

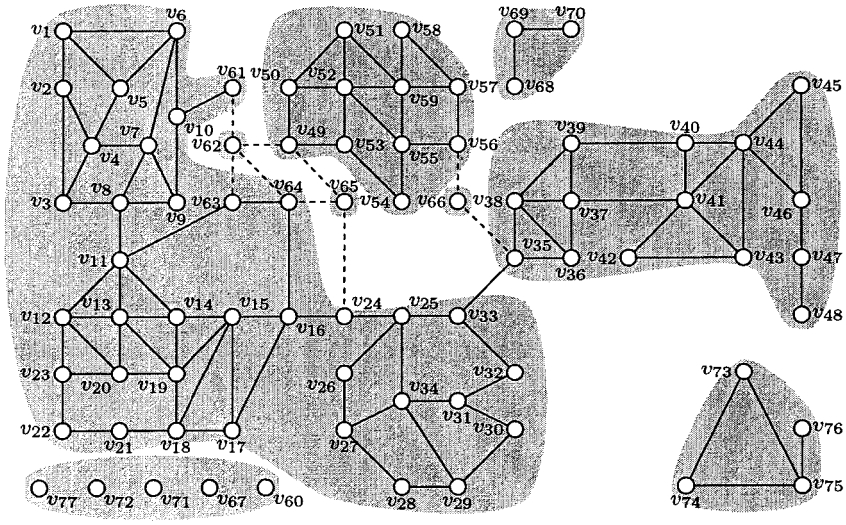
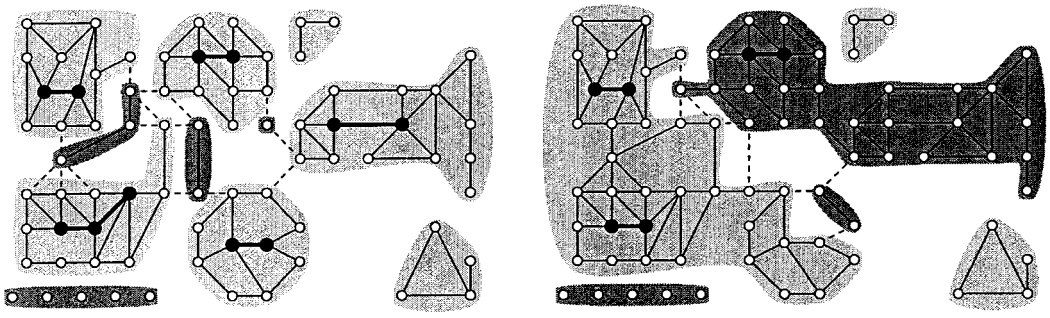


FIG. 4.12 – La partition P.



(a) $k = 1$.

(b) f différent.

FIG. 4.13 – Partitions produites par des paramètres différents.

4.4 Conclusion.

L'algorithme de fragmentation par noyaux semble contourner les faiblesses des algorithmes du chapitre 3 relativement à notre problématique d'ingénierie inverse. La fusion des noyaux et la répartition des sommets en dehors des rayons dans les sous-graphes permettent d'obtenir des partitions avec des tailles de sous-graphes variées. Deux paramètres distincts, le rayon et le seuil minimal d'acceptation, assistent l'utilisateur dans le processus de recherche d'une partition optimale selon des critères de génie logiciel. Cependant, cet algorithme n'a pas encore fait ses preuves dans un environnement logiciel. C'est à la mise-en-oeuvre de l'algorithme que se consacrera le prochain chapitre. Nous y verrons son interface-utilisateur, ses résultats et, dépendamment de ceux-ci, les améliorations possibles à notre algorithme.

CHAPITRE 5

MISE EN OEUVRE.

Nous traiterons maintenant de la mise en oeuvre de l'algorithme de fragmentation par noyaux introduit au chapitre 4. La première partie du chapitre portera sur l'interface utilisateur. Pour que l'utilisateur de l'algorithme soit en mesure d'en exploiter toute la puissance, il faut considérer différents critères. Non seulement nous allons voir ces critères, mais nous étudierons l'interface choisie pour les appliquer. Deuxièmement, nous utiliserons l'algorithme dans le contexte de la problématique, l'ingénierie inverse du logiciel, pour observer ce dont il est capable. À partir de graphes issus de systèmes logiciels variés, nous allons générer des partitions que nous analyserons par la suite. Nous terminerons par une discussion sur nos objectifs, afin de s'assurer qu'ils sont atteints, ainsi que sur les améliorations possibles.

5.1 L'interface utilisateur.

Avant de discuter de l'interface de l'algorithme de fragmentation par noyaux, penchons nous sur les critères d'une interface utilisateur ainsi que sur le système logiciel avec lequel nous devons travailler.

5.1.1 Quelques critères d'une bonne interface.

La création d'une interface utilisateur n'est pas une science exacte. Une multitude de critères, souvent contradictoires, s'applique à chaque cas. Prenons par exemple le choix des boutons dans un dialogue. Des critères d'esthétisme et de cohérence ([27]) préconisent des boutons de formes similaires et alignés alors que des critères plus fonctionnels d'apprentissage et de discrimination, évoqués dans [22], suggèrent des boutons très variés au placement

irrégulier. Lors de la création d'une interface utilisateur, il faut sélectionner les critères à appliquer de manière intelligente et trouver le juste équilibre dans leur usage. Les prochaines lignes décrivent les principaux critères considérés, issus de sources diverses [27, 22, 65, 69, 71].

Manipulation directe (« Direct Manipulation ») : L'utilisateur manipule directement une version informatique de son objet d'intérêt (ici, le graphe). Les changements engendrés par les manipulations sont faits rapidement et illustrés en temps réel. Nous disposons d'un outil de manipulation directe de graphes, le logiciel « FW3D-Graphes », dans lequel sera implémenté l'algorithme de fragmentation par noyaux.

Apprentissage facile : L'interface utilisateur est conçue soit d'une manière qui favorise un apprentissage rapide, soit d'une manière qui n'en nécessite quasiment aucun. Les indicateurs visuels de l'interface ont une signification claire, liée au domaine d'emploi. Leur usage est évident. Afin de faciliter la mémorisation, ils sont aisément distinguables. Faciliter l'apprentissage de la fragmentation par noyaux passe par une interface claire, à la terminologie propre à la théorie des graphes et à l'ingénierie inverse.

Cohérence : Les objets de l'interface utilisateur se comportent de manières similaires partout. L'apparence de l'interface respecte un certain standard dans l'ensemble du système logiciel. Des dialogues aux fonctionnalités similaires se ressemblent, afin que l'utilisateur passe facilement de l'un à l'autre. La cohérence s'applique aussi au fonctionnement. L'utilisateur s'attend à ce que deux dialogues d'un même logiciel se comportent de la même façon. Des contradictions augmentent le nombre d'erreurs commises. Pour éviter cela, l'interface de la fragmentation par noyaux sera construite en prenant exemple sur les dialogues déjà présents dans FW3D-Graphes.

Récupération des erreurs : L'erreur est potentiellement présente dans toute action de l'utilisateur. Une bonne interface permet à l'utilisateur de revenir en arrière pour éliminer les résultats de ses actions. L'effacement des sous-graphes des mauvaises partitions étant déjà prévu dans FW3D-Graphes, ce critère est respecté.

Interfaces étendues : Les dialogues sont traités avec la même importance que l'objet d'inté-

rêt (le graphe). Ils sont manipulables, enregistrables et non-bloquants. Les valeurs d'un dialogue peuvent varier indépendamment de l'objet d'intérêt. Dans le cas évoqué ici, seul l'aspect dialogue non-bloquant est conservé, les valeurs employées étant par nécessité liées au graphe affiché.

Complétude : Toutes les fonctionnalités sont implémentées à même l'interface utilisée. Pour l'algorithme de fragmentation par noyaux, la complétude viendra par le contrôle de la valuation force de cohésion et des deux paramètres de l'algorithme. Quelques options seront toutefois ajoutées pour supporter des besoins futurs ou plus spécifiques.

Esthétisme : Un bel esthétisme stimule le désir qu'a l'utilisateur d'employer l'interface. C'est un élément important dans la promotion d'une technologie informatique.

Le juste dosage, parmi ces critères, permettra d'obtenir une interface utilisateur plaisante.

5.1.2 L'environnement FW3D-Graphes.

Le logiciel FW3D-Graphes affiche un graphe en $3D$ à l'aide d'un algorithme de placement par un système masses-ressorts (voir 1.2.3). Il contient aussi divers algorithmes de visualisation et de fragmentation de graphes. Cet outil logiciel a été développé par Yves Chiricota dans le cadre de ses recherches et une version de l'exécutable devrait être disponible en 2006. C'est dans ce logiciel qu'a été implémenté l'algorithme de fragmentation par noyaux.

La fenêtre de l'application est illustrée à la figure 5.1. Les numéros en gras réfèrent à cette figure. La fenêtre de FW3D-Graphes est divisée en deux. La partie de gauche (**1**) contient l'arborescence des graphes actuellement ouverts. On en compte sept sur l'illustration. L'arborescence est hiérarchisée pour qu'un graphe G' soit l'enfant d'un graphe G si G' est lié par une opération à G, ou si G' dépend de G. Une opération est un traitement quelconque, implémenté dans le logiciel, qui s'applique sur un graphe d'entrée et produit un ou plusieurs graphes de sortie. Dans le cas d'une partition P d'un graphe G, l'arborescence est construite comme suit : Le graphe de départ G a pour enfant le graphe quotient de P. Les sous-graphes de P sont tous des enfants du graphe quotient. Chaque graphe contenu dans l'arborescence

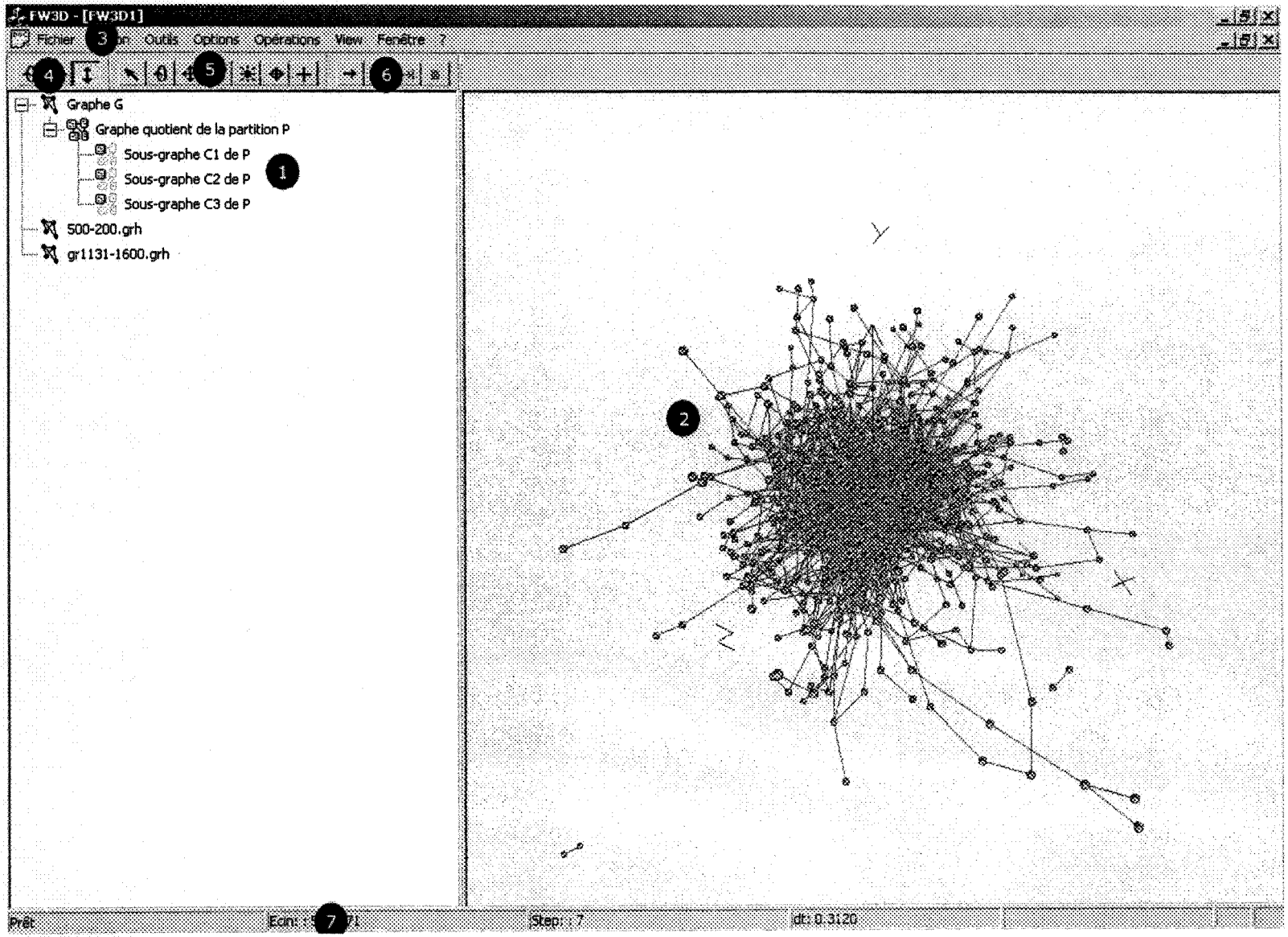


FIG. 5.1 – Éléments de la fenêtre de FW3D-Graphes.

est affiché sous la forme d'un nom précédé d'un icône (voir le tableau 5.1). Le graphe courant s'affiche à droite de la fenêtre de l'application (**2**). Les options d'affichage et de simulation, la gestion des graphes et les différentes opérations de l'application sont contenues dans le menu principal (**3**). La barre d'outils est constituée de trois palettes. La première palette (**4**) contient les outils d'affichages. Ces outils permettent des rotations, des translations et des homothéties de la vue 3D du graphe (**2**) à l'aide de la souris. Les opérations de base sur le graphe et ses noeuds sont dans la deuxième barre d'outils (**5**). La troisième barre d'outils (**6**) contrôle la simulation de l'algorithme de placement. Une barre des tâches (**7**) affiche des informations sur l'état de la simulation et sur le dernier noeud sélectionné.

Icône	Signification
	Graphe.
	Sous-graphe partiel.
	Graphe quotient.
	Sous-graphe d'une partition.

TAB. 5.1 – Signification des différents icônes.

5.1.3 Le dialogue de la fragmentation par noyaux.

Les opérations forment un élément important de FW3D-Graphes. La filtration d'arêtes valuées (voir 3.3.2) est implémentée sous forme d'une opération, contrôlée par le dialogue de la figure 5.2.

L'interface de la fragmentation par noyaux (figure 5.3) s'inspire de celle de la filtration d'arêtes valuées, ce qui assure la cohérence visuelle. Quelques propriétés des dialogues de FW3D-Graphes sont particulièrement intéressantes, et les respecter comble notre besoin de cohérence fonctionnelle. Les glissières avec affichage des valeurs permettent une correction facile lors d'erreurs, une valeur fautive se rajustant d'un simple mouvement de souris. Les dialogues déjà présents sont non-bloquants, une caractéristique des interface étendues. Lors de

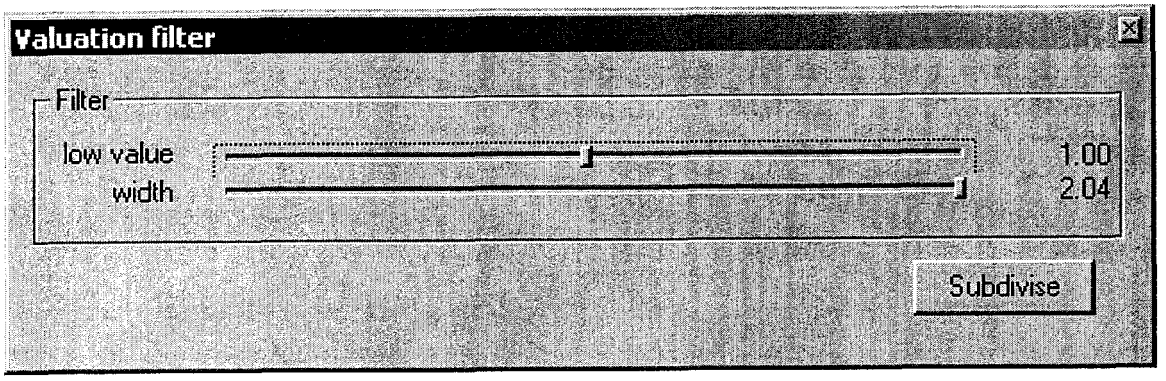


FIG. 5.2 – Dialogue de la filtration d'arêtes valuées.

leur affichage, rien n'empêche l'utilisateur de continuer de manipuler le graphe ou les autres dialogues. Les opérations, comme la filtration d'arêtes valuées, appliquent la manipulation directe. Les changements sont soit effectifs immédiatement, soit accompagnés d'un aperçu du résultat final.

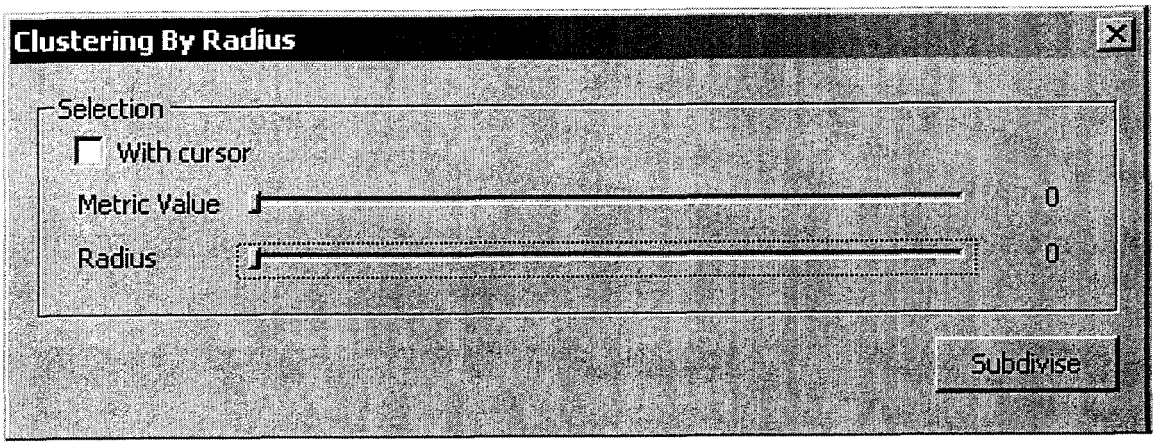


FIG. 5.3 – Première version du dialogue.

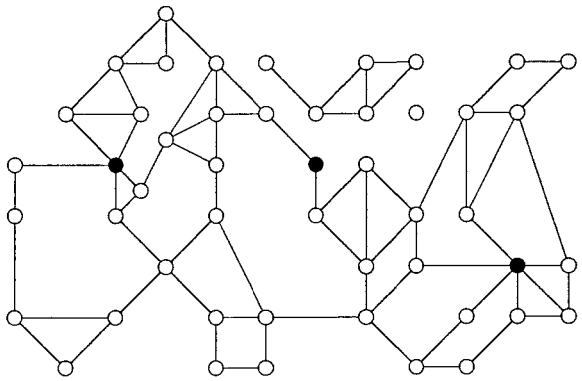
La complétude implique que le dialogue de l'algorithme de fragmentation par noyaux contient un maximum de paramètres non-redondants. On retrouve les deux informations nécessaires à l'algorithme (voir 4.2) : un seuil d'acceptation et un rayon. Théoriquement, les sommets de référence sont adjacents aux arêtes dont le poids, calculé selon la valuation force de cohésion, est supérieur au seuil d'acceptation. En pratique, nous autorisons une sélection plus variée, et ce pour deux raisons. La première est que pour respecter la manière

de faire du dialogue filtration, nous devons dissocier la valuation de notre opération. Dans une telle situation, il est contre-intuitif et incohérent de ne permettre l'opération que lorsque la valuation appliquée est force de cohésion. L'autre raison réside dans l'affichage du graphe. Le placement par simulation physique tend à produire des regroupements visuels de noeuds. Ajouter la sélection manuelle de l'ensemble de sommets de référence dans le graphe affiché permet d'exploiter les regroupements visuels dans la recherche de la bonne partition.

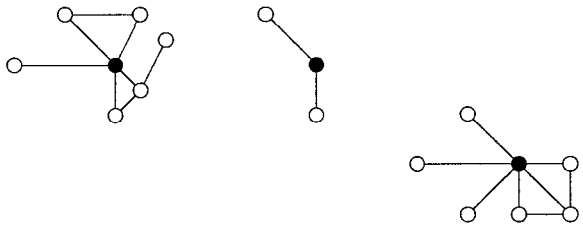
La manipulation directe et le respect des objectifs de notre problématique passent par un affichage en temps réel des effets qu'ont les manipulations du dialogue sur le graphe. Pour cela, il faut réappliquer l'algorithme de fragmentation à chaque changement. L'algorithme étant complexe et exigeant beaucoup de calculs, l'interface est ralentie au point de devenir inutilisable, sauf pour des graphes relativement très petits. À la place d'un affichage complet des changements, on a recourt à un algorithme d'approximation qui donne une prévisualisation partielle. En gros, l'algorithme de prévisualisation génère un sous-graphe $G' = \{V', E'\}$ du graphe G à partir d'un ensemble de sommets de référence V_{ref} et d'un rayon k. L'ensemble V' contient tous les sommets présents dans au moins un noyau, ou $N_k[V_{ref}]$. L'ensemble E' contient toute arête $e \in E$ incidente à deux sommets dans V'. La figure 5.4 illustre un graphe (figure 5.4(a)), une prévisualisation (figure 5.4(b)) pour $k = 1$ et pour un V_{ref} formé des noeuds pleins de la figure 5.4(a), ainsi que la fragmentation par noyaux (figure 5.4(c)) correspondante. Les noyaux à la base des différents sous-graphes sont entourés de pointillés.

Deux défauts majeurs accompagnent cette prévisualisation. Tout d'abord, elle n'affiche qu'une information partielle, les noyaux. Bien que cette information assiste l'utilisateur dans le processus de sélection des paramètres pour l'algorithme de fragmentation, elle ne donne pas de certitudes par rapport aux résultats finaux. Ensuite, il peut se produire des cas où l'information est erronée. En effet, la prévisualisation fonctionne à partir du sous-graphe de G engendré par $N_k[V_{ref}]$. Supposons que deux sommets v et v' de $N_k[V_{ref}]$ sont tous deux de noyaux différents :

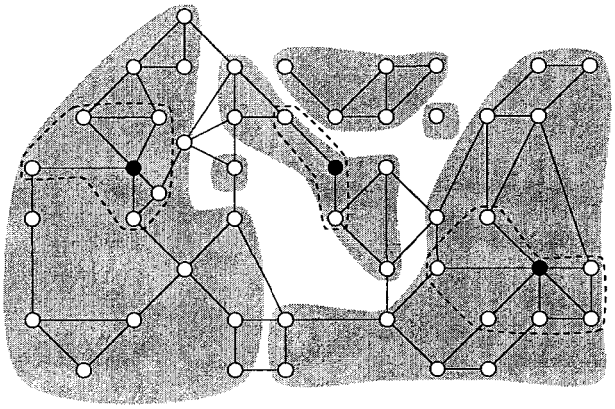
$$v \in N_k[v_i] \text{ et } v' \in N_k[v_j] \tag{5.1}$$



(a) Le graphe.



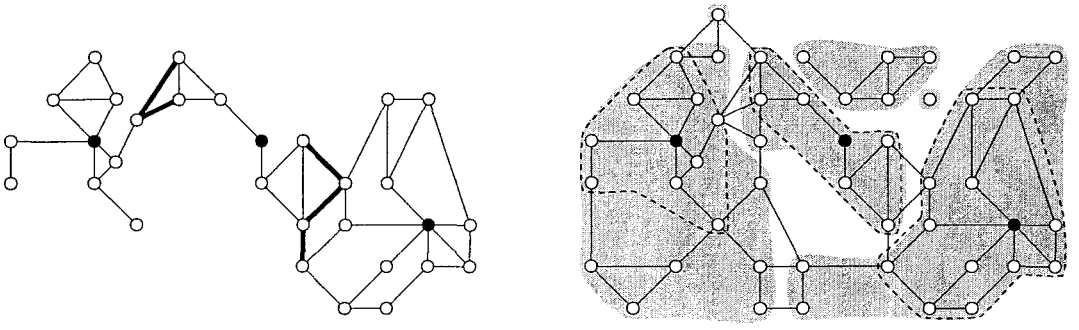
(b) Prévisualisation.



(c) Partition.

FIG. 5.4 – Algorithme de prévisualisation.

pour $v_i \neq v_j$ et $v_i, v_j \in V_{ref}$. Les deux noyaux $N_k[v_i]$ et $N_k[v_j]$ vont être reliés par une arête dans le sous-graphe même s'il n'y a pas de fusion (i.e. $N_k[v_i] \cap N_k[v_i] = \emptyset$). La figure 5.5(a) affiche une prévisualisation du graphe 5.4(a) où les sommets de référence sont les sommets pleins et où le rayon est 2. Les arêtes entre deux noyaux non-connexes (les arêtes erronées) sont en gras. La figure 5.5(b) illustre la partition correspondante, avec les noyaux entourés de pointillés.



(a) Prévisualisation. (b) Partition.

FIG. 5.5 – Erreurs de l'algorithme de prévisualisation.

Les premières manipulations ont dévoilé deux faiblesses à l'interface de la fragmentation par noyaux. Le dialogue ressemble beaucoup trop au dialogue de filtration, ce qui rend possible la confusion entre les deux lors de manipulations rapides. Les graphes affichés étant parfois très denses, la prévisualisation s'avère difficile à distinguer. Des petites retouches ont été apportées au dialogue. Des concepts de discrimination [22] ont été appliqués pour régler le problème de ressemblance. Le bouton « subdivise » du dialogue de la fragmentation par noyaux a été déplacé par rapport à son homologue du dialogue de la fragmentation par filtration, facilitant la discrimination. Le problème de densité visuelle a été corrigé par l'ajout de deux cases à cocher permettant de choisir l'information affichée à l'écran lors des manipulations. Le dialogue final est illustré par la figure 5.6.

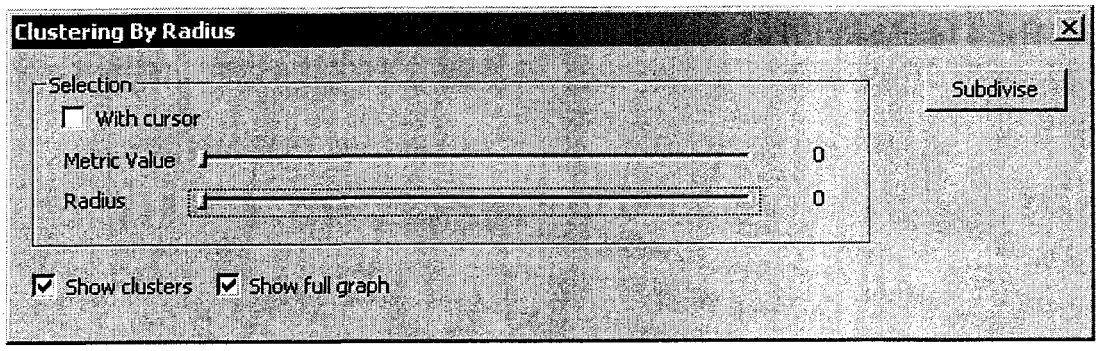


FIG. 5.6 – Deuxième version du dialogue.

5.2 Résultats expérimentaux.

L'algorithme de fragmentation par noyaux a été employé sur les fichiers d'inclusions des logiciels FW3D-Graphes, TEXnicCenter et Torque. Les pages suivantes résument et analysent les résultats obtenus.

5.2.1 FW3D-Graphes.

Le premier test de l'algorithme de fragmentation par noyaux a été effectué sur *FW3D-Graphes*, le logiciel dans lequel il a été implémenté. Le graphe d'inclusions utilisé combine le logiciel, diverses librairies utilisateur et les fichiers des librairies systèmes référés par des commandes d'inclusions. C'est le graphe de la figure 5.7(a). Ce graphe a été valué par force de cohésion (figure 5.7(b)). Les paramètres de l'algorithme ont été ajustés de façon à obtenir une prévisualisation intéressante (figure 5.7(c)). Les valeurs de 1.72 pour le seuil d'acceptation et de 1 pour le rayon génèrent une partition de cinq sous-graphes dont le graphe quotient est illustré à la figure 5.7(d). Trois des sous-graphes n'ont que quelques sommets et sont plutôt inintéressants. L'un provient des sommets isolés, les deux autres des sommets conflictuels. Les deux sous-graphes significatifs proviennent des noyaux. Le premier sous-graphe (figure 5.7(e)) contient deux logiciels externes utilisés par FW3D-Graphes : l'application « Nauty »[53] et une librairie « 3d-Studio ». Séparées de ces logiciels externes et dans le second sous-graphe, il y a l'application elle-même et sa librairie utilisateur (figure 5.7(f)). Les fichiers de la librairie utilisateur étant inclus partout dans l'application, il est logique que les deux modules partagent

un même sous-graphe.

5.2.2 TEXnicCenter.

Le logiciel *TEXnicCenter* [78] est un éditeur à interface graphique pour TEX et LATEX sous license « GNU GPL »[29]. Il contient les modules suivants : un module enveloppant pour l'analyseur syntaxique XML de Microsoft, un éditeur à surlignement syntaxique, une librairie de gestion des expressions régulières (pour le surlignement), un correcteur d'orthographe, deux variantes linguistiques, un assistant de création de nouveaux projets, une librairie d'objets d'interface pour les dialogues, un habillage « Windows XP™ » et le coeur de l'application qui intègre le tout dans un gestionnaire de projets.

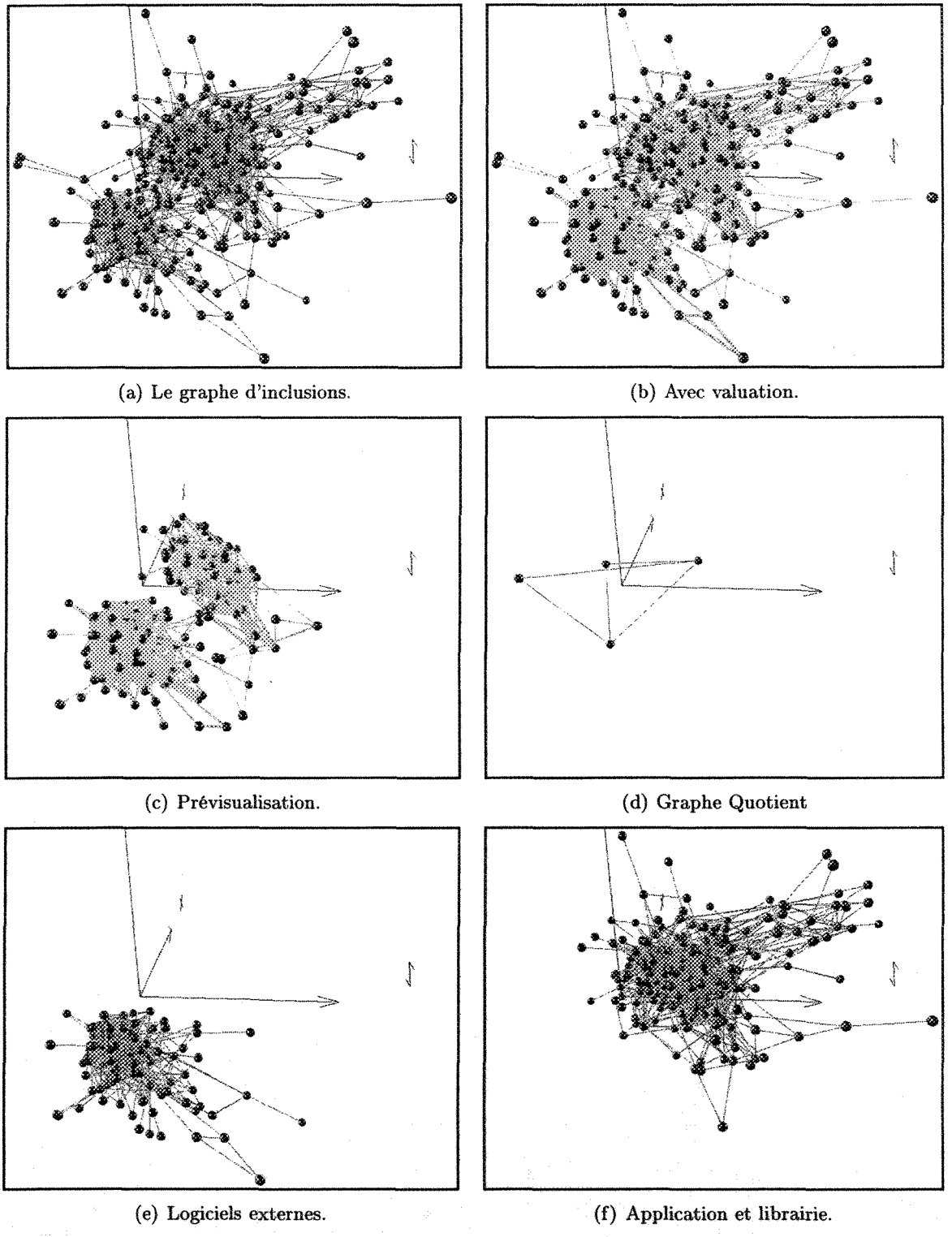
Deux graphes d'inclusions ont été générés pour « TEXnicCenter1 Beta 6.31 ». Le graphe G_1 contient les fichiers de l'application ainsi que les fichiers des librairies systèmes référés dans le code source de l'application (figure 5.8(a)). Le graphe G_2 ne contient que les fichiers de l'application (figure 5.8(b)). Cinq partitions ont été produites à partir de G_1 et G_2. Ces partitions ont été ordonnées en ordre de qualité selon les impressions d'un programmeur familier avec le logiciel, mais pas avec son code source. C'est ce que résume le tableau 5.2. Les partitions sont ordonnées de P_1, la meilleure selon le programmeur consulté, à P_5, la pire.

Partition	Algorithme			Contenu (nombre de sous-graphes)			
	Graphe	f	R	par noyaux	de conflits	non-connexes	noeuds libres
P_1	G_1	1.00	1	4	11	0	0
P_2	G_2	1.96	1	2	1	1	1
P_3	G_2	2.27	1	2	2	1	1
P_4	G_2	1.00	1	3	1	0	1
P_5	G_1	2.16	1	2	1	0	0

TAB. 5.2 – Partitions de TEXnicCenter.

Voici quelques observations sur les partitions obtenues :

- Toutes les partitions sauf P_5 isolent le module d'habillage. N'étant pas référé directement dans les inclusions par les autres fichiers, ce module forme une composante connexe dans le graphe G_2. Dans G_1, il est lié au reste du graphe par



(a) Le graphe d'inclusions.

(b) Avec valuation.

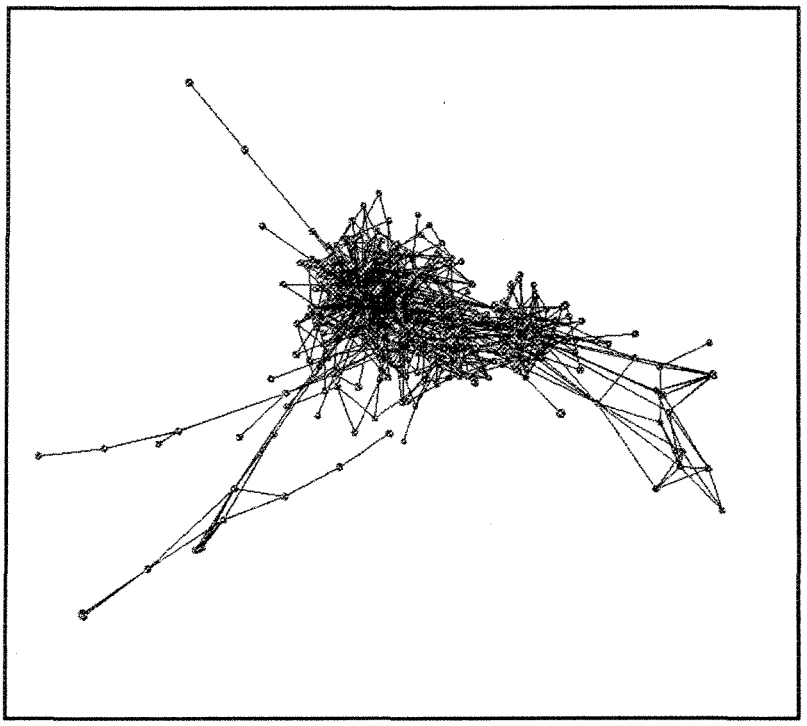
(c) Prévisualisation.

(d) Graphe Quotient

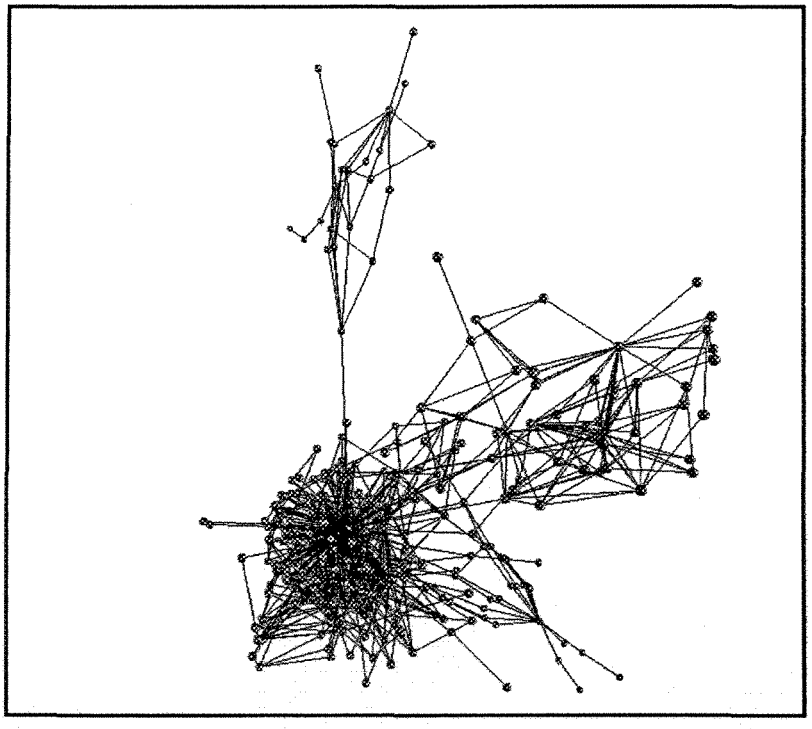
(e) Logiciels externes.

(f) Application et librairie.

FIG. 5.7 – L'algorithme appliqué à FW3D-Graphes.



(a) G_1



(b) G_2

FIG. 5.8 – Graphes d'inclusions de TEXnicCenter.

des fichiers des librairies systèmes.

- La partition P_1 contient un sous-graphe pour l'assistant de création de projets et les deux modules linguistiques. C'est le cas aussi pour les partitions générées à l'aide de G_2 (les partitions P_2,P_3 et P_4). Sans les fichiers systèmes, les sommets de l'assistant de création de projets et des deux modules linguistiques se retrouvent isolés. L'algorithme les place alors tous ensemble dans le sous-graphe des sommets isolés.
- Les partitions P_1 et P_4 isolent avec succès les fichiers des classes enveloppantes de l'analyseur syntaxique sous la forme de deux sous-graphes. Dans les deux cas, un sous-graphe est produit par noyaux et l'autre par une composante connexe des sommets conflictuels.
- Deux partitions, P_2 et P_5, ont des sous-graphes spécifiques pour l'éditeur et son correcteur orthographique. Là aussi, deux sous-graphes sont produits, l'un par noyaux, l'autre en regroupant des sommets conflictuels.
- La partition P_1 est la seule à associer un sous-graphe à la librairie d'objets d'interface. Une partie de la librairie appartient à un sous-graphe produit par noyaux, en compagnie de l'éditeur, du correcteur orthographique et du gestionnaire de projets. L'autre partie a son propre sous-graphe, une composante connexe des noeuds conflictuels.
- Neuf des onze sous-graphes de P_1 issus de sommets conflictuels contiennent un ou quelques sommets des fichiers des librairies systèmes.

Le premier constat face à ces résultats est que l'algorithme identifie avec succès au moins un sous-système dans chaque graphe. Cependant, toutes les partitions contiennent au moins un sous-système séparé en deux sous-graphes avec une partie du sous-sytème dans un sous-graphe produit par noyaux et l'autre partie dans une composante connexe des sommets conflictuels. Le graphe G_2 tend à produire de meilleures partitions que G_1. Les sous-systèmes liés à travers des fichiers systèmes (plus spécifiquement les modules linguistiques, l'assistant de création de projet et l'habillage) se retrouvent en composantes connexes dans G_2 et sont traités plus faci-

lement par l'algorithme. La partition P_1 offre une bonne vue de TEXnicCenter, avec pour seul défaut important de ne pas avoir isolé avec succès l'éditeur et son correcteur orthographique du reste du projet.

D'autres essais sur TEXnicCenter.

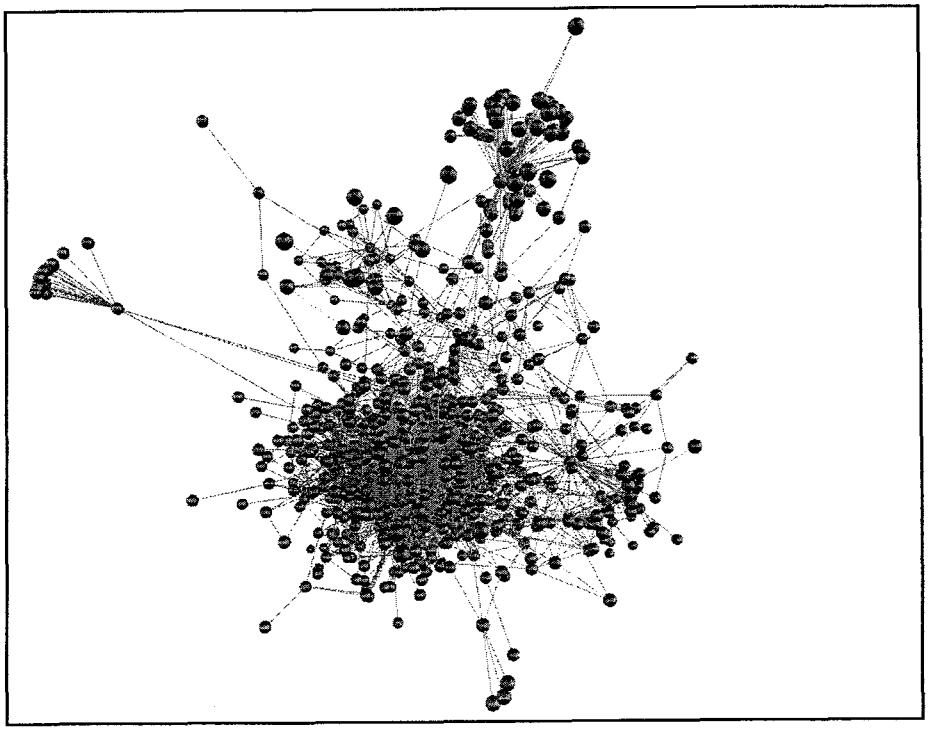
L'algorithme de fragmentation par noyaux a été appliqué récursivement dans une tentative d'améliorer la partition P_1. Il s'agit de reproduire P_1 et d'appliquer la fragmentation par noyaux sur le plus gros sous-graphe obtenu. Bien que cette idée soit prometteuse, elle a rencontré une limitation de l'algorithme. La fragmentation par noyaux ne fonctionne pas lorsque la connectivité est trop élevée. Les arêtes identifiées pour construire les noyaux sont alors incidentes à des sommets communs et l'algorithme ne trouve qu'un seul noyau. Une deuxième tentative, en enlevant au préalable les fichiers systèmes du sous-graphe, a donné le même résultat. La connectivité reste trop élevée pour raffiner la partition récursivement.

5.2.3 Torque.

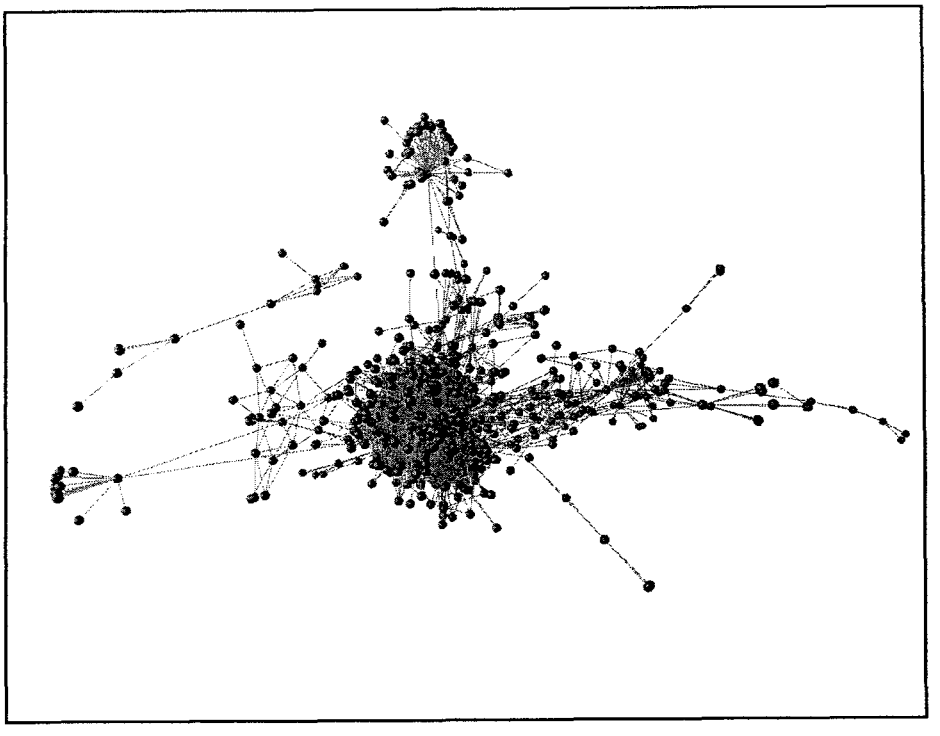
Le troisième logiciel est *Torque* [30], un moteur de jeu commercial multiplateforme. Torque inclut la gestion de scripts, le moteur graphique, le moteur audio, le support réseau et les outils nécessaires pour convertir, construire ou importer la matière première d'un jeu (sons, modèles, images...). Le code source du logiciel est composé du moteur lui-même avec ses librairies internes et ses outils, ainsi que de quelques librairies externes : support pour divers formats de fichiers de médias (jpeg, png, gif, Ogg Vorbis), gestion audio et vidéo (OpenAL, DirectX, OpenGL), importateurs de modèles (3D Studio Max, DTS, Maya).

Cinq partitions ont été produites à partir de deux graphes d'inclusions de Torque. Comme pour TEXnicCenter, un premier graphe d'inclusions G_1 (figure 5.9(a)) contient les inclusions de Torque ainsi que les liens vers les fichiers systèmes. Le graphe G_2 (figure 5.9(b)) ne contient que les relations d'inclusions entre éléments de Torque. Les cinq partitions obtenues ont été classifiées par une personne familière avec le code source de Torque. Le tableau 5.3 contient les cinq partitions ordonnées de la meilleure (P_1) vers la pire (P_5).

Voici quelques observations sur les partitions, toujours selon la personne ressource :



(a) G_1



(b) G_2

FIG. 5.9 – Graphes d'inclusions de Torque.

Partition	Algorithme			Contenu (nombre de sous-graphes)			
	Graphe	f	R	par noyaux	de conflits	non-connexes	noeuds libres
P_1	G_1	1.99	1	5	10	0	1
P_2	G_2	1.50	1	5	2	1	0
P_3	G_2	2.09	1	5	9	1	0
P_4	G_2	2.16	2	3	4	1	0
P_5	G_1	2.21	1	3	17	0	1

TAB. 5.3 – Partitions de Torque.

- Aucune des cinq partitions ne divise Torque lui-même (librairies, outils, moteur) ni ne le dissocie des librairies externes audio-vidéo.
- La partition P_1 est la meilleure car elle isole le mieux Torque des librairies externes. Les partitions P_2 et P_3 ressemblent beaucoup à P_1, mais n'ont pas la même qualité de découpage.
- La librairie externe de gestion des fichiers 3D Studio Max ainsi que celle de JPEG sont isolées avec aisance par l'algorithme.
- La partition P_5 est particulièrement mauvaise. Il n'y a pas de liens logiques évidents dans les regroupements obtenus. La partition P_4 ressemble à P_5, mais certains regroupements sont logiques.
- Il manque de sous-graphes par noyaux dans toutes les partitions. D'un autre côté, l'algorithme produit beaucoup de sous-graphes de conflits. Le contenu de certains des sous-graphes de conflits est aisément associable à l'un ou l'autre des sous-graphes par noyaux.
- Testé sur une version modifiée de Torque, l'algorithme a isolé les éléments ajoutés.
- Les résultats sont moyens. Il y a une bonne partition des librairies externes, mais le coeur de Torque ne se sépare pas.
- La compréhension des sous-systèmes de taille raisonnable est facilitée par leur mise en sous-graphes et leur affichage sous cette forme.

Les résultats de la fragmentation de Torque ne sont guère surprenants. Le moteur

ne se divise pas en sous-sytèmes à cause de la manière dont le code est écrit. Les inclusions n'y sont pas gérées de manière intelligente. L'exemple suivant explique une gestion intelligente des inclusions. Supposons quatre fichiers A, B, C et D. Le fichier A réfère à des éléments de code dans B et dans C. Le fichier B réfère à des éléments de code dans C et dans D. Le fichier C réfère à D. Pour que la compilation du code source se fasse, il faut que ces références soient présentes sous forme d'inclusions. Or, il y a plus d'une manière de formuler des inclusions qui respectent les dépendances énoncées. La figure 5.10 contient le code source partiel de deux versions. La première se contente des commandes d'inclusions nécessaires. La deuxième applique littéralement les relations décrites. Les deux graphes d'inclusions sont illustrés à la figure 5.11. Le graphe d'inclusions de la figure 5.11(b) est plus complexe que le graphe de la figure 5.11(a) et sa connectivité est plus élevée. Par conséquent, il est plus difficile à fragmenter. C'est le même problème avec les inclusions de Torque. Au lieu d'être ajoutées au besoin et hiérarchiquement, les commandes d'inclusions se répètent bêtement de fichiers à fichiers. On se retrouve avec une multitude de fichiers inclus un peu partout qui rendent le graphe d'inclusions fortement connexe.

Version 1

Fichier A

```
#include "B"
```

Fichier B

```
#include "C"
```

Fichier C

```
#include "D"
```

Version 2

Fichier A

```
#include "B"
#include "C"
```

Fichier B

```
#include "C"
#include "D"
```

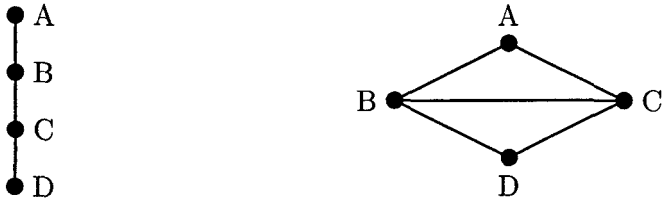
Fichier C

```
#include "D"
```

FIG. 5.10 – Deux codes sources C.

5.3 Analyse de la fragmentation par noyaux.

Maintenant que l'algorithme de fragmentation par noyaux a été utilisé dans un contexte d'ingénierie inverse, il est temps de vérifier s'il correspond à notre problématique et



(a) Version 1. (b) Version 2.

FIG. 5.11 – Graphes d'inclusions des codes sources de la figure 5.10.

quels sont les améliorations à y apporter.

5.3.1 La fragmentation par noyaux et nos objectifs.

Dans le chapitre 2, nous avons défini un but pour la fragmentation par noyaux : combler quatre des six objectifs clés de l'ingénierie inverse (voir 2.2.2). Les prochaines lignes reviennent sur l'algorithme par rapport à ces objectifs.

Synthétiser des abstractions.

Synthétiser des abstractions consiste à générer des partitions de façon à diminuer la quantité d'information que doit assimiler le programmeur dans son processus de compréhension d'un système logiciel. C'est le but principal de la fragmentation par noyaux et elle remplit cet objectif, mais avec un bémol. La qualité et la validité des regroupements dans les partitions produites dépendent de la connectivité du graphe. Lorsqu'un système logiciel contient des sous-sytèmes avec faible connectivité entre eux, mais avec une forte connectivité interne, la fragmentation par noyaux tend à produire des partitions qui ont du sens. Plus la connectivité entre sous-systèmes est élevée, toutefois, moins les partitions générées sont logiques. Idéalement, ce n'est pas un problème, puisqu'un système logiciel bien programmé assure des communications minimales entre ses différents sous-systèmes. Malheureusement, nous avons constaté que ce n'est pas toujours le cas en réalité. Par exemple, le logiciel Torque présente une très forte connectivité entre sous-systèmes, et isoler d'autres sous-systèmes que quelques librairies externes est presque impossible avec l'algorithme actuel.

Récupérer de l'information perdue.

Récupérer de l'information perdue, c'est générer des données utiles au programmeur à partir du code source. Les trois graphes d'ingénierie inverse qui servent à la fragmentation documentent les relations entre les différents éléments du code source. Lorsqu'une partition est obtenue, le graphe quotient illustre quels sont les sous-systèmes qui interagissent ensemble. Les sous-graphes regroupent les éléments selon leur liens. Toutes ces nouvelles informations sont mises à la disposition du programmeur pour l'assister dans le processus d'ingénierie inverse.

Prendre en charge la complexité.

Un code source contient de grandes quantités d'information. La fragmentation par noyaux rend cette information plus abordable par des sous-graphes affichables indépendamment, qui sont des structures plus petites que le système logiciel entier et plus faciles à comprendre. L'utilisation de la fragmentation par noyaux se fait à travers une interface simple qui s'opère de la même façon pour des graphes de toutes tailles. La complexité est donc bien prise en charge.

Fournir des vues alternatives.

Comme il a été mentionné en 2.2.2, produire des partitions d'un graphe issu d'un système logiciel comble en soit l'objectif des vues alternatives consistant à donner des aperçus différents de l'information.

Faciliter la réutilisation et trouver des effets de bord.

Les deux derniers objectifs clés de l'ingénierie inverse n'ont pas été jugés très pertinents pour cette recherche. Pourtant, la fragmentation par noyaux a prouvé lors des expérimentations pratiques qu'elle remplissait ces deux objectifs. Plus la connectivité d'un sous-système d'un logiciel avec les autres sous-systèmes est faible, plus ce sous-système est propice à la réutilisation. La fragmentation par noyaux a donc plus de facilité à isoler en sous-graphes les sous-systèmes très réutilisables que ceux qui le sont moins. Pour ce qui est des effets de bord, la

fragmentation par noyaux met en évidence certaines erreurs de conception. Par exemple, avec Torque, la gestion brouillon des relations d'inclusions apparaît dans les partitions produites. Bien qu'involontairement, la fragmentation par noyaux s'avère utile pour ces deux objectifs.

5.3.2 Améliorations possibles à la fragmentation par noyaux.

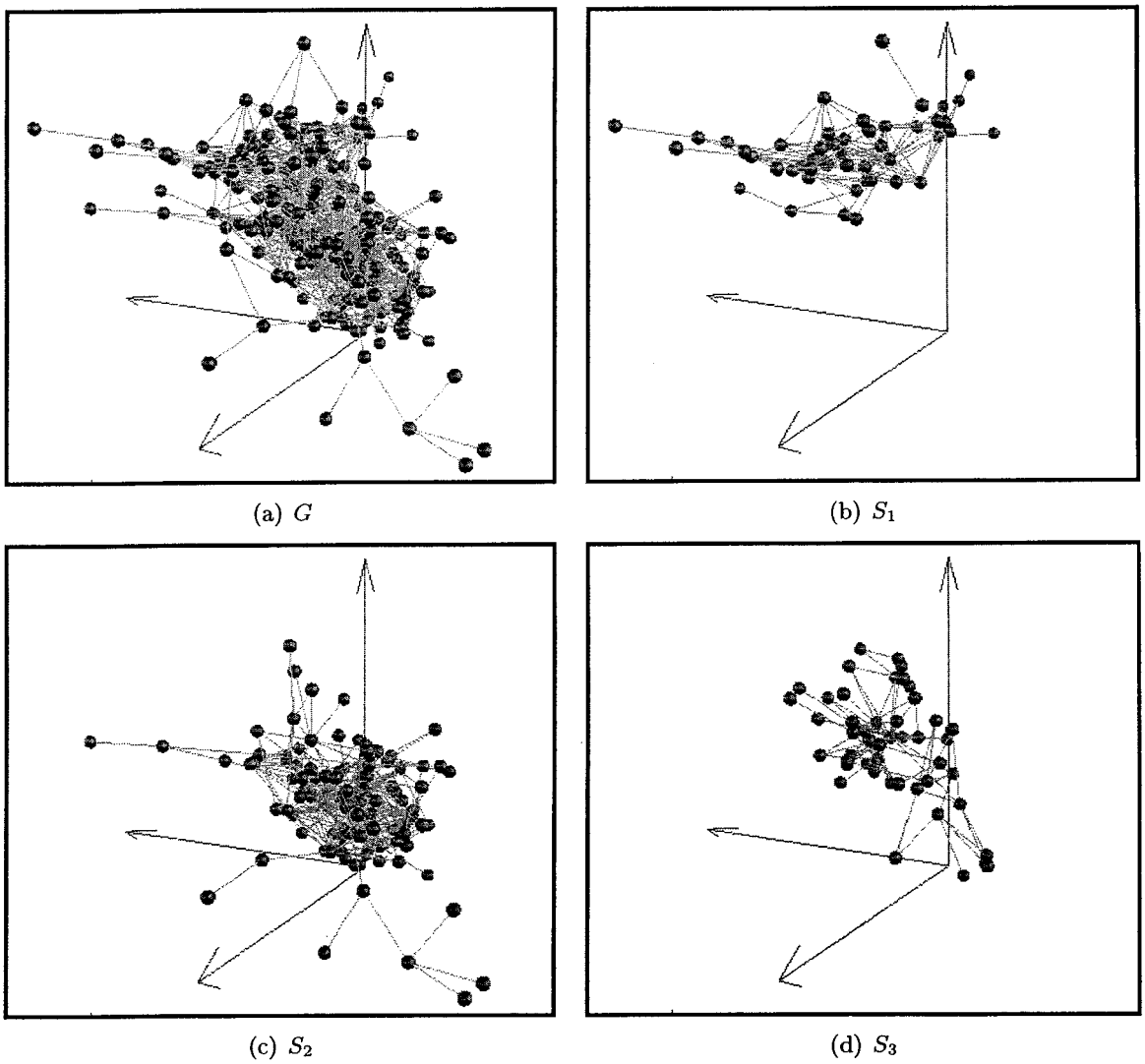
Voici quelques améliorations à apporter à l'algorithme de fragmentation par noyaux et à son implémentation. Les améliorations évoquées proviennent de constats d'analyses ou de remarques des utilisateurs.

Meilleure répartition des noeuds conflictuels.

Certaines partitions donnent des sous-graphes de sommets conflictuels de taille relativement importante. Prenons pour exemple un graphe G et les trois principaux sous-graphes d'une partition P de G (figure 5.12). Le sous-graphe S_3 est composé de sommets conflictuels obtenus lors de la création des sous-graphes S_1 et S_2. Dans de tels cas, les noeuds conflictuels regroupés ensemble n'ont pas toujours des relations logiques suffisantes pour justifier leur appartenance à un même sous-graphe. De plus, certains sommets présentent une forte connectivité avec un sous-graphe particulier par rapport aux autres, ce qui indique qu'ils ne sont pas dans le bon sous-graphe. Le traitement des sommets conflictuels nécessite une révision afin que les regroupements obtenus aient plus de sens.

Identifier et retirer les noeuds omniprésents.

Les auteurs des systèmes logiciels d'ingénierie inverse *Rigi* [58] et *Bunch* [50] ont constaté l'existence d'éléments (fonctions, objets, fichiers...) présents un peu partout dans le code. Ils apparaissent dans un graphe d'ingénierie inverse comme des noeuds dont le voisinage englobe une forte proportion des autres noeuds du graphe. Il s'agit souvent d'éléments fondamentaux comme des routines d'entrée/sortie génériques ou des structures de données. La présence de ces éléments *omniprésents* vient nuire à la recherche d'une bonne partition d'ingénierie inverse. Nous avons constaté la présence d'éléments omniprésents lors des tests de la fragmentation par noyaux. Retirer les éléments omniprésents diminue la connectivité, donc



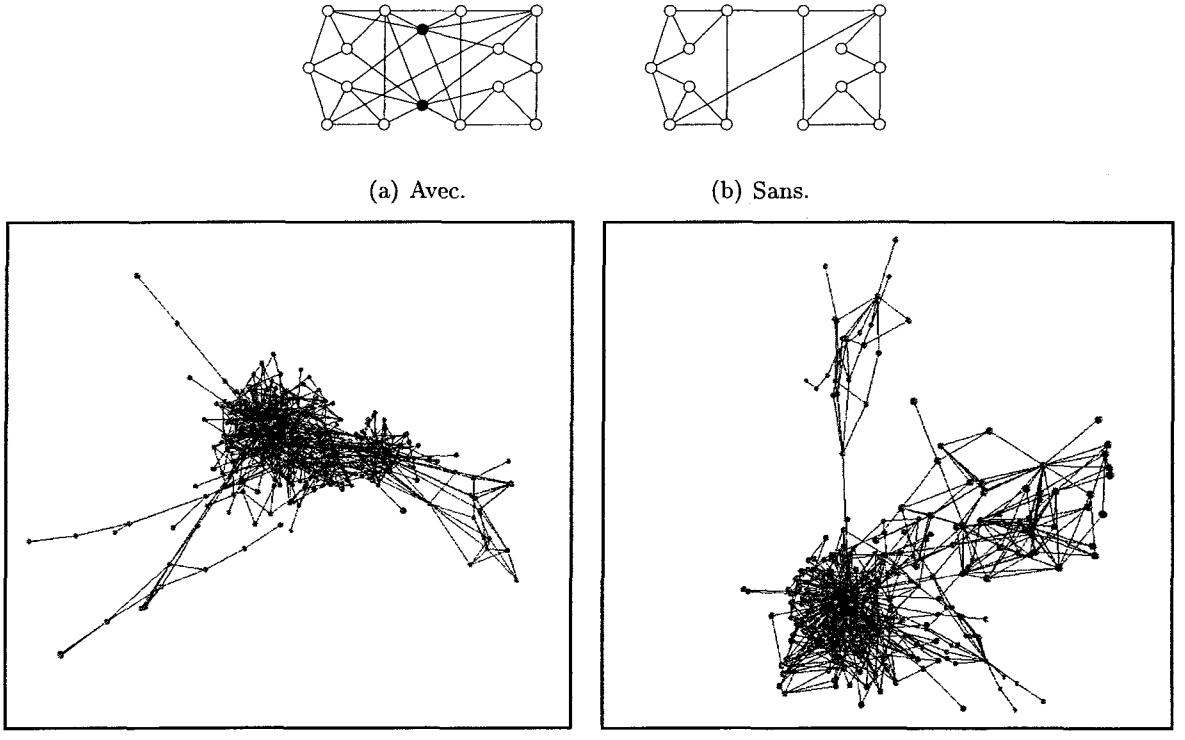
(a) G (b) S_1

(c) S_2 (d) S_3

FIG. 5.12 – Fragmentation avec sommets conflictuels.

facilite l'identification des sous-graphes.

Les deux noeuds pleins du graphe de la figure 5.13(a) sont des exemples d'éléments omniprésents dans un graphe d'ingénierie inverse. On constate que le graphe ne semble pas contenir de sous-graphes. Si on retire les éléments considérés omniprésents, cependant, on découvre des sous-graphes potentiels (figure 5.13(b)). Voici un autre exemple obtenu dans une sitution d'ingénierie inverse. Nous avons généré deux graphes d'inclusions à partir du code source de TEXnicCenter. Le premier graphe (figure 5.13(c)) contient des fichiers sources des librairies systèmes en plus des fichiers sources de l'application. Le second graphe (figure 5.13(d)) n'inclut que les fichiers sources de l'application. L'absence des fichiers de la librairie système simplifie le graphe et des agrégats apparaissent dans la représentation visuelle.



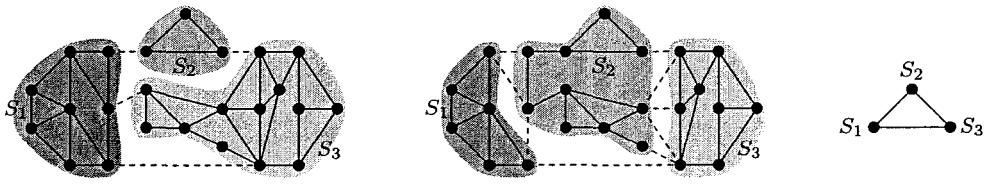
(a) Avec. (b) Sans.

(c) Avec. (d) Sans.

FIG. 5.13 – Effets des fichiers omniprésents.

Clarifier le graphe quotient.

Chaque utilisation de l'algorithme génère deux produits : une partition et son graphe quotient. Nous avons vu en 1.1.3 que les sommets du graphe quotient représentent des sous-graphes de la partition et les arêtes indiquent la présence de relations externes entre les sous-graphes. Indiquer la présence ou l'absence de relations externes n'est pas une information suffisante à la comparaison mutuelle de plusieurs partitions. Deux partitions peuvent avoir des graphes quotients identiques tout en ayant une connectivité externe différente. Prenons par exemple deux partitions d'un même graphe, P_1 et P_2, illustrées respectivement par les figures 5.14(a) et 5.14(b). Les deux partitions produisent le même graphe quotient (figure 5.14(c)), pourtant P_1 a 4 arêtes externes, alos que P_2 en a 9.



(a) P_1. (b) P_2. (c) G_Q.

FIG. 5.14 – Deux partitions P_1 et P_2 ayant le même graphe quotient G_Q.

En génie logiciel, une connectivité externe réduite indique souvent une bonne partition du graphe. C'est d'ailleurs la caractéristique principale d'une bonne partition dans les algorithmes Bunch (voir 3.3.1) et filtration d'arêtes valuées par force de cohésion (voir 3.3.2). La connectivité affichée à même le graphe quotient facilite la tâche de l'utilisateur.

Corriger la prévisualisation et l'interface.

L'algorithme n'ayant pas les performances nécessaires pour un affichage temps réel, nous avons recours à un algorithme de prévisualisation pour donner à l'utilisateur un aperçu des conséquences de ses actions. La prévisualisation actuelle affiche les noyaux avec quelques erreurs, ce qui complexifie la tâche de l'utilisateur, d'où la nécessité d'un algorithme de prévisualisation plus précis.

Certains utilisateurs se sont plaints de l'interface graphique. La séparation de paramètres cruciaux à l'algorithme sur plusieurs fenêtres, en particulier les valuations, augmente le nombre d'étapes nécessaires à l'utilisation et ralentit l'apprentissage de l'interface. Cette manière de travailler est justifiée par la cohérence et la réutilisation, puisque les valuations ont plus d'un usage. Une étude plus détaillée de l'interface graphique s'impose afin de trouver un meilleur équilibre entre simplicité et cohérence.

Améliorer les performances de l'algorithme.

Deux étapes de la fragmentation par noyaux effectuent chacune un calcul des distances du graphe pour chaque noyau. Avec une meilleure gestion de l'information, il devrait être possible de ramener le nombre de calculs des distances à une fois pour chacune de ces deux étapes. Ainsi, la complexité temporelle de l'algorithme est réduite. Moins l'utilisateur a besoin d'attendre après l'information, plus ses impressions de l'algorithme sont positives.

Utiliser d'autres valuations.

L'algorithme de fragmentation par noyaux présenté ici fonctionne autour de la valuation force de cohésion. Les arêtes au-dessus du seuil d'acceptation servent à déterminer les sommets de référence et, à travers eux, le résultat. Voici quelques changements à cette mécanique dont les résultats n'ont pas été évalués.

Filtrer dans un intervalle contrôlé par l'utilisateur, au lieu de filtrer toutes les arêtes sous le seuil d'acceptation, offre plus de souplesse. Cette technique permet même de retirer une partie des sommets omniprésents et de contourner le problème de connectivité trop forte de certains graphes. La validité des résultats générés ainsi reste toutefois à vérifier.

La théorie des graphes contient nombre de valuations autres que la force de cohésion, et certaines ont un bon potentiel pour l'obtention d'arêtes ou de sommets centraux à des sous-graphes. De nouvelles valuations sont d'autres moyens de trouver de bon sommets de référence pour la fragmentation.

La version actuelle de l'algorithme mesure les distances à deux étapes : lors de la construction des noyaux et lors de la répartition des sommets. Dans les deux cas, les distances

sont mesurées sans examiner le poids des arêtes. Que l'arête $\{u, v\}$ du graphes soit pondérée ou non, l'algorithme considère que la distance entre les sommet u et v est 1. L'utilisation des distances pondérées à l'une ou à l'autre de ces étapes (ou aux deux) n'a pas été testée. Avec différentes valuations appropriées qui lient poids et structure du graphe, on a potentiellement une plus grande variété de résultats. Plus on explore une vaste section de l'espace des partitions, plus l'utilisateur a de chances de trouver la partition qui lui convient.

CONCLUSION.

Ce travail a porté sur l'étude de fragmentations de graphes dans une optique d'analyse de systèmes logiciels. Plusieurs types de graphes sont issus d'éléments constitutifs du code source d'un système logiciel et des relations entre ces éléments. La fragmentation de ces graphes produit des regroupements parmi les éléments représentés, ce qui fournit une vue à un niveau d'abstraction différent du système logiciel. Pour que l'abstraction soit valide, il faut toutefois que la partition respecte certains objectifs de l'ingénierie inverse et certains besoins de l'utilisateur. Une partition doit avoir, entre autres, une faible connectivité entre ses sous-graphes, une forte connectivité à l'intérieur des sous-graphes et une certaine logique sémantique dans les regroupements. La production automatisée de regroupements sémantiques dans de l'information pose problème. Ce n'est pas une capacité des ordinateurs actuels. Pour arriver à un bon résultat, l'utilisateur doit disposer d'un certain contrôle de la fragmentation à travers des paramètres.

Plusieurs algorithmes de fragmentation de la littérature existante ont été étudiés dans une tentative d'en trouver un respectant à la fois nos critères d'ingénierie inverse et concédant à l'utilisateur le niveau de contrôle attendu. Nous avons regardé des algorithmes de coloration, les cliques, la dominance, la dominance par distance, des algorithmes de routage, la filtration d'arêtes et le logiciel d'optimisation Bunch. Tous les algorithmes observés présentent des limitations, parfois mineures, les empêchant de remplir l'ensemble de nos objectifs. Cependant, ils regorgent d'idées et de concepts potentiellement utiles, qui nous ont inspirés dans

l'introduction d'un nouvel algorithme : la fragmentation par noyaux.

La fragmentation par noyaux fonctionne en deux étapes. Une valuation est appliquée sur le graphe à fragmenter. Cette valuation, dans notre cas la valuation force de cohésion, associe un poids à chaque arête selon sa centralité par rapport à des sous-graphes possibles. Une sélection est effectuée dans les arêtes en comparant le poids de chacune à un intervalle paramétré. Les arêtes sélectionnées sont jugées centrales aux sous-graphes de la partition. Les k-voisinage de ces arêtes, où k est un paramètre de la fragmentation, forment des noyaux. Les noyaux sont à la base des sous-graphes de la partition. Les sommets hors noyaux sont répartis dans la partition selon leur relation avec les noyaux et des noyaux trop proches (qui partagent des sommets communs) sont fusionnés.

Implémentée dans le logiciel FW3D-Graphes, la fragmentation par noyaux a été testée avec différents systèmes logiciels. Dans chaque cas pratique (Torque, TEXnicCenter, Fw3D-Graphes), elle parvient à une nouvelle abstraction du système logiciel selon les caractéristiques recherchées. L'objectif d'abstraction d'ingénierie inverse est rempli par l'obtention de partitions dont les sous-graphes ont un sens sémantique. De nouvelles informations et des vues alternatives sont produites à travers les graphes générés, comme le graphe quotient qui représente les relations entre sous-systèmes. L'algorithme est prévu pour s'appliquer sur des graphes de toutes tailles. Les structures des partitions générées par l'algorithme mettent en évidence les composantes logicielles réutilisables, mais aussi certaines erreurs de conceptions. Finalement, le logiciel FW3D-Graphe ainsi que les paramètres et l'interface de la fragmentation par noyaux comblent les besoins utilisateur.

La mise en pratique de l'algorithme nous a aussi permis de constater certaines de ses limitations. Dans sa version actuelle, les performances laissent à désirer. Les gros graphes sont longs à fragmenter et viennent avec une consommation de mémoire élevée. La manipulation directe est une caractéristique importante pour faciliter la tâche de l'utilisateur, mais l'information produite par la prévisualisation de l'algorithme de fragmentation est incomplète et même parfois erronée. L'algorithme a été prévu pour travailler avec des systèmes logiciels qui présentent une bonne ingénierie telle que les fonctionnalités sont regroupées en sous-systèmes

faiblement interconnectés. Or, les graphes issus d'un système logiciel mal programmé ou mal entretenu sont probablement des graphes « enchevêtrés », c'est-à-dire des graphes qui ne présentent pas de sous-ensembles dans la connectivité de leurs sommets. La fragmentation par noyaux est incapable de bien fragmenter des graphes enchevêtrés, ce qui limite son usage à des graphes générés sur des systèmes logiciels bien programmés. Cette restriction est agaçante, les systèmes logiciels produisant des graphes enchevêtrés étant souvent rencontrés en ingénierie inverse.

Les problèmes liés à la manipulation directe et aux performances ont pour source l'implémentation plutôt que les concepts fondamentaux de la fragmentation par noyaux. Pour corriger ces problèmes, il suffit de revoir les algorithmes de façon à ce qu'ils donnent de meilleurs résultats ou effectuent plus rapidement les calculs. Le problème posé par les graphes enchevêtrés est plus complexe à résoudre. Les principes de fonctionnement de la fragmentation par noyaux étant bons, la solution ne réside pas dans leur changement. Il faut plutôt regarder au niveau du graphe. Les graphes enchevêtrés sont mentionnés fréquemment dans la littérature sur les graphes en ingénierie inverse. Une solution possible, et exploitée dans Bunch, consiste à diminuer la connectivité de ces graphes. Une technique utilisée est d'enlever les sommets omniprésents. Trouver des techniques similaires capables de transformer un graphe enchevêtré en diminuant sa connectivité est une voie de recherche prometteuse. De telles techniques ne bénéficient pas uniquement à la fragmentation par noyaux, mais aussi aux autres algorithmes d'ingénierie inverse qui font face à ce problème, comme Bunch ou la fragmentation par filtration d'arêtes.

Régler le problème des graphes enchevêtrés permet un nouvel usage de l'algorithme qui n'a pas pu être testé de manière concluante : l'utilisation récursive. Dans bien des cas, une application récursive de l'algorithme n'est pas nécessaire. Il arrive toutefois qu'un sous-système soit suffisamment complexe pour que certains des sous-graphes produits s'avèrent encore relativement difficiles à comprendre. C'est le cas avec le logiciel Torque. L'application récursive de la fragmentation par noyaux afin de changer le niveau d'abstraction des plus gros sous-graphes n'a pas réussi. Chaque sous-graphe d'une partition produite par noyaux

est un ensemble de sommets fortement connexes isomorphe à un graphe enchevêtré, donc presque indivisible avec la fragmentation par noyaux. Une bonne technique pour diminuer la connectivité des graphes enchevêtrés s'applique aussi au sous-graphe, ce qui ouvre la porte à une application récursive de la fragmentation par noyaux et, peut-être, de meilleurs résultats.

Un autre usage de l'algorithme n'a pas été étudié dans le cadre de cette recherche : son emploi sur d'autres graphes que ceux produits en ingénierie du logiciel. Si l'on modélise un domaine de connaissance quelconque sous la forme de graphes, les regroupements d'éléments fortement connexes présentent un intérêt particulier. Prenons pour exemple le sujet de la musique classique. Supposons un graphe où les noeuds représentent des compositeurs, des interprètes ou des instruments. Il existe une arête entre deux noeuds s'il existe un lien entre les entités que représente ce noeud, par exemple si un compositeur travaille pour un instrument, si un interprète joue d'un instrument ou d'un compositeur, si un interprète travaille avec un autre interprète, etc. Ce graphe devient rapidement très complexe, mais des sous-ensembles s'y forment, comme des interprètes qui jouent ensemble ou quelques compositeurs qui travaillent sur les mêmes instruments... Les réseaux informatiques sont un autre exemple de domaine d'application possible pour la fragmentation par noyaux. Identifier des regroupements dans un réseau informatique est important lorsque vient le temps d'établir une stratégie de routage (voir 3.2). Ce ne sont là que deux exemples parmi plusieurs.

La fragmentation par noyau présente, que ce soit au niveau du domaine d'emploi, des valuations utilisées ou de la manière de faire les choses, tout un potentiel encore inexploité.

BIBLIOGRAPHIE

[1] ABIAN, A. *The Theory of Sets and Transfinite Arithmetic.* W.B. Saunders Company, 1965.

[2] AIROLDI, J.-P. *Le terrier du Campagnol Terrestre considéré comme un graphe.* Presses Polytechniques Romandes, 1980, pp. 145–149.

[3] AUBER, D., DELEST, M., AND CHIRICOTA, Y. Strahler based graph clustering using convolution. In *8th International Conference on Information Visualisation, IV* (2004), IEEE Computer Society, pp. 44–51. `http://dept-info.labri.u-bordeaux.fr/~auber/documents/publi/auberIV04London.pdf`.

[4] AUILLANS, P., AND BAUDON, O. Graph clustering for very large topic maps.

[5] AWERBUCH, B., BAR-NOY, A., LINIAL, N., AND PELEG, D. Compact distributed data structures for adaptive routing. In *STOC '89 : Proceedings of the twenty-first annual ACM symposium on Theory of computing* (New York, NY, USA, 1989), ACM Press, pp. 479–489. `http://doi.acm.org/10.1145/73007.73053`.

[6] BALL, T., AND EICK, S. G. Software visualization in the large. *IEEE Computer 29*, 4 (1996), 33–43. `http://citeseer.nj.nec.com/ball96software.html`.

[7] BATTISTA, G. D., EADES, P., TAMASSIA, R., AND TOLLIS, J. G. *Graph Drawing : Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[8] BENSON, H., LEFEBVRE, N., SÉGUIN, M., AND VILLENEUVE, B. *Physique I : Mécanique.* Éditions du Renouveau Pédagogique Inc., 1991.

[9] BERGE, C. *Graphes et hypergraphes.* Dunod, 1970.

[10] BOLLOBÁS, B. *Graph Theory : An Introductory Course.* Springer-Verlag, 1979.

[11] BUSACKER, R. G., AND SAATY, T. L. *Finite Graphs and Networks : An Introduction with Applications.* International Series in Pure and Applied Mathematics. McGraw-Hill Book Company, 1965.

[12] BUSS, E. B., DE MORI, R., GENTLEMAN, W. M., HENSHAW, J., JOHNSON, H., KONTOGIANNIS, K., MERLO, E., MULLER, H. A., MYLOPOULOS, J., PAUL, S., PRAKASH, A., STANLEY, M., TILLEY, S. R., TROSTER, J., AND WONG, K. Investigating reverse engineering technologies for the CAS program understanding project. *IBM Systems Journal 33*, 3 (1994), 477–500. `http://citeseer.nj.nec.com/buss94investigating.html`.

[13] CHEN, C. Information visualization. *Information Visualization 1*, 1 (2002), 1–4.

[14] CHIKOFSKY, E. J., AND JAMES H. CROSS, I. Reverse engineering and design discovery : A taxonomy. *IEEE Software 7*, 1 (1990), 13–17.

[15] CHIRICOTA, Y. Systèmes masses-ressorts et graphes. Séminaire au LaBRI (Laboratoire Bordelais de Recherche en Informatique), Juin 2001.

[16] CHIRICOTA, Y., JOURDAN, F., AND MELANÇON, G. Software components capture using graph clustering. In *IWPC '03 : Proceedings of the 11th IEEE International Workshop on Program Comprehension* (Washington, DC, USA, 2003), IEEE Computer Society, p. 217.

[17] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction à L'algorithmique.* Dunod, 1994.

[18] CREEK, A. Forces of nature - angle : An experimental test-bed for graph embedders and the big bang : a new force directed embedder. `http://citeseer.ist.psu.edu/557404.html`.

[19] DE WERRA, D. *Fantaisies chromatiques sur diverses partitions.* Presses Polytechniques Romandes, 1980, pp. 127–143.

[20] DIX, A., FINLAY, J., ABOWD, G., AND BEALE, R. *Human-Computer Interaction.* Prentice Hall, 1993.

[21] DOVAL, D., MANCORIDIS, S., AND MITCHELL, B. S. Automatic clustering of software systems using a genetic algorithm. In *IEEE Proceedings of the 1999 Int. Conf. on Software Tools and Engineering Practice (STEP'99)* (1999). `http://citeseer.nj.nec.com/doval98automatic.html`.

[22] DUBUIS, E. Graphical user interfaces : Mess them up! In *Winter School of Computer Graphics 1996* (1996). `http://citeseer.ist.psu.edu/1788.html`.

[23] DWYER, T. Three dimensional UML using force directed layout. In *Australian Symposium on Information Visualisation, (invis.au 2001)* (Sydney, Australia, 2001), P. Eades and T. Pattison, Eds., ACS. `http://citeseer.nj.nec.com/dwyer01three.html`.

[24] EADES, P. A heuristic for graph drawing. *Congressus Numerantium 42* (1984), 149–160.

[25] EDACHERY, J., SEN, A., AND BRANDENBURG, F.-J. Graph clustering using distance-k cliques. In *GD '99 : Proceedings of the 7th International Symposium on Graph Drawing* (1999), Springer-Verlag, pp. 98–106.

[26] EILAM, T., GAVOILLE, C., AND PELEG, D. Compact routing schemes with low stretch factor (extended abstract). In *PODC '98 : Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1998), ACM Press, pp. 11–20. `http://doi.acm.org/10.1145/277697.277702`.

[27] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics : Principles and Practice in C, 2nd Edition.* Addison-Wesley, 1995.

[28] FOOTE, B., AND YODER, J. W. Big ball of mud. In *Pattern Languages of Program Design*, N. Harrison, B. Foote, and H. Rohnert, Eds., vol. 4. Addison Wesley, 2000, pp. 654–692. `http://citeseer.ist.psu.edu/article/foote97big.html`.

[29] FREE SOFTWARE FOUNDATION. Gnu general public license (gnu gpl). `http://www.gnu.org/licenses/licenses.html#TOCGPL`.

[30] GARAGE GAMES. Torque game engine. http://www.garagegames.com/mg/projects/tge/.

[31] GERSHON, N., EICK, S. G., AND CARD, S. Information visualization. *interactions 5*, 2 (1998), 9–15. http://doi.acm.org/10.1145/274430.274432.

[32] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1997), ACM Press, pp. 108–124. http://doi.acm.org/10.1145/263698.264352.

[33] HAYNES, T., HEDETNIEMI, S., AND SLATER, P., Eds. *Domination in Graphs : Advanced Topics.* Marcel Dekker, 1998.

[34] HAYNES, T., HEDETNIEMI, S., AND SLATER, P., Eds. *Domination in Graphs : The Theory.* Marcel Dekker, 1998.

[35] HENNING, M. A. *Distance domination in graphs.* Marcel Dekker, 1998, pp. 321–349.

[36] HERMAN, MELANÇON, G., AND MARSHALL, M. S. Graph visualization and navigation in information visualization : A survey. *IEEE Transactions on Visualization and Computer Graphics 6*, 1 (/2000), 24–43. http://citeseer.nj.nec.com/herman00graph.html.

[37] IEEE COMPUTER SOCIETY : STANDARDS COORDINATING COMMITTEE. *610.12-1990 : IEEE Standard Glossary of Software Engineering Terminology.* Institute of Electrical and Electronics Engineers, 1990.

[38] IVKOVIC, I., AND GODFREY, M. Architecture recovery of dynamically linked applications : A case study. In *IWPC* (2002). http://citeseer.nj.nec.com/ivkovic02architecture.html.

[39] JR., W. H. H., AND BUCK, J. A. *Engineering Electromagnetics (6th Edition).* McGraw-Hill, 2001.

[40] KANEHISA, M. A database for post-genome analysis. *Trends Genet.*, 13 (1997), 375–376. http://www.ncbi.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=9287494&dopt=Abstract.

[41] KANEHISA, M., AND GOTO, S. Kegg : Kyoto encyclopedia of genes and genomes. *Nucleic Acids Res. 28* (2000), 27–30. http://www.ncbi.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=10592173&dopt=Abstract.

[42] KAO, M.-Y. Tree contractions and evolutionary trees. *SIAM J. Comput. 27*, 6 (1998), 1592–1616. http://dx.doi.org/10.1137/S0097539795283504.

[43] KAZMAN, R., AND CARRIèRE, S. J. Playing detective : Reconstructing software architecture from available evidence. *Automated Software Engg. 6*, 2 (1999), 107–138. http://dx.doi.org/10.1023/A:1008781513258.

[44] KING, V., ZHANG, L., AND ZHOU, Y. On the complexity of distance-based evolutionary tree reconstruction. In *SODA '03 : Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (2003), Society for Industrial and Applied Mathematics, pp. 444–453.

[45] KLÖSH, R. R. Reverse engineering : Why and how to reverse engineer software. In *Proceedings of the California Software Symposium (CCS '96)* (1996), U. of Southern California, Ed., pp. 92–99.

[46] KUHN, F., AND WATTENHOFER, R. Constant-time distributed dominating set approximation. In *PODC '03 : Proceedings of the twenty-second annual symposium on Principles of distributed computing* (New York, NY, USA, 2003), ACM Press, pp. 25–32. http://doi.acm.org/10.1145/872035.872040.

[47] LABELLE, J. *Théorie des Graphes.* Modulo, 1981.

[48] LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of IEEE 68*, 9 (1980), 1960–1976.

[49] LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. Characteristics of application software maintenance. *Commun. ACM 21*, 6 (1978), 466–471. http://doi.acm.org/10.1145/359511.359522.

[50] MANCORIDIS, S., MITCHELL, B. S., CHEN, Y.-F., AND GANSNER, E. R. Bunch : A clustering tool for the recovery and maintenance of software system structures. In *ICSM* (1999), pp. 50–. http://citeseer.nj.nec.com/mancoridis99bunch.html.

[51] MANCORIDIS, S., MITCHELL, B. S., RORRES, C., CHEN, Y., AND GANSNER, E. R. Using automatic clustering to produce high-level system organizations of source code. In *IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98)* (Piscataway, NY, 1998), IEEE Press. http://citeseer.nj.nec.com/mancoridis98using.html.

[52] MARCHIONINI, G. *Information Seeking in Electronic Environments.* Cambridge University Press, Cambridge, UK, 1995.

[53] MCKAY, B. D. Nauty. http://cs.anu.edu.au/~bdm/nauty/.

[54] MISUE, K., EADES, P., LAI, W., AND SUGIYAMA, K. Layout adjustment and the mental map. *Journal of Visual Languages and Computing 2*, 6 (/1995), 183–210.

[55] MITCHELL, B. S., AND MANCORIDIS, S. Clustering module dependency graphs of software systems using the bunch tool. http://citeseer.nj.nec.com/410489.html.

[56] MITCHELL, B. S., AND MANCORIDIS, S. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *ICSM* (2001), pp. 744–753. http://citeseer.ist.psu.edu/mitchell01comparing.html.

[57] MULLER, H. A., JAHNKE, J. H., SMITH, D. B., STOREY, M.-A. D., TILLEY, S. R., AND WONG, K. Reverse engineering : a roadmap. In *ICSE — Future of SE Track* (2000), pp. 47–60. http://citeseer.nj.nec.com/muller00reverse.html.

[58] MÜLLER, H. A., ORGUN, M. A., TILLEY, S. R., AND UHL, J. S. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance : Research and Practice 5(4)* (December 1993), 181–204. http://citeseer.ist.psu.edu/uller93reverse.html.

[59] NELSON, M. A survey of reverse engineering and program comprehension. http://citeseer.ist.psu.edu/nelson96survey.html.

[60] NORTH, S. C., AND KOUTSOFIOS, E. Application of graph visualization. In *Proceedings of Graphics Interface '94* (Banff, Alberta, Canada, 1994), pp. 235–245. http://citeseer.ist.psu.edu/221206.html.

[61] OBJECT MANAGEMENT GROUP. *OMG Unified Modeling Language Specification Version 1.5.* Object Management Group, 2003. http://www.omg.org/docs/formal/03-03-01.pdf.

[62] PELEG, D., AND UPFAL, E. A trade-off between space and efficiency for routing tables. *J. ACM 36*, 3 (1989), 510–530. http://doi.acm.org/10.1145/65950.65953.

[63] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C : The Art of Scientific Computing.* Cambridge University Press, 1992.

[64] PRESSMAN, R. S. *Software Engineering : A Practitioner's Approach (4th Edition).* McGraw-Hill, 1997.

[65] QUAN, D., HUYNH, D., KARGER, D. R., AND MILLER, R. User interface continuations. In *UIST '03 : Proceedings of the 16th annual ACM symposium on User interface software and technology* (New York, NY, USA, 2003), ACM Press, pp. 145–148. http://doi.acm.org/10.1145/964696.964712.

[66] ROBERT, P. *Le Nouveau Petit Robert.* Dictionnaires Le Robert, 2004. Étendu et révisé par Josette Rey-Debove et Alain Rey.

[67] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[68] SEBESTA, R. W. *Concepts of programming langages : Sixth Edition.* Addison-Wesley, 2004.

[69] SHNEIDERMAN, B. Direct manipulation : A step beyond programming languages. *IEEE Computer 16*, 8 (1983), 57–69.

[70] SHNEIDERMAN, B. The eyes have it : A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL '96)* (1996), IEEE.

[71] SHNEIDERMAN, B. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Intelligent User Interfaces* (1997), pp. 33–39. http://citeseer.ist.psu.edu/shneiderman97direct.html.

[72] SIPSER, M. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[73] SLAVÍK, P. A tight analysis of the greedy algorithm for set cover. In *STOC '96 : Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (New York, NY, USA, 1996), ACM Press, pp. 435–441. http://doi.acm.org/10.1145/237814.237991.

[74] TERROUX, E. Études des valuations dans le contexte de la visualisation de graphes. Mémoire de maîtrise, Université du Québec à Montréal, 2003.

[75] VAN HEESCH, D. Doxygen. www.doxygen.org.

[76] WAH, B. W., AND RAMAMOORTHY, C. V. *Handbook of Software Engineering.* Electrical/Computer science and Engineering Series. Van Nostrad Reinhold Company, 1984, ch. Theory of Algorithms and Computation Complexity with Applications to Software Design.

[77] WARE, C., AND FRANCK, G. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics 15*, 2 (1996), 121–140. http://citeseer.nj.nec.com/ware96evaluating.html.

[78] WIEGAND, S., STCHERBATCHENK, A., MADDOCK, D. J., AND HENDRICKS, K. B. TEXniccenter. http://www.toolscenter.org/.

[79] ZIMMER, C., AND GOULD, S. J. *Evolution : The Triumph of an Idea.* HarperCollins, 1991.

[80] ZWICK, U. Exact and approximate distances in graphs — A survey. *Lecture Notes in Computer Science 2161* (2001), 33–?? http://citeseer.ist.psu.edu/zwick01exact.html.