République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

ESI ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

Mémoire

Pour Obtention du diplôme d'ingénieur En Informatique

Option : Système Informatique (SIQ)

Algorithmes de compression de graphes par extraction de motifs et k2-trees : Étude et implémentation

---

Réalisé par :

Mlle. Hafsa Bousbiat
eh_bousbiat@esi.dz
ESI

Mlle. Sana Ihadadene
es_ihadadene@esi.dz
ESI

Encadré par :

Dr. Karima Amrouche
k_amrouche@esi.dz
ESI

Dr. Hamida Seba
hamida.seba@univ-lyon1.fr
Université de Lyon

Octobre 2018
Année Universitaire : 2018-2019

Remerciement

La construction de ce mémoire n'aurait été possible sans l'intervention de plusieurs personnes. Qu'elles trouvent ici l'expression de nos plus sincères remerciements pour leurs précieux conseils.

Nous tenons d'abord à exprimer notre gratitude à l'égard de nos deux encadrantes, Madame Seba Hamida, professeur à l'Université Claude Bernard de Lyon et Madame Amrouche Karima, enseignante à l'École Nationale Supérieure d'Informatique, pour leur précieuse aide, leurs judicieux conseils et pour le temps qu'elles nous ont consacré tout au long de ce travail.

Nous remercions également tout le corps professoral et administratif de l'équipe pédagogique de l'École Nationale Supérieure d'Informatique, et à leur tête nos enseignants, pour la richesse et la qualité de la formation qu'ils nous ont offerte tout au long de notre cursus.

Nous tenons aussi à remercier les membres du jury qui nous ont fait l'honneur d'accepter de juger cet humble travail.

Enfin, nous adressons nos plus sincères remerciements à tous nos proches et amis, qui nous ont toujours encouragés au cours de la réalisation de ce mémoire.

Merci à tous et à toutes ...

Résumé

Nous vivons dans un monde où la quantité d'informations ne cesse d'augmenter et dont la bonne gestion implique l'utilisation des graphes qui se sont répandus dans différents domaines allant des réseaux sociaux et de communication jusqu'aux domaines de la chimie et de la biologie. Cette abondance de données générées fait appel à une technique aussi vieille que la discipline de traitement de données mais qui connait de nouveaux défis aujourd'hui : la compression. La compression de graphes est un domaine dans lequel le graphe initial subit des transformations pour en obtenir une version plus réduite et compacte permettant, dans la majorité des cas, d'effectuer les traitements dans un temps nettement meilleur.

Deux classes de méthodes de compression feront l'objet de notre étude : les méthodes de compression par extraction de motifs et les méthodes basées sur les k2-trees. De ce fait, nous proposons deux moteurs de compression chacun englobant une ou plusieurs méthodes de chaque classe. Le premier moteur permet de compresser un graphe à travers ses structures les plus denses. Tant dis que le deuxième moteur exploite les propriétés de la matrice d'adjacence pour obtenir une représentation compacte. Nous proposons aussi une méthode de compression destinée aux graphes dynamiques et qui se situe à l'intersection des deux classes étudiées. En effet, elle permet de compresser le graphe à travers ces structures les plus denses tout en gardant trace de l'erreur, généralement représentée sous forme de matrices creuses, dans une structure k2-trees.

Nous conclurons par une étude comparative des performances des différents algorithmes de compression existants et la méthode que nous proposons où nous nous sommes basées sur des métriques d'évaluation tels que : le taux de compression et le gain obtenu. Afin de pouvoir réaliser cette étude en toute objectivité, les méthodes de compression seront testées sur des benchmarks connus de graphes issus de divers domaines.

Mots Clés : *Compression de graphes, Big Data, Extraction de motifs, K2-trees, Graphe du Web.*

Abstract

We live in a world where the amount of data is constantly increasing and whose good management involves the use of graphs that have spread in different fields from social and communication networks to the fields of chemistry and biology. This abundance of generated data calls for a technique that is as old as the discipline of data processing but which is facing new challenges today : compression. Graph compression is a field in which the initial graph undergoes transformations in order to obtain a smaller and more compact version allowing, in the majority of cases, to perform the processings in a much better time.

Two classes of compression methods will be the focus of our study : pattern extraction compression methods and k2-tree based methods. As a result, we propose two compression engines each encompassing one or more methods of each class. The first engine makes compress a graph through its densest substructures. While, the second engine exploits the properties of the adjacency matrix to obtain a compact representation. We also propose a novel compression method for dynamic graphs at the intersection of the two classes studied. Indeed, it allows to compress the graph through his most important substructures, while keeping track of the error, usually represented as sparse matrices, in a k2-trees structure.

We will conclude with a comparative study of the performances of the different existing compression algorithms and the method we propose where we use based on evaluation the folowing metrics : the compression ratio, number of bits per vertex and processing time. In order to be able to carry out this study in all objectivity, the compression methods will be tested on known benchmarks of graphs coming from various domains.

Key words : *Graph compression, Big Data, Pattern extraction, K2-trees, Web graph.*

ملخص

إننا نعيش في وقتنا الحالي واقعا يعرف تزايد في الكم المعرفي و المعلوماتي مما اقتضى و أوجب استعمال المنحنيات التي اصبحت واسعة الانتشار و اكتسحت عدة ميادين مختلفة انطلاقا من مواقع التواصل الاجتماعي و الاتصالات وصولا لميادين الكمياء و البيولوجيا . إن هذه الكمية الهائلة من المعلومات تلزم الرجوع إلى تقنية قديمة قدم مجال معالجة البيانات و التي تواجه تحديات جديدة : الضغط … ضغط المنحنيات هو مجال يخضع فيه الرسم البياني الأولي لتحولات من اجل الحصول على نسخة أصغر تسمح في غالبية الحالات بتحسين الوقت اللازم لمعالجة و تحليل البيانات.

سنركز في دراستنا على فئتان من أساليب الضغط : طرق ضغط عن طريق استخراج الأنماط والأساليب المعتمدة على الأشجار. نتيجة لذلك ، نقترح محركي ضغط يشمل كل منهما طريقة أو أكثر من كل فئة. المحرك الأول يجعل من الممكن ضغط الرسم البياني من خلال هياكله الأكثر كثافة والأكثر أهمية . في حين أن المحرك الثاني يستغل خصائص مصفوفة الجوار للحصول على تمثيل مضغوط . نقترح أيضاًا طريقة جديدة لضغط الرسوم البيانية الديناميكية تقع عند تقاطع الفئتين المدروستين. في الواقع، فإنه يجعل من الممكن ضغط الرسم البياني من خلال هذه الهياكل الأكثر كثافة مع تتبع الخطأ ، الذي عادة ما يتم عرضه كمصفوفات مجوفة ، في بنية الأشجار .

سوف نختتم بدراسة مقارنة لأداء خوارزميات الضغط المختلفة والخوارزمية التي نقترحها حيث نستخدم تقييم مقاييس المتابعة: نسبة الضغط والحجم اللازم لتخزين كل معلومة بالاضافة الى وقت المعالجة. من أجل أن نكون قادرين على إجراء هذه الدراسة بموضوعية ، سيتم اختبار طرق الضغط على معايير معروفة من الرسوم البيانية مستخرجة من وضعيات حقيقية و من مجالات مختلفة.

كلمات مفتاحية: ضغط الرسم البياني ، البيانات الكبيرة ، استخراج الأنماط ، أشجار ، الرسم البياني على الويب.

Table des matières

Table des figures

Liste des tableaux

Acronymes

k^2-GraCE k^2-trees Graph Compression engine. 1, 50, 54

BFS Breadth First Search. 36, 52, 59, 86

CCG Composant Connecté Géant. 61

CONDENSE CONditional Diversified Network Summarization. 38

DAC Directly Addressable Codes. 22

DDSM Dynamic Dense Subgraph Mining. 1, 68

DFS Deapth First Search. 22, 52, 86

DSM Dense SubGraph Mining. 45, 68

ECWG Efficient Compression of web graph. 39

GCUPMT Graph Compression Using Pattern Matching. 39

GNF Greedy'nForget. 66, 92

MDL Minimum Description Length. 17, 34, 38, 39, 65

OLAP Online Analytical Processing. 26

P-GraCE Pattern Graph Compression engine. VIII, 1, 57, 58

RDF Resource Description Framework. 16, 17, 20

SNAP Stanford Network Analysis Package. 74, 77, 99, 101

VNM Virtual Node Miner. 42, 43, 45

VOG Vocabulary Graph. 34–37

Introduction générale

Avec l'énorme quantité de données produites par les activités humaines de nos jours, le problème de données massives (Big data) est devenu un enjeu essentiel. Un des outils les plus efficaces pour structurer et manipuler ces données est l'utilisation des graphes. Les graphes sont des outils de modélisation utilisés dans beaucoup de domaines pour la représentation des données : réseaux sociaux et de communication (entités reliées entre elles par des liens physiques ou communautaires), chimie (relations entre les atomes), biologie (interactions entre protéines par exemple) et bien d'autres domaines.

Face à cette infobésité, les algorithmes classiques de traitement et de gestion des données se montrent incapables d'offrir des réponses dans un temps raisonnable. Plusieurs solutions ont été pensées pour contrer ce volume de données. Une des solutions les plus anciennes mais qui connait de nouveaux défis de nos jours est la *compression de données*.

Le domaine de compression de données est une branche de la théorie de l'information qui s'intéresse à minimiser la taille des données à stocker, traiter et transmettre améliorant ainsi de façon directe les temps de traitement. Parallèlement à cela, nous trouvons la compression des graphes qui est un domaine dans lequel le graphe initial subit des transformations pour en obtenir une version plus réduite. Différentes techniques, basées sur différentes approches, permettent cette compression, avec ou sans perte d'information, et génèrent de nouveaux graphes sur lesquels il est beaucoup plus intéressant d'effectuer les différents traitements.

Cependant, deux types de méthodes de compression de graphes se sont distinguées parmi tout les autres types de méthodes : les méthodes de compression en utilisant les arbres k2-trees et les méthodes de compression par extraction de motifs. En effet, elles permettent de trouver dans la majorité des cas un bon compromis entre l'espace mémoire et les temps de traitement. Ces deux classes de méthodes feront l'objet de notre étude.

Notre première contribution portera sur la conception, l'implémentation et l'évaluation de deux moteurs de compression, k^2-trees Graph Compression engine (k^2-GraCE) et P-GraCE, chacun englobant les méthodes relatives à une classe. Nous visons à travers cela à comparer entre les performances des méthodes s'intégrant dans ces deux classes. Notre deuxième contribution consiste en la proposition d'une nouvelle méthode de compression pour les graphes dynamiques, s'intitulant Dynamic Dense Subgraph Mining (DDSM).

Nous avons hiérarchisé notre mémoire en cinq grands chapitres. Le premier est une introduction au domaine de la théorie des graphes. Dans le second chapitre, nous introduisons les définitions de base du domaine de compression de données appliqué aux graphes ainsi que les différentes méthodes de compression existantes sous forme d'une classification que nous propo-

sons. Par la suite, dans le troisième chapitre , nous détaillons la conception de nos deux moteurs de compression de graphes et nous décrirons les bases théoriques de notre méthode, puis nous donnons dans le quatrième chapitre les détails de l'implémentation que nous proposons. Finalement, nous présentons dans le dernier chapitre les différents tests de performance et les résultats obtenus.

Première partie

Synthèse bibliographique

Chapitre 1

Théorie des graphes

Pour faciliter la compréhension d'un problème, nous avons tendance à le dessiner ce qui nous amène parfois même à le résoudre. La théorie des graphes est fondée à l'origine sur ce principe. Historiquement, elle représente un domaine mathématique qui s'est développé au sein d'autres disciplines comme la chimie, la biologie, ... Elle constitue aujourd'hui un corpus de connaissances très important et un instrument efficace pour résoudre une multitude de problèmes.

Dans ce chapitre, nous présenterons les notions et les concepts clés relatifs aux graphes , à savoir : la définition d'un graphe, ses types et sa représentation structurelle. Nous clôturons le chapitre avec quelques domaines d'application des graphes.

1.1 Graphe non orienté

1.1.1 Définitions et généralités

Un graphe non orienté G est la donnée d'un couple (V , E) où V = $\{v_1, v_2, ..., v_n\}$ est un ensemble fini dont les éléments sont appelés sommets ou nœuds (Vertices en anglais) et E=$\{e_1, e_2, \ldots, e_m\}$ est un ensemble fini d'arêtes (Edges en anglais). Toute arête e de E correspond à un couple non ordonné de sommets (v_i, v_j) $\in$ E $\subset V \times V$ représentant ses extrémités (Müller, 2012) (Fages, 2014).
Soient $e = (v_i, v_j)$ et e'=(v_k, v_l) deux arêtes de E, On dit que :

- v_i et v_j sont les extrémités de e et e est incidente à v_i et v_j (Hennecart et al., 2012).
- v_i et v_j sont voisins ou adjacents, s'il y a au moins une arête entre eux dans E (IUT, 2012).
- L'ensemble des sommets adjacents aux deux extrémités de e est appelé le voisinage de e (Müller, 2012).
- e et e' sont voisins s'ils ont une extrémité commune (Lopez, 2003).
- L'arête e est une boucle si ses extrémités coïncident, i.e, $v_i = v_j$ (IUT, 2012).
- L'arête e est multiple si elle a plus d'une seule occurrence dans l'ensemble E.

1.1.2 Représentation graphique

Un graphe non orienté G peut être représenté par un dessin sur un plan comme suit (Müller, 2012) :

— Les nœuds $v_i \in$ V de G sont représentés par des points distincts.

— Les arêtes $e = (v_i, v_j) \in$ E de G sont représentés par des lignes, pas forcement rectilignes, qui relient les extrémités de chaque arête e.

Exemple : Soit g=(V1 , E1) un graphe non orienté tel que : V1={ 1,2,3,4,5 } et E1={(1,2), (1,4), (2,2), (2,3), (2,5), (3,4)}. La représentation graphique de g est alors donnée par le schéma de la figure 1.1.

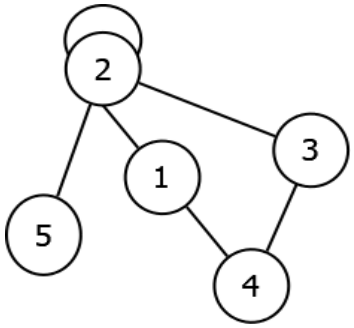


FIGURE 1.1 – Exemple de représentation graphique d'un graphe non orienté

1.1.3 Propriétés d'un graphe

- **Ordre d'un graphe :** On appelle ordre d'un graphe le nombre de ses sommets, i.e, Card(V) (Roux, 2014).
- **Taille d'un graphe :** On appelle taille d'un graphe le nombre de ses arêtes, i.e, Card(E) (Roux, 2014).
- **Degré dans un graphe :**
  - **Degré d'un sommet :** Le degré d'un sommet noté $d(v_i)$ est le nombre d'arêtes incidentes à ce sommet, sachant qu'une boucle compte pour deux (Müller, 2012). Dans l'exemple de la figure 1.1, le degré du sommet (1) est : $d(1)=2$.
  - **Degré d'un graphe :** Le degré d'un graphe est le degré maximum de ses sommets, i.e, $\max(d(v_i))$ (Müller, 2012). Dans l'exemple de la figure 1.1, le degré du graphe g est $d(2)=5$.
- **Rayon et diamètre dans un graphe :**
  - **Distance :** La distance entre deux sommets v et u est le plus petit nombre d'arêtes qu'on doit parcourir pour aller de v à u ou de u à v (Müller, 2012).
  - **Diamètre d'un graphe :** C'est la plus grande distance entre deux sommets de ce graphe (Müller, 2012).

- **Rayon d'un graphe :** C'est la plus petite distance entre deux sommets de ce graphe (Parlebas, 1972).

1.2 Graphe orienté

1.2.1 Définitions et généralités

Un graphe orienté G est la donnée d'un couple (V , E) où V est un ensemble fini dont les éléments sont appelés les sommets de G et E ⊂ V x V est un ensemble de couples ordonnés de sommets dits arcs (Müller, 2012). G est appelé dans ce cas digraphe (directed graph).
Pour tout arc e = (v_i , v_j) ∈ E :

— v_i est dit extrémité initiale ou origine de e et v_j est l'extrémité finale de e (Müller, 2012).

— v_i est le prédécesseur de v_j et v_j est le successeur de v_i (IUT, 2012).

— les sommets v_i , v_j sont des sommets adjacents (Jean-Charles Régin, 2016).

— e est dit sortant en v_i et incident en v_j (Jean-Charles Régin, 2016).

— e est appelé boucle si $v_i = v_j$, i.e, l'extrémité initiale et finale sont identiques (IUT, 2012).

1.2.2 Représentation graphique

Un graphe G = (V , E) peut être projeté sur le plan en représentant :

— Dans un premier temps les nœuds v_i ∈ V par des points disjoints du plan.

— Et dans un second temps les arcs e = (v_i , v_j) ∈ E par des lignes orientées reliant par des flèches les deux extrémités de e.

Exemple :
Soit g = (V_1 , E_1) un digraphe tel que : V_1 = { 1,2,3,4 } et E_1 = {(1,2),(1,3),(3,2),(3,4),(4,3)}.
La représentation graphique de g est alors donnée par le schéma de la figure 1.2.

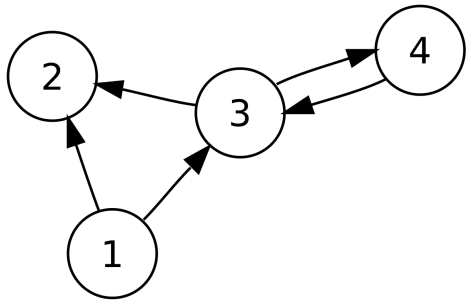


FIGURE 1.2 – Exemple de représentation graphique d'un digraphe.

1.2.3 Quelques Propriétés :

- **Ordre d'un digraphe :** est le nombre de sommets n = Card(V) (Roux, 2014).

- **taille d'un digraphe :** est le nombre d'arcs m = Card(A) (Roux, 2014).
- **Degré dans un digraphe :** Le degré d'un sommet $v_i \in$ V dans un digraphe G=(V,E) est donné par la formule :

$$\mathrm{d}(v_i) = d^+(v_i) + d^-(v_i)$$

où $d^+(v_i)$ est le nombre d'arcs sortants du sommet v_i et est appelé degré extérieur et $d^-(v_i)$ représente le nombre d'arcs incidents et est appelé degré intérieur (Müller, 2012).

- **Voisinage dans un digraphe :** Le voisinage d'un sommet $v_i \in$ V, noté V(v_i), dans un digraphe G = (V , E) est :

$$\mathrm{V}(v_i) = \mathrm{succ}(v_i) \bigcup \mathrm{pred}(v_i),$$

où succ(v_i) est l'ensemble des successeurs de v_i et pred(v_i) est l'ensemble de ses prédécesseurs (Rigo, 2010), i.e, le voisinage de v_i est l'ensemble des sommets qui lui sont adjacents.

1.3 Notion de Connexité

Les graphes sont généralement exploitables à travers leur interrogation qui permet de fournir des réponses aux problèmes modélisés. L'une des informations les plus importantes dans un graphe est la notion de relations (directes ou indirectes) entre deux nœuds ou plus formellement *la connexité* dans un graphe. Dans cette partie, nous allons définir les concepts relatifs à cette notion dans le cas d'un graphe non orienté (resp. orienté).

- **Chaine (resp. Chemin) :** est une liste de sommets S= ($v_0, v_1, v_2, ..., v_k$) tel qu'il existe une arête (resp. un arc) entre chaque couple de sommets successifs (Müller, 2012).
- **Cycle (resp. Circuit) :** est une chaine (resp. chemin) dont le premier et le dernier sommet sont identiques (Roux, 2014).
- **Graphe connexe :** Un graphe non orienté (resp. orienté) est dit connexe (resp. fortement connexe) si pour toute paire de sommets (v_i, v_j), il existe une chaine (resp. chemin) S les reliant (Müller, 2012).

1.4 Quelques types de graphes

Avec les avancées technologiques au fil du temps, plusieurs types de graphes ont vu le jour. En effet, la complexité et la variété des problèmes scientifiques existants modélisés par ces derniers ont poussé les chercheurs à adapter leur structure selon le problème auquel ils font face. Durant cette section, nous allons définir les principaux types existants.

- **Graphe Complet :** Un graphe G = (V , E) est un graphe complet si tous les sommets $v_i \in$ V sont adjacents (Jean-Charles Régin, 2016). Il est souvent noté K_n où n = card(V) (Roux, 2014).

- **Graphe étiqueté et graphe pondéré :** Un graphe étiqueté G = (V , E , W) est un graphe non orienté (resp. orienté) dont chacune des arêtes (resp. arcs) $e_i \in$ E est doté d'une étiquette w_i. Si de plus, w_i est un nombre alors G est dit graphe pondéré (valué) (Roux, 2014).
- **Graphe simple et graphe multiple :** Un graphe G = (V , E) non orienté (resp. orienté) est dit simple s'il ne contient pas de boucles et toute paire de sommets sont reliés par au plus une arête (resp. un arcs). Dans le cas contraire, G est dit multiple (IUT, 2012).

1.5 Graphe partiel et sous-graphe

La quantité de données disponible aujourd'hui et sa croissance de manière exponentielle ont favorisé la décomposition des graphes en des entités plus petites afin de garantir une facilité de compréhension et d'analyse dans le but d'extraire l'information la plus pertinente. Dans cette partie, nous allons définir de manière plus formelle ce que ces entités sont, ainsi que leurs types.

1.5.1 Définitions :

Soient G = (V , E), $G' = (V', E')$ et $G'' = (V'', E'')$ trois graphes.

- Le graphe G' est appelé **graphe partiel** de G si : $V' = V$ *et* $E' \subset$ E (Roux, 2014). En d'autres termes, un graphe partiel est obtenu en supprimant une ou plusieurs arêtes de G.
- Le graphe G'' est dit **sous-graphe** de G si : $V'' \subset V$ et $E'' \subset E \cap (V''xV'')$ (Rigo, 2010), i.e, un sous-graphe est obtenu en enlevant un ou plusieurs nœuds du graphe initial ainsi que les arêtes dont ils représentent l'une des deux extrémités.

1.5.2 Quelques Types de sous graphes :

- **Une Clique :** est un sous-graphe complet de G (Rigo, 2010).
- **Biparti :** G' est un sous-graphe biparti si il existe une partition de V' en deux sous ensembles notés V_1 et V_2, i.e V' = $V_1 \cup V_2$ et $V_1 \cap V_2 = \phi$, tel que E' = V_1 x V_2 (Rigo, 2010).
- **Étoile :** est un cas particulier de sous-graphe biparti où V_1 est un ensemble contenant le sommet central (dit *hub*) uniquement et V_2 contient le reste des nœuds (dits *spokes*)(Koutra et al., 2015) .

1.6 Représentation Structurelle d'un graphe

Bien que la représentation graphique soit un moyen pratique pour définir un graphe, elle n'est clairement pas adaptée ni au stockage du graphe dans une mémoire, ni à son traitement. Pour cela, plusieurs structures de données ont été utilisées pour représenter un graphe, ces structures varient selon l'usage du graphe et la nature des traitements appliqués. Nous allons présenter dans cette partie les structures les plus utilisées.

Soit un graphe G(V , E) d'ordre n et de taille m dont les sommets v_1, v_2,..., v_n et les arêtes (ou arcs) e_1, e_2,..., e_m sont ordonnés de 1 à n et de 1 à m respectivement.

1.6.1 Matrice d'adjacence

La matrice d'adjacence de G est une matrice booléenne carrée d'ordre n : $(m_{ij})_{(i,j)\in[0;n]^2}$, dont les lignes (i) et les colonnes (j) représentent les identifiants des sommets de G. Les entrées (ij) prennent une valeur de "1" s'il existe un arc (une arête dans le cas d'un graphe non orienté) allant du sommet i au sommet j et un "0" sinon, i.e, (Lehman et al., 2010) (SABLIK, 2018) (IUT, 2012) :

$$m_{ij} := \begin{cases} 1 & si \quad (v_i, v_j) \in E \\ 0 & sinon \end{cases}$$

Dans le cas d'un graphe non orienté, la matrice est symétrique par rapport à la première diagonale, i.e, $m_{ij} = m_{ji}$. Dans ce cas le, graphe peut être représenter avec la composante triangulaire supérieure de la matrice d'adjacence (Müller, 2012).

Note :

- Cette représentation est valide pour le cas d'un graphe non orienté et orienté.
- Dans le cas d'un graphe pondéré, les "1" sont remplacés par les poids des arêtes (ou arcs) (Lopez, 2003).
- Ce mode de représentation engendre des matrices très creuses (comprenant beaucoup de zero) (Hennecart et al., 2012).

Exemple : La figure 1.4 représente un exemple de matrice d'adjacence pour le graphe G ci-contre (figure 1.3) :

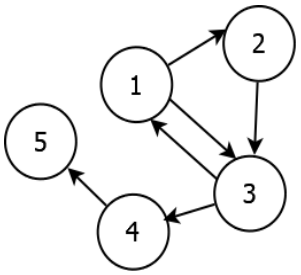


FIGURE 1.3 – Graphe orienté G

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

FIGURE 1.4 – Matrice d'adjacence du graphe G

Place occupée en mémoire : n^2 pour un graphe d'ordre $|n|$ (Lopez, 2003).

1.6.2 Matrice d'incidence

La matrice d'incidence d'un graphe orienté G est une matrice de taille $n \times m$ dont les lignes représentent les identifiants des sommets (i ∈ V) et les colonnes représentent les identifiants des

arcs (j ∈ E) et dont les coefficients (m_{ij}) sont dans {-1, 0, 1}, tel que (Hennecart et al., 2012) (SABLIK, 2018) :

$$m_{ij} := \begin{cases} 1 & si \text{ le sommet i est l'extrémité final de l'arc j} \\ -1 & si \text{ le sommets i est l'extrémité initial de l'arc j} \\ 0 & sinon \end{cases}$$

Pour un graphe non orienté, les coefficients (m_{ij}) de la matrice sont dans {0, 1}, tel que (Hennecart et al., 2012) :

$$m_{ij} := \begin{cases} 1 & si \text{ le sommet i est une extrémité de l'arête j} \\ 0 & sinon \end{cases}$$

Exemple : La figure 1.6 représente un exemple d'une matrice d'incidence pour le graphe G ci-contre (figure 1.5) :

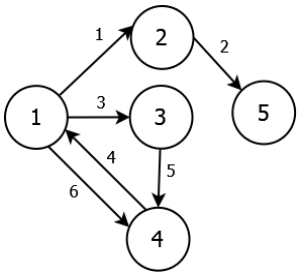


FIGURE 1.5 – Graphe orienté G

$$M = \begin{pmatrix} -1 & 0 & -1 & 1 & 0 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

FIGURE 1.6 – Matrice d'incidence du graphe G

Place occupée en mémoire : $n \times m$

1.6.3 Liste d'adjacence

La liste d'adjacence d'un graphe G est un tableau de |*n*| listes, où chaque entrée (i) du tableau correspond à un sommet et comporte la liste T[i] des successeurs (ou prédécesseurs) de ce sommet, c'est à dire tous les sommets j tel que (i,j) ∈ E (SABLIK, 2018).
Dans le cas d'un graphe non orienté, on aura : j ∈ la liste T[i] $\iff$ i ∈ la liste T[j] (IUT, 2012).

Exemple : La figure 1.8 représente un exemple d'une liste d'adjacence pour le graphe G ci-contre (figure 1.7) :

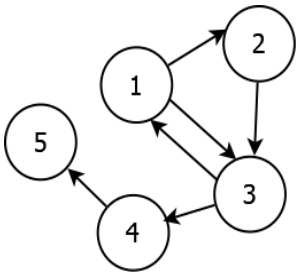


FIGURE 1.7 – Graphe orienté G

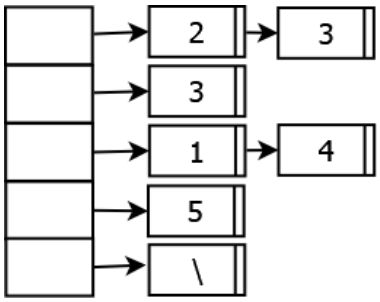


FIGURE 1.8 – Liste d'adjacence du graphe G

Place occupée en mémoire : Dans le cas d'un graphe orienté, l'espace occupé par le graphe est de n + m. Dans le cas d'un graphe non orienté, l'espace est de n + 2m car une arête est représentée deux fois.

1.7 Les domaines d'application

La diversité des domaines faisant appel à la modélisation par des graphes ne cesse d'augmenter, allant des réseaux sociaux aux réseaux électriques et réseaux biologiques et arrivant jusqu'aux World Wide Web. Dans cette partie, nous allons décrire trois domaines d'application les plus répandus des graphes.

1.7.1 Graphes des réseaux sociaux :

Les réseaux sociaux représentent un lieu d'échange et de rencontre entre individus (entités) et dont l'utilisation est devenue de nos jours une nécessité. Pour représenter les interactions entre ces individus, nous avons généralement besoin d'avoir recours aux graphes où les sommets sont des individus ou des entités et les interactions entre eux sont représentées par des liens. Vue la diversité des interactions sociales, la modélisation de ces réseaux nécessite différents types de graphes : graphes non orientés pour les réseaux sociaux avec des relations symétriques, graphes orientés pour représenter des relations non symétriques comme c'est la cas dans les réseaux de confiance, graphes pondérés pour les réseaux sociaux qui contiennent différents niveaux d'intensité dans les relations, ..., etc. (Lemmouchi, 2012)

1.7.2 Graphes en Bioinformatique :

La bio-informatique est un domaine qui se trouve à l'intersection de deux grands domaines celui de l'informatique et celui de la biologie. Elle a pour but d'exploiter la puissance de calcul des équipements informatiques pour effectuer des traitements sur des données moléculaires massives (Pellegrini et al., 2004).

Elle est largement utilisée dans l'analyse des séquences d'ADN et des protéines à travers leur modélisation sous forme de graphe. A titre d'exemple, les graphes non orientés multiples sont un outil de modélisation des réseaux d'interaction protéine-protéine (Pellegrini et al., 2004), le but dans ce cas est l'étude du comportement d'un protéine par rapport à un autre.

1.7.3 Le Graphe du web :

Le graphe du Web est un graphe orienté dont les sommets sont les pages du web et les arcs modélisent l'existence d'un lien hypertexte dans une page vers une autre (Brisaboa et al., 2009). Il représente l'un des graphes les plus volumineux : en juillet 2000 déja, on estimait qu'il contenait environ 2,1 milliards de sommets et 15 milliards d'arêtes avec 7,3 millions de pages ajoutées chaque jour (Guillaume and Latapy, 2002). De ce fait, ce graphe a toujours attiré l'attention des chercheurs. En effet, l'étude de ses caractéristiques a donné naissance à plusieurs algorithmes intéressants, notamment l'algorithme PageRank de classement des pages web qui se trouve derrière le moteur de recherche le plus connu de nos jours : Google.

1.8 Conclusion

Dans ce chapitre nous avons présenté les notions et les concepts généraux qui touchent à la théorie des graphes : définitions des graphes, leurs principales propriétés, leurs représentations ainsi que leurs domaines d'application.
Le point important qu'on a pu tirer de cette partie est que les graphes sont devenus un moyen crucial et indispensable dans la modélisation des problèmes dans plusieurs domaines. Cependant ils deviennent de plus en plus complexes et volumineux avec la grande quantité de données disponible de nos jours. De ce fait, leur stockage, visualisation et traitement sont devenus difficile. La compression de graphe est naît comme solution à ce problème. Dans le chapitre suivant nous allons présenter la compression de graphes, son rôle et ses différentes méthodes.

Chapitre 2

Compression de graphes

La puissance des processeurs, de nos jours, augmente plus vite que les capacités de stockage ce qui engendre un déséquilibre entre le volume des données qu'il est possible de traiter et de stocker. Dès lors, la réduction de la taille des données, plus formellement la compression de données, a été un domaine de recherche très active.

Dans ce chapitre, nous allons tout d'abord introduire le domaine de compression des données et son application dans la théorie des graphes. Puis dans un second temps, nous allons présenter une étude bibliographique sur les méthodes de compression existantes et qui s'inscrivent dans l'une des deux classes de méthodes de compression : les méthodes de compression par les k2-trees et les méthodes de compression par extraction de motifs, pour finir avec une étude comparative entre elles.

2.1 Compression de données :

La compression de données est principalement une branche de la théorie de l'information qui traite des techniques et méthodes liées à la minimisation de la quantité de données à transmettre et à stocker. Sa caractéristique de base est de convertir une chaîne de caractères vers un autre jeu de caractères occupant un espace mémoire le plus réduit possible tout en conservant le sens et la pertinence de l'information (Lelewer and Hirschberg, 1987).

Les techniques de compression de données sont principalement motivées par la nécessité d'améliorer l'efficacité du traitement de l'information. En effet, la compression des données en tant que moyen peut rendre l'utilisation des ressources existantes beaucoup plus efficace.

De ce fait, une large gamme d'applications usant du domaine de compression tel que le domaine des télécommunications et le domaine du multimédia est apparue offrant une panoplie d'algorithmes de compression (Sethi et al., 2014). Sans les techniques de compression, Internet, la télévision numérique, les communications mobiles et les communications vidéo, qui ne cessaient de croître, n'auraient été que des développements théoriques.

2.2 Compression appliquée aux graphes :

La compression des graphes a été proposée comme solution pour le traitement et le stockage des graphes volumineux. Elle permet de transformer un grand graphe en un autre plus petit tout en préservant ses propriétés générales et ses composants les plus importants. Les principaux avantages de la compression sont (Liu et al., 2018a) :

— **La réduction de la taille des données et de l'espace de stockage :** De nos jours, les graphes représentant les bases de données, les réseaux sociaux et tous types de données numériques sont caractérisés par une croissance exponentielle de leurs volumes, ce qui rend leur stockage difficile et couteux en terme d'espace mémoire. Les techniques de compression produisent des graphes plus petits qui nécessitent moins d'espace. Cela permet aussi de réduire le nombre d'opérations d'E/S ainsi que les communications entre nœuds dans un environnement distribué et de charger le graphe en mémoire centrale.

— **L'exécution rapide des algorithmes de traitement et des requêtes sur les graphes :** l'exécution des différents algorithmes de traitement sur des graphes volumineux peut s'avérer couteuse en terme de temps et peut ne pas donner les résultats attendus. La compression permet d'obtenir de petits graphes qui peuvent être traités, analysés et interrogés plus efficacement et dans un temps raisonnable.

— **La Facilité d'analyse et de visualisation des graphes :** Les techniques de compression permettent de représenter les données et les structures des graphes massives d'une manière plus significative permettant ainsi leur analyse et leur visualisation contrairement au graphes d'origines qui ne peuvent même pas être chargés en mémoire.

— **L'élimination du bruit :** l'un des exemples les mieux illustrant cet avantage sont les grands graphes du web qui sont considérablement bruités. Ce bruit peut perturber l'analyse en faussant les résultats et en augmentant la charge de travail liée au traitement des données. La compression permet donc de filtrer le bruit et de ne mettre en évidence que les données importantes.

2.2.1 Les types de compression :

La compression de graphes est définie comme l'ensemble des méthodes et techniques permettant de réduire l'espace mémoire occupé par ces derniers tout en gardant la même signification que le graphe d'origine. Dès lors, deux approches se présentent : la compression avec ou sans perte, que nous allons détailler dans ce qui suit.

a) 2.2.1.1 Compression Sans Perte :

Certains domaines d'application de la compression nécessitent un niveau élevé d'exactitude et une restitution exacte, donc une compression sans perte. Dans cette catégorie, le graphe G subit des transformations pour avoir une représentation compacte G' qui lors de la décompression donne exactement G. La figure ci-dessous illustre cette définition.

FIGURE 2.1 – Compression sans perte.

b) **2.2.1.2 Compression Avec Perte :**

Contrairement à la compression sans perte, la compression avec perte permet la suppression permanente de certaines informations jugées inutiles (redondantes) pour améliorer la qualité de la compression. En d'autres termes, le graphe G subit des transformations pour avoir une représentation compacte G' qui lors de la décompression donne un graphe G'' probablement différent de G mais l'approximant le plus possible. La figure ci-dessous illustre cette définition.

FIGURE 2.2 – Compression avec perte.

2.2.2 Les métriques d'évaluation des algorithmes de compression :

Devant la panoplie d'algorithmes et de techniques de compression de graphe disponibles dans la littérature, des critères de comparaison et d'évaluation entre ces méthodes doivent être bien définis. Dans cette partie, nous présenterons les principales mesures de performances.

a) **2.2.2.1 Le temps de compression :**

C'est une métrique qui donne le temps d'exécution de l'algorithme de compression. Elle est généralement mesurée en secondes (ou ms).

b) **2.2.2.2 Le ratio de compression :**

Le ratio de compression (CR) est la mesure la plus courante pour calculer l'efficacité d'un algorithme de compression. Il est défini comme le rapport entre le nombre total de bits requis pour stocker les données non compressées et le nombre total de bits nécessaires pour stocker les données compressées.

$$CR = \frac{Nbre.\ de\ bits\ du\ graphe\ originale}{Nbre.\ de\ bits\ du\ graphe\ finale}$$

Le CR est parfois appelé bit par bit (bpb) et il est défini alors comme étant le nombre moyen de bits requis pour stocker les données compressées (Uthayakumar et al., 2018). Dans le cas des algorithmes de compression de graphe on a :

— **Le nombre de bits par nœud :** représente l'espace mémoire nécessaire pour stocker un nœud (bpn pour « *bits per node* » en Anglais).

— **Le nombre de bits par lien :** représente l'espace mémoire nécessaire pour stocker un arc dans le cas d'un graphe orienté ou une arête dans le cas d'un graphe non orienté (bpe pour « *bits per edge* » en Anglais).

c) **2.2.2.3 Le taux de compression :**

Exprimée en pourcentage, cette métrique permet de mesurer la performance de la méthode de compression. Elle peut être exprimée de deux manières différentes :

— **Le taux de compression :** Le rapport entre volume du graphe après compression et le volume initial du graphe.

$$t = \frac{La\ taille\ du\ graphe\ finale}{La\ taille\ du\ graphe\ originale}$$

— **Le gain d'espace :** Le gain d'espace représente la réduction de la taille du graphe compressé par rapport à la taille du graphe original.

$$G = 1 - \frac{La\ taille\ du\ graphe\ finale}{La\ taille\ du\ graphe\ originale}$$

2.2.3 Classification des méthodes de compression :

Le domaine de compression des graphes est un domaine qui a connu une grande évolution vu son importance. Une multitude de méthodes ont été proposées au cours des dernières années. Elles diffèrent les unes des autres selon plusieurs points : le type de graphe en entrée, le type de structure en sortie, le type de compression et la technique utilisée pour la compression. En se basant sur ces différences, plusieurs classifications ont été suggérées. Nous allons dans ce qui suit présenter les plus importantes parmi ces classifications.

Dans (Maneth and Peternek, 2015), les auteurs proposent une classification basée tout d'abord sur le type de compression. Ils regroupent les méthodes en deux catégories principales : les méthodes de compression sans perte et les méthodes de compression avec perte. La première catégorie est subdivisée à son tour selon le type de représentation du graphe en sortie : représentation succinte, représentation structurelle ou une représentation sous forme de fichier RDF [1]. Les méthodes donnant une représentation succincte représentent le graphe sous forme d'une chaine de bits succincte irréversible. La sortie de ces méthodes est ainsi une structure compacte du graphe original. Parmi les méthodes de cette classe, nous trouvons : Web Framework de Boldi et Vigna (Boldi and Vigna, 2004). La deuxième classe est la représentation structurelle. Contrairement à l'approche précédente, les méthodes de cette classe modifient la structure du graphe initial, sachant que les modifications apportées sont réversibles. La sortie sera donc une structure réduite et non pas compacte de la version initiale. Parmi ces méthodes, nous citons :

1. Resource Description Framework (RDF) : est un modèle de graphe destiné à décrire de façon formelle les ressources Web et leurs métadonnées, de façon à permettre le traitement automatique de telles descriptions.

RePair de Claude et Navarro (Claude and Navarro, 2010b). La dernière classe est la compression des fichiers RDF et qui comporte des méthodes assez récentes. Nous trouvons parmi ces techniques : Dcomp de (Martínez-Prieto et al., 2012) . Les méthodes de compression avec perte quant à elles apportent des modifications irréversibles sur le graphe en supprimant les informations redondantes et le bruit. Comme exemple, nous citons : ASSG de Zhang et al. (Zhang et al., 2014a).

Une autre classification a été exposée par Lui et al. dans (Liu et al., 2018a) qui classe les méthodes sur trois niveaux. Au premier et deuxième niveaux, les techniques de compression sont regroupées en fonction du type de graphe en entrée selon deux critères : graphe statique ou dynamique et graphe simple ou étiqueté. Pour le troisième niveau, les auteurs catégorisent les méthodes selon la technique de traitement utilisée. Quatre catégories sont définies : les méthodes de regroupement ou d'agrégation, ces méthodes permettent d'agréger de manière récursive un ensemble de nœuds, liens ou carrément un cluster en un super nœud (appelé parfois nœud virtuel), comme exemple de ces techniques, nous trouvons Grass (LeFevre and Terzi, 2010). Le deuxième type de méthodes englobe les méthodes de compression de bits. Ces méthodes minimisent le nombre de bits nécessaires au stockage du graphe en se basant sur le principe de description minimal (Minimum Description Length (MDL) en Anglais). Elles peuvent être avec ou sans perte. Parmi elles, nous citons LSH-based (Khan et al., 2014). La troisième classe comporte les méthodes de simplification qui suppriment les arêtes les moins importantes selon un certain critère. Parmi ces méthodes, nous trouvons celle proposée dans pqr Shen et al. (Shen et al., 2006). La dernière catégorie est la classe des méthodes basées sur l'influence, les méthodes de cette catégorie décrivent le graphe par les flux d'influence les plus importants ce qui permet de l'analyser plus facilement. Ces méthodes permettent de formuler le problème de compression comme un processus d'optimisation dans lequel la quantité de données liée à l'influence est maintenue en sortie. Parmi ces techniques, nous mentionnons (Shi et al., 2015).

La dernière classification que nous allons présenter est la classification proposée dans un master de l'année dernière par le binôme : Mlle. Belhocine et Mr. Guermah (W. GUERMAH, 2018). Cette classification se base sur le principe utilisé dans le processus de compression. Elle regroupe six classes de méthodes : 1) compression basée sur l'ordre des nœuds en exploitant le principe de similarité et de localité du graphe, les méthodes de cette catégorie cherchent à trouver un ordre des nœuds, qui doit répondre à deux propriétés essentielles : la similarité [2] et la localité [3], cet ordre est ensuite utilisé dans la construction d'une structure de données qui compresse le graphe en entrée, comme exemple de ces méthodes, nous trouvons Layered Label Propagation (Boldi et al., 2011) qui utilise l'ordre LLP [4] et Recursive Graph Bisection de (Dhulipala et al., 2016) qui utilise un ordre BP [5]. 2) compression basée sur l'ordre des nœuds

2. Deux nœuds proches ont tendance à avoir des voisins similaires.

3. Les liens sortants d'un nœud ont tendance à se diriger vers un ensemble de nœuds qui sont proches.

4. LLP est un algorithme itératif qui produit une séquence d'ordres de nœuds en se basant sur les étiquettes affectées aux clusters après avoir partitionner le graphe initial.

5. BP est un ordre basé sur le problème de bissection de graphes, qui cherche à trouver une meilleure partition des nœuds du graphe tout en minimisant une fonction objectif.

en exploitant la linéarisation du graphe, les méthodes de cette classe se basent sur une nouvelle structure de données intitulée structure de données eulérienne, elles sont conçues principalement pour les grands graphes, parmi ces méthodes nous trouvons Neighbor Query Friendly Compression (Maserrat and Pei, 2012). 3) compression basée sur l'étiquetage des nœuds par des intervalles, ces méthodes visent à construire une structure d'index à partir du graphe initial qui permet de répondre aux requêtes de voisinage, parmi ces méthodes nous citons DAGGER (Yıldırım et al., 2012). 4) compression basée sur la structure d'arbre K^2, les méthodes de cette catégorie s'appuient sur la représentation K^2-trees pour compresser le graphe, ces méthodes font l'objet de notre travail et seront détaillées par la suit. 5) compression basée sur l'agrégation des nœuds, ces méthodes sont parmi les méthodes les plus populaires dans le domaine de compression, elles cherchent à compresser le graphe initial en agrégeant un certain nombre de nœuds en un seul nœud appelé super-nœud, cette agrégation se fait selon différentes manières, elle peut être orientée par une fonction objectif représentant l'espace de stockage optimal généralement établie par le principe de la longueur de description minimale MDL ou par la similarité des caractéristiques des nœuds tels que les labels et les attributs ou par l'extraction de motifs en utilisant des techniques de pattern mining, cette dernière représente l'une des matières de notre recherche. 6) compression basée sur l'agrégation des liens, contrairement aux techniques de la classe précédente, les méthodes de cette classe visent à compresser le graphe initial en fusionnant certains ensembles de ses liens en un seul liens appelé super-lien, elle est établie selon deux façons, elle est orientée soit par le biais des règles de grammaire, ou bien par l'extraction de motifs que nous allons étudier par la suit. la figure 2.3 représente le schéma de cette classification.

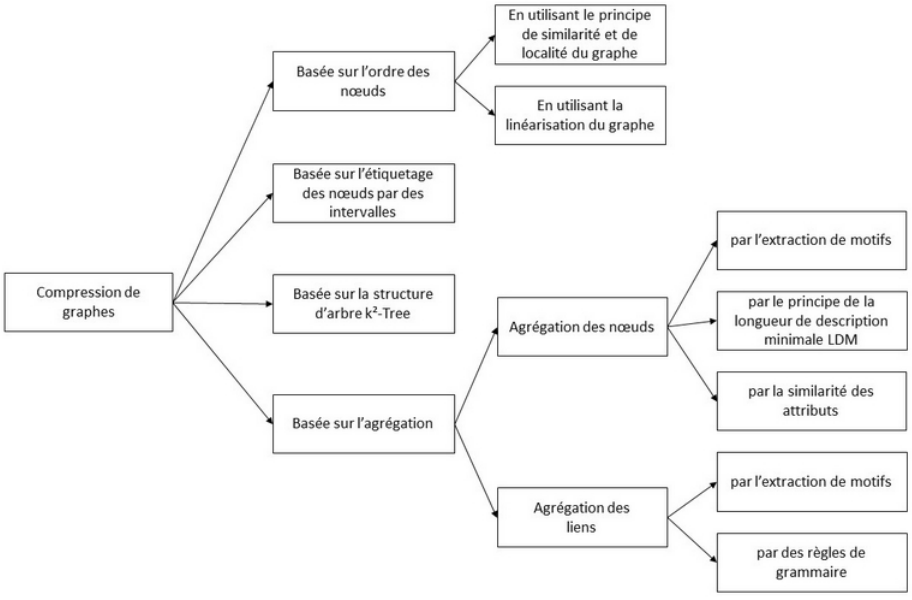


FIGURE 2.3 – Classification des méthodes de compression (W. GUERMAH, 2018)

Après l'étude des méthodes basées sur l'agrégation par extraction de motifs, nous avons constaté que certaines classes ne sont pas bien définies. Nous avons remarqué que la classifi-

cation a proposé d'inclure les méthodes basées sur l'extraction de motifs dans deux classes de méthodes qui sont : les méthodes basée sur l'agrégation des liens et les méthodes basée sur l'agrégation des nœuds, or ce sont les méthodes d'extraction de motifs qui englobent certaines méthodes d'agrégation et non pas le contraire. En effet, notre étude bibliographique a clairement montré que les méthodes de compression de graphe par extraction de motifs ne sont pas toutes des méthodes agrégatives. Nous trouvons, par exemple, certaines de ces méthodes qui donnent en sortie une liste des motifs les plus pertinents du graphe avec une matrice d'erreurs sans avoir recours à l'agrégation. Donc nous avons proposé de dissocier la classe des méthodes basées sur l'extraction de motifs de la classe des méthodes basées sur l'agrégation. En outre, nous avons considéré, contrairement à la classification précédente, que les méthodes de compression basées sur les règles de grammaire font parties des méthodes de compression par extraction de motifs. Nous justifiant cela par le fait que ces méthodes utilisent l'un des deux principes suivants :

1. La transformation de la liste d'adjacence en une chaine de caractères et le remplacement, de manière itérative, de la sous-chaine de longueur deux la plus fréquente par une règle de production et qui représente donc dans ce cas un motif.
2. La recherche d'un motif, de type *digram* [6], satisfaisant une certaine condition et le remplacement de toutes ses occurrences par une production dans la grammaire.

Le fait que les motifs dans le cas de cette classe n'ont pas une structure de sous-graphes connus (clique, étoile, ...) n'empêche pas que le processus appliqué s'intègrent toujours dans la discipline d'extraction de motifs. Nous avons essayé donc de rectifier ces imperfections tout en raffinant davantage les deux classes qui font l'objet de notre Master.

La classe des méthodes de compression par les arbres k2-trees ne peut être encore raffinée car elles partagent toutes le même principe et diffèrent dans le type de graphe en entrée ou dans le codage en sortie.

Pour la classe des méthodes de compression par extraction de motifs, nous distinguons cinq classes : 1)les méthodes de compression basées vocabulaire faisant appel aux méthodes de clustering, 2) les méthodes de compression basées vocabulaire exploitant les propriétés de la matrice d'adjacence, 3) les méthodes de compression basées agrégation des nœuds des motifs, 4) les méthodes de compression basées agrégation des liens des motifs utilisant des règles de grammaire, 5) les méthodes de compression basées agrégation des liens des motifs faisant appel à des heuristiques de partitionnement de graphes. La première et la deuxième classe se caractérisent par l'ensemble des motifs qui est prédéfini au départ. Cependant, elles se distinguent l'une de l'autre dans le principe de fonctionnement : l'une utilise des méthodes de clustering et de détection de communautés tant dis que l'autre utilise directement la matrice d'adjacence en exploitant ses propriétés. Les trois (03) dernières classes forment l'ensemble des méthodes d'extraction de motifs basées sur l'agrégation de ces derniers.

La figure 2.4 représente la classification de l'an dernier après raffinement où nous avons mis en évidence nos apports et nos modifications en pointillé.

6. Un digramme : est sous-graphe composé de deux arêtes ayant un sommet en commun.

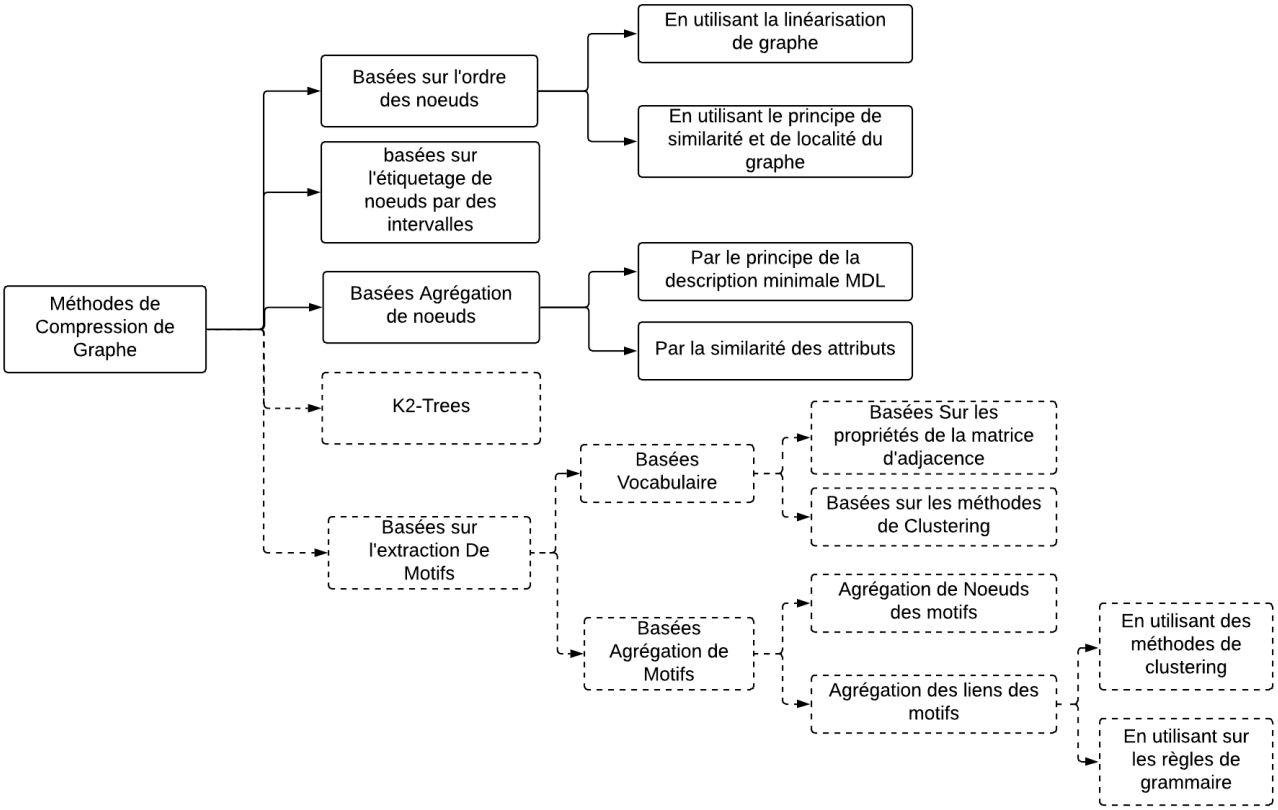


FIGURE 2.4 – Classification proposées des méthodes de compression

2.3 Compression par les arbres K^2-Trees

k^2-tree est une structure de données conçue à l'origine pour la compression des graphes du web. L'algorithme de base a été proposé par Bisaboa et al. dans leur article *k^2-trees for Compact Web Graph* (Brisaboa et al., 2009). Elle a été appliquée ensuite dans d'autres types de données tel que les réseaux sociaux (Shi et al., 2012), les données rasters (De Bernardo et al., 2013) et les bases de données RDF (Alvarez-Garcia et al., 2017).

En général, l'algorithme peut être appliqué à n'importe quelle matrice binaire. Dans le cadre de notre étude nous nous intéressons seulement à la matrice d'adjacence d'un graphe. La compression par les k^2-trees exploite les propriétés de la matrice d'adjacence et tire parti des zones vides pour réduire l'espace de stockage et permettre au graphe de tenir en mémoire centrale. Il offre aussi la possibilité de naviguer dans le graphe sans le décompresser, et de répondre aux requêtes de voisinage direct et inverse.

Étant donné une matrice d'adjacence A d'ordre $|n|$, k^2-tree la représente sous forme d'un arbre de recherche k^2-aires [7] de hauteur h = $[log_k$ n]. Chaque nœud de l'arbre contient un seul bit avec deux valeurs possibles : 1 pour les nœuds internes et 0 pour les feuilles, sauf le dernier niveau où les feuilles représentent les cases de la matrice A et peuvent prendre une valeur 0 ou

7. Les arbres n-aires sont une généralisation des arbres binaires : chaque nœud a au plus n fils.

1. Chaque nœud interne de l'arbre a exactement k^2 fils. Avant la construction de l'arbre, il faut s'assurer que n est une puissance de k. Dans le cas contraire, l'algorithme étend la matrice en rajoutant des zéros à droite et en bas de la matrice. L'ordre de la matrice devient donc $n' = k^{log_k n}$.

Pour construire l'arbre, k^2-tree commence par diviser la matrice en k^2 sous matrices d'ordre $|n/k|$. La racine correspond à la matrice complète. Chaque sous matrice représente un nœud dans le premier niveau de l'arbre, elle est ajoutée comme un fils à la racine suivant un ordre de gauche à droite et de haut en bas. Le nœud est à 1 si la sous matrice qu'il représente contient au moins un 1, et à 0 si elle ne contient que des 0. Le processus est répété de manière récursive sur les sous matrices représentées par des 1. k^2 sous matrices sont considérées à chaque subdivision. L'opération est répétée jusqu'à ce que la subdivision atteint les cases de la matrice qui représenterons les feuilles de l'arbre au dernier niveau.

La figure 2.5 illustre la représentation k^2-tree d'une matrice de taille 10×10, étendue à une taille 16×16 pour un k=2 (Brisaboa et al., 2015).

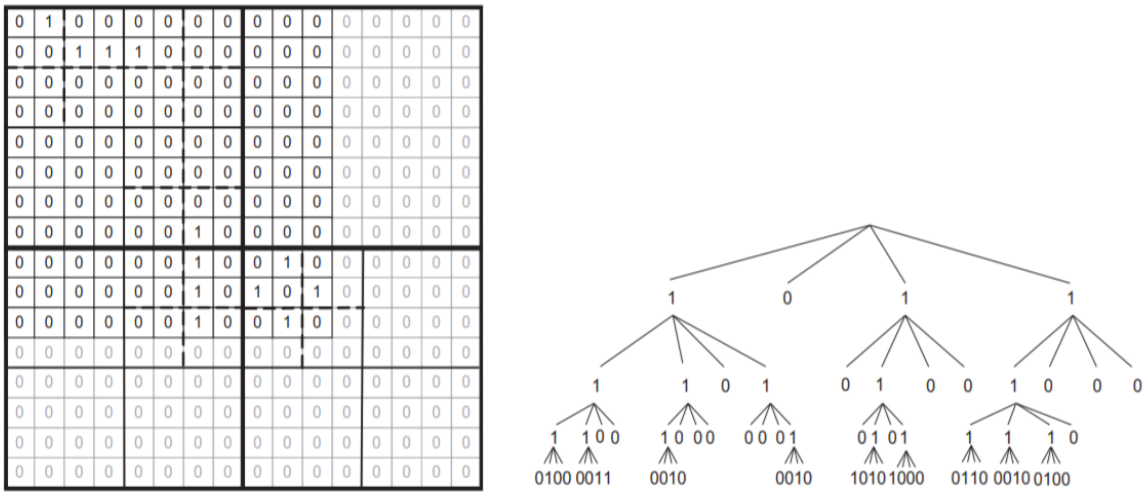


FIGURE 2.5 – Exemple de représentation k^2-tree

Pour le stockage de l'arbre, l'algorithme utilise deux tableaux binaires : un tableau T (Tree) contenant tous les nœuds de l'arbre à l'exception du dernier niveau et un tableau L (Leaves) contenant les feuilles du dernier niveau. Les nœuds et les feuilles sont ordonnés selon un parcours en largeur de l'arbre. Ci-dessous les deux tableaux T et L de l'exemple précédent (figure 2.5) :

T = 1011 1101 0100 1000 1100 1000 0001 0101 1110
L = 0100 0011 0010 0010 1010 1000 0110 0010 0100

Dans le pire des cas, l'espace total pour la description de la structure est $k^2 * m(log_{k^2} \frac{n^2}{m} + O(1))$, où n est le nombre de nœuds du graphe et m le nombre de liens. Cependant, pour les graphes réels, l'espace nécessaire pour le stockage est bien meilleur.

Dans le même article (Brisaboa et al., 2009), et dans le but d'obtenir un compromis entre la taille de l'arbre et le temps de parcours, les auteurs proposent une hybridation qui consiste à changer la valeur du paramètre k en fonction du niveau de l'arbre en donnant à k une grande valeur au début pour réduire le nombre de niveaux et améliorer ainsi le temps de recherche, et une petite valeur à la fin pour avoir des petites sous matrices et réduire l'espace de stockage. Pour le stockage de l'arbre, un tableau T_i est utilisé pour chaque valeur k_i, le tableau L reste le même.

Plusieurs variantes de l'algorithme de base ont été proposées dans la littérature dont le but était soit d'obtenir un meilleur résultat de compression, soit d'appliquer la méthode sur d'autres types de graphes. Nous allons dans ce qui suit présenter les principaux travaux qui traitent ce sujet.

Dans (Shi et al., 2012), les auteurs proposent deux techniques d'optimisation de l'algorithme : la première consiste à trouver un certain ordre des nœuds qui permet de regrouper les 1 de la matrice d'adjacence dans une seule sous matrice au lieu qu'ils soient dispersés de manière aléatoire. La recherche d'un ordre optimal des nœuds n'est pas envisageable. Avec k=2,le problème peut être réduit à un autre problème (min bisection [8]) qui est NP-difficile. Les auteurs utilisent alors un parcourt Deapth First Search (DFS) [9] avec des heuristiques pour trouver une approximation de l'ordre optimal. Cette optimisation permet de réduire le nombre de nœuds internes de l'arbre et produit ainsi un arbre optimal. La deuxième optimisation est de trouver la valeur de k la plus adéquate pour chaque nœud interne, calculer cette valeur pour chaque nœud peut engendrer un temps de calcul très important. Pour éviter cela, les auteurs affectent la même valeur k pour les nœuds ayant le même parent.

Dans un travail ultérieur (Brisaboa et al., 2014b), les auteurs de l'article de base apportent deux améliorations principales de leur méthode dans le but d'optimiser l'espace et le temps de parcours de l'arbre produit. La première est de construire k^2 arbres distincts pour les k_0^2 sous matrices du premier niveau. Elle présente plusieurs avantages : (1) l'espace est réduit étant donné que la taille de chaque arbre est en fonction de $\frac{n^2}{k^2}$, (2) le temps de parcours s'améliore puisque T et L sont plus petits. La deuxième amélioration est la compression de L qui consiste à construire un vocabulaire V de toutes les sous matrices du dernier niveau sous forme de séquences de bits, les classer par fréquence d'apparition et remplacer leurs occurrences dans L par des pointeurs. Cela permet d'éviter la redondance et de réduire la taille de la structure. Les pointeurs sont représentés par des codes de longueur variable ordonnés, le plus petit correspond à la sous matrice la plus fréquente. Néanmoins, cette représentation ne permet pas un accès direct dans L étant donné qu'une décompression séquentielle est nécessaire pour récupérer une position. Pour remédier à ce problème, les auteurs utilisent le principe de Directly Addressable Codes (DAC) [10] (Brisaboa et al., 2013) pour garantir un accès rapide au pointeur et conserver

8. Le problème de bissection minimale (MBP) est un problème NP-hard bien connu, qui est destiné à diviser les sommets d'un graphe en deux moitiés égales afin de minimiser le nombre de ces arêtes avec exactement une extrémité dans chaque moitié.

9. DFS : Parcours en profondeurs du graphe

10. DAC est une technique qui encode une séquence d'entiers ou mots en utilisant une structure à longueur

ainsi une navigation efficace.
Exemple : Pour la figure 2.5, le vocabulaire et L sont représentés comme suit :

$$V = [0010\ 0100\ 0011\ 1010\ 1000\ 0110]$$
$$\mathrm{L} = c_1 c_2 c_0 c_0 c_3 c_4 c_5 c_0 c_1$$

Dans (Brisaboa et al., 2012), les même auteurs développent la représentation k^2-trees pour les graphes dynamiques. Ils proposent une nouvelle structure nommée dk^2-trees pour *Dynamique k^2-trees* qui offre les même capacités de compression et fonctionnalités de navigation que le cas statique et qui permet également d'avoir des mises à jour sur le graphe. Pour atteindre ces objectifs, dk^2-tree remplace la structure statique de k^2-tree par une implémentation dynamique. Dans cette nouvelle implémentation, les deux tableaux T et L sont remplacés par deux arbres, nommés T_{tree} et L_{tree} respectivement. Les feuilles de T_{tree} et L_{tree} stockent des parties des bitmaps T et L. La taille de ces feuilles est une valeur paramétrable. Les nœuds internes des deux arbres permettent d'accéder aux feuilles et de les modifier. Chaque nœud interne de T_{tree} contient trois(03) éléments : deux compteurs b et o, qui contiennent respectivement le nombre de bits et le nombre de "1" stockés dans les feuilles descendantes de ce nœud, et un pointeur P vers le nœud fils. Les nœuds internes de L_{trees} sont similaires sauf qu'ils ne contiennent que b et P. Avec cette structuration, T_{tree} et L_{tree} permettent la mise à jour directe de l'arbre $k_2 - tree$ dans le cas l'ajout et la suppression des liens dans le graphe.
La figure 2.6 présente une représentation dk^2-tree (Brisaboa et al., 2012) :

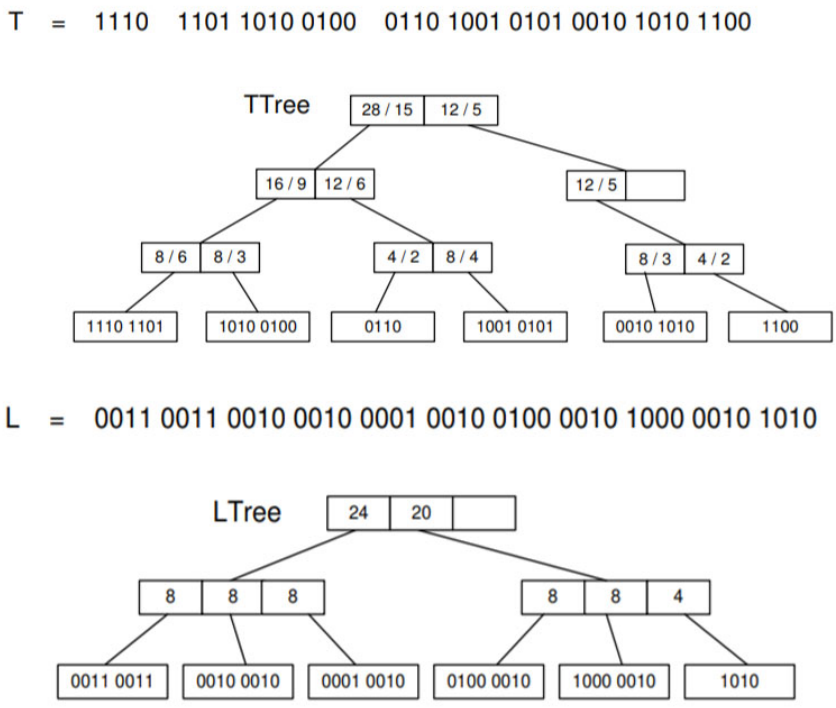


FIGURE 2.6 – Exemple d'une représentation dk^2-tree

variable, son avantage principale est l'accès direct au mot sans passer par le décodage.

Sandra et al. (De Bernardo et al., 2013) présentent k^n-tree, une généralisation des k^2-tree pour les problèmes multidimensionnels. Cette méthode possède plusieurs applications. Elle est utilisée pour représenter les bases de données multidimensionnelles, les données rasters et les graphes dynamiques. k^n-tree repose sur k^2-tree pour représenter une matrice à n-dimensions (dites *tensor* en Anglais). La matrice est décomposé en k^n sous-matrices de même taille, comme suit : sur chaque dimension, K-1 hyperplans devisent la matrice dans les positions i*$\frac{n}{K}$,pour $i \in [1, K-1]$. Une fois les dimensions partitionnées, k^n sous-matrices sont induites, elles sont représentées par des nœuds dans l'arbre comme dans l'algorithme de base. Les structures utilisées pour le stockage sont aussi les mêmes (T et L). En posant n=3, la méthode peut être appliquée sur les graphes dynamiques ou temporels. Ce type de graphe est représenté par une grille à 3 dimensions $X \times Y \times T$, où les deux premières dimensions représentent les nœuds de départ et de destination, et la troisième dimension représente le temps. La figure 2.7 présente une représentation k^3-tree d'un graphe dynamique (de Bernardo Roca, 2014).

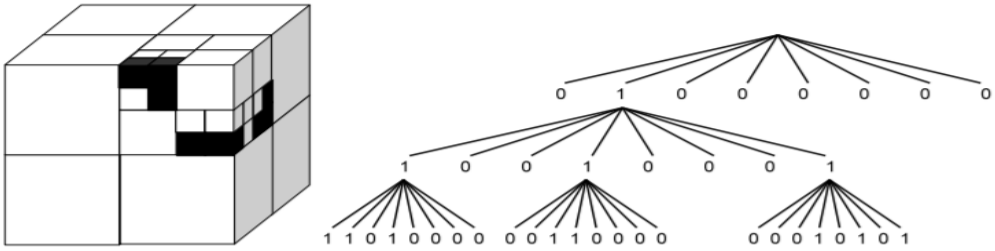


FIGURE 2.7 – Exemple d'une représentation k^3-tree

La représentation de base des arbres k2-trees regroupe seulement les zones de zéros, puisqu'elle a été conçue au début pour les graphes du web qui possèdent une matrice d'adjacence extrêmement creuse. Dans (de Bernardo Roca, 2014), Les auteurs proposent d'étendre cette représentation en regroupant les zones de "1" également. L'idée générale est d'arrêter la décomposition de la matrice d'adjacence quand une zone unie est trouvée, à savoir des zéros ou des uns. Pour distinguer entre les différents nœuds, une représentation quadtree est utilisée (De Berg et al., 1997). Une couleur est attribuée à chaque nœud, blanc pour une zone de zéro, noir pour une zone de uns et gris pour les nœuds internes, i.e, les zones contenant des uns et des zéros. Pour le stockage des nœuds, les auteurs proposent quatre encodages présentés dans ce qui suit :

- k^2**-***trees*$1^{2-bits-naive}$ **:** Dans cet encodage, deux bits sont utilisés pour représenter chaque type de nœud. L'attribution des bits n'est pas arbitraire, le premier bit du poids fort indique si le nœud est un nœud interne (0) ou une feuille (1), le deuxième détermine si les feuilles sont blanches (0) ou noires (1). Nous aurons donc : "10" pour les nœuds gris, "01" pour les nœuds noirs et "00" pour les nœuds blancs. Notant que les feuilles du dernier niveau sont représentées par un seul bit. Après le codage, le premier bit de chaque nœud sauf ceux du dernier niveau est stocké dans T, un autre tableau T' est créé pour sauvegarder le deuxième bit. Les nœuds du dernier niveau sont stockés dans L.

- k^2**-***tree*1^{2-bits} **:** Le même principe que l'encodage précédant sauf que les nœuds gris sont représentés par un seul bit, toujours à "1". Le tableau T' va contenir dans ce cas la couleur des feuilles ce qui va réduire la taille de la structure.
- k^2**-***tree*1^{DF} **:** Cet encodage est similaire à k^2-*trees*1^{2-bits}, mais il utilise un seul bit pour les nœuds blancs et deux bits pour les nœuds noirs et gris, compte tenue de la fréquence des nœuds blancs dans les graphes du monde réel par rapport aux autres. Nous aurons donc : "0" pour les nœuds blancs, "10" pour les nœuds gris et "11" pour les nœuds noirs.
- k^2**-***tree*1^{5-bits} **:** Le dernier encodage repose sur la représentation de base. Un nœud blanc est représenté par "0", un nœud noir ou gris par "1", exactement comme le k^2-tree d'origine. Pour identifier un nœud noir (zone de uns), il sera représenté par une combinaison impossible : k^2 fils de "0" sont ajoutés aux nœuds noirs pour les distinguer.

Dans (Zhang et al., 2014b), les auteurs proposent Delta-k^2-tree, une variante qui exploite la propriété de similarité entre les nœuds voisins du graphe pour réduire le nombre de uns dans la matrice d'adjacence. Notons par *Matrix* la matrice d'adjacence, Delta-k^2-trees construit une nouvelle matrice appelée *Delta-matrix*, une ligne *i* de *Delta-matrix* va contenir la différence entre les deux lignes *Matrix*[i] et *Matrix*[i-1] si cela décroit le nombre de uns sinon elle sera égale à *Matrix*[i], comme suit :

$$\begin{cases} \textbf{Si } count1s(matrix[i]) < countDif(matrice[i], matrix[i-1]) \\ \quad Delta-matrix[i] := matrix[i] \\ \textbf{Sinon} \\ \quad Delta-matrix[i] := matrix[i] \bigoplus matrix[i-1] \end{cases}$$

Où *count1s* compte le nombre de 1 dans une ligne, *countDif* compte le nombre de bits différents entre deux lignes et $\bigoplus$ représente le "ou exclusif".

Pour déterminer si une ligne est identique à celle de la matrice d'adjacence ou non, un tableau D est utilisé : Si D[i]=1 la ligne est identique, sinon c'est une ligne de différence.
La matrice *Delta-matrix* contient moins de uns que la matrice d'adjacence, d'où elle est plus creuse, ce qui permet de réduire la taille de la structure et avoir un meilleur taux de compression. Cependant le temps de parcours est plus grand car pour accéder à certaines lignes (lignes de différence), le graphe doit être décompressé et la matrice d'adjacence reconstituée.
La figure 2.8 présente un exemple de la représentation Delta-k^2-tree (Zhang et al., 2014b).

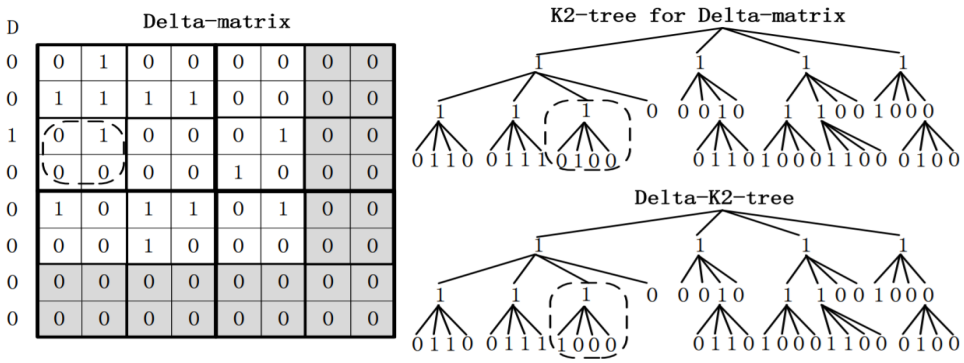


FIGURE 2.8 – Exemple d'une représentation Delta-k^2-tree

K^2-treap est une autre variante de k^2-tree, elle a été proposée dans (Brisaboa et al., 2014a). Cette variante combine les k^2-trees avec une autre structure de données appelée treap [11] (Aragon and Seidel, 1989). Les auteurs appliquent cette méthode sur des grilles multidimensionnelles comme Online Analytical Processing (OLAP) pour pouvoir les stocker et répondre efficacement aux requêtes top-K (Badr, 2013). La méthode peut être également appliquée sur les graphes pondérés, où chaque case de la matrice d'adjacence du graphe comporte le poids de l'arête qu'elle représente au lieu d'un 1. Comme dans l'algorithme de base, une décomposition récursive en k^2 sous-matrices est appliquée sur la matrice d'adjacence et un arbre k^2-air est construit comme suit : la racine de l'arbre va contenir les coordonnées ainsi que la valeur de la cellule ayant le plus grand poids de la matrice. La cellule ajoutée à l'arbre est ensuite supprimée de la matrice. Si plusieurs cellules ont la même valeur maximale, l'une d'elles est choisie au hasard. Ce processus est répété récursivement sur chaque sous matrice. La procédure continue sur chaque branche de l'arbre jusqu'à ce qu'on tombe sur les cellules de la matrice d'origine ou sur une sous matrice complètement vide (contient que des zéros).
La figure 2.9 suivante illustre la représentation k^2-treap d'un graphe pondéré (Badr, 2013) :

11. Les Treaps sont des arbres de recherche binaire avec des nœuds ayant deux attributs : clé et priorité. La recherche dans ces arbres s'effectue selon ces attributs.

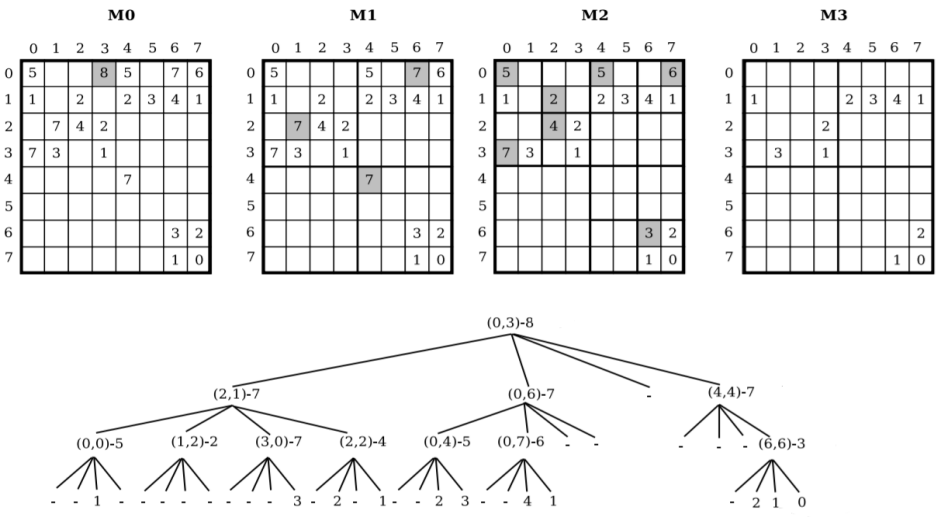


FIGURE 2.9 – Exemple d'une représentation k^2-treap d'un graphe pondéré

Pour avoir une bonne compression, k^2-treap effectue des transformations sur les données stockées. La première transformation consiste à changer les coordonnées représentées dans l'arbre en des coordonnées relatives par rapport à la sous matrice actuelle. La deuxième est de remplacer chaque poids dans l'arbre par la différence entre sa valeur et celle de son parent.
Trois structures de données sont utilisées pour sauvegarder les coordonnées et les valeurs des cellules ainsi que la topologie de l'arbre. Chaque structure est détaillée dans ce qui suit :

— *Listes de coordonnées locales :* La séquence de coordonnées de chaque niveau l de l'arbre est stockée dans une liste $coord[l]$.

— *Liste des valeurs :* Le parcours de l'arbre se fait en largeur, la séquence des valeurs récupérées est stockée dans une liste nommée *values*. Un tableau nommé *first* est utilisé pour sauvegarder la position début de chaque niveau dans *values*.

— *L'arbre :* La structure de l'arbre k^2-treap est sauvegardée avec un arbre k^2-tree, les nœuds contenant des valeurs dans k^2-treap sont représentés par des uns et les nœuds vides par des zéros. Pour le stockage de l'arbre, un seul tableau T est utilisé.

Garcia et al. (Garcia et al., 2014) proposent Ik^2-tree pour Interleaved k^2-tree. Elle est appliquée sur les bases de données RDF ainsi que sur les graphes dynamiques. Dans les graphes dynamiques, les deux premières dimensions correspondent aux nœuds source et destination et la troisième dimension reflète le temps. Le graphe est donc défini par |T| matrices d'adjacence prisent à des instants t_k différents. Ik^2-tree représente les matrices simultanément. Chaque matrice est représentée par un arbre k^2-tree et ils sont par la suite regroupés dans un seul arbre. Chaque nœud de l'arbre obtenu représente une sous-matrice comme dans l'algorithme de base, sauf qu'au lieu d'utiliser un seul bit, Ik^2-tree utilise 1 à |T| bits pour représenter le nœud. Le nœud racine contient |T| bits. Le nombre de bits de chaque fils dépend du nombre de uns de son parent. L'arbre finale est toujours stocké avec deux tableaux : T et L. La figure 2.10 est un exemple de Ik^2-tree appliqué sur un graphe dynamique représenté dans trois instants (Garcia et al., 2014) :

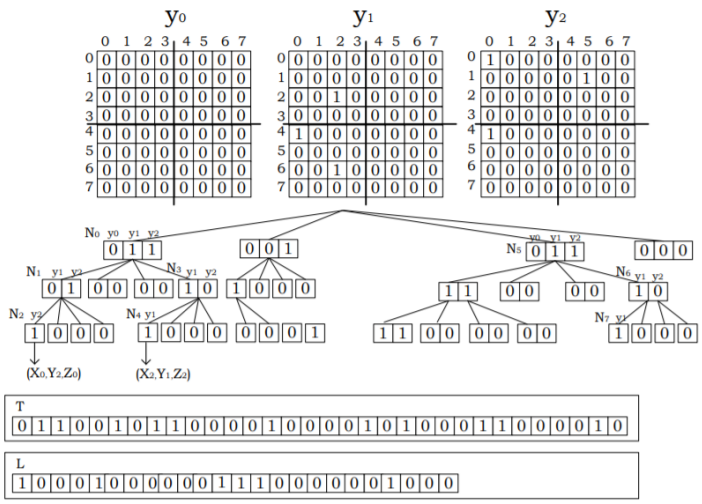


FIGURE 2.10 – Exemple d'une représentation Ik^2-tree

Une variante de Ik^2-tree appelée Differential Ik^2-tree a été étudiée dans (Alvarez-Garcia et al., 2017). Elle vise à améliorer le taux de compression en représentant uniquement les changements survenus sur le graphe à un instant t_i au lieu d'une instance complète : à l'instant t_0, une capture complète du graphe (matrice d'adjacence) est stockée. À l 'instant t_k, pour k>0, seules les arrêtes qui changent de valeurs entre t_{k-1} et t_k sont stockées. Les matrices sont représentées à la fin de la même manière que Ik^2-tree. La limite de cette représentation est que la structure doit être décompressée lors d'une requête. La figure 2.11 montre un exemple d'une représentation diffIk^2-trees (Alvarez-Garcia et al., 2017).

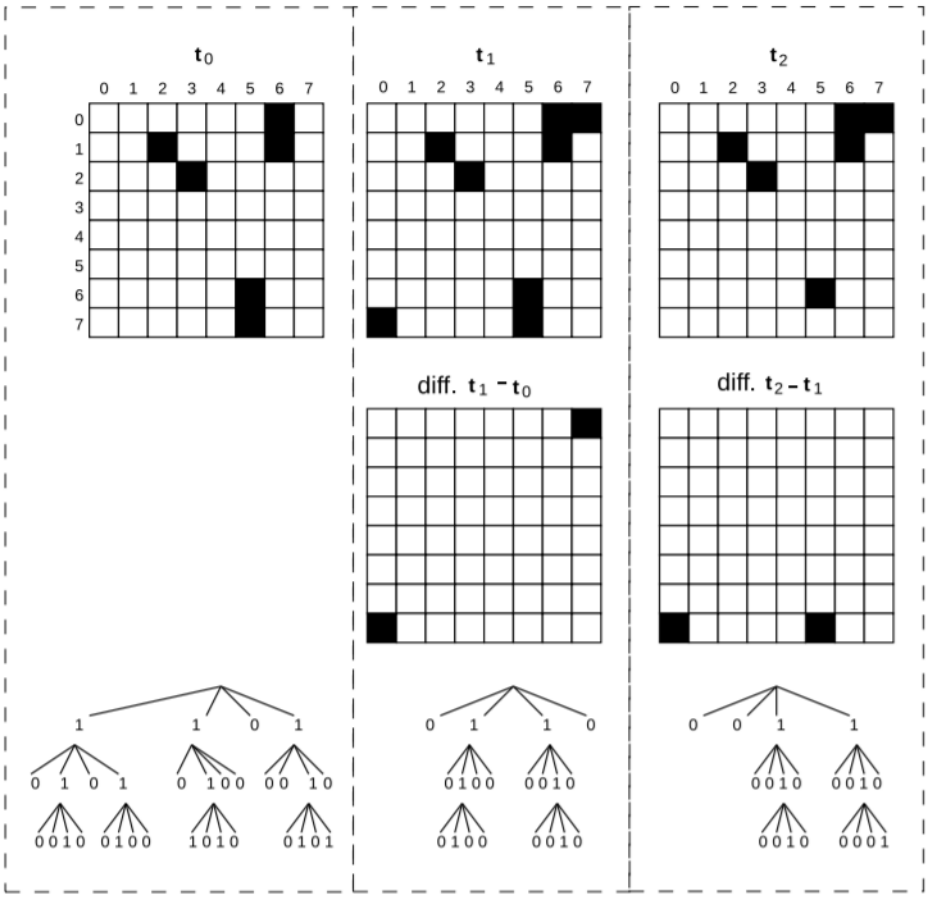


FIGURE 2.11 – Exemple d'une représentation diff Ik^2-tree

Dans (Álvarez-García et al., 2018), les auteurs étendent la représentation k^2-tree pour les bases de données orientées graphes. Ces graphes sont étiquetés, attribués, orientés et ont des arêtes multiples. Ils présentent le graphe sous forme d'une nouvelle structure intitulée Attk^2-tree pour Attributed k^2-trees.

Structures de données : La représentation obtenue par la compression est composée d'un ensemble d'arbres k^2-tree et d'autres structures supplémentaires. Le graphe est représenté par trois composantes : un schéma de données, les données incluses dans les nœuds et les liens et finalement la relation entre les éléments du graphe. Chaque composant est présenté dans ce qui suit :

— *Schéma :* Ce composant gère les étiquettes et les attributs de chaque type d'éléments, il joue le rôle d'un index dans la représentation. Il est composé de :

Un schéma de nœuds : représenté par un tableau qui contient les étiquettes des nœuds ordonnées lexicographiquement. Un identifiant est attribué à chaque nœud du graphe selon l'ordre du tableau, les m_1 nœuds possédant la première étiquette du tableau vont avoir des identifiants de 1 à m_1, les m_2 nœuds avec la deuxième étiquette du tableau vont avoir des identifiants de m_1+1 à m_1+m_2 et ainsi de suite. Chaque entrée du tableau va ainsi stocker le plus grand identifiant portant son étiquette, cela permet de trouver l'étiquette d'un nœud à travers son identifiant.

Un schéma d'arêtes : Comme dans le cas des nœuds, un tableau est utilisé pour stocker les étiquettes des arêtes avec le même principe.

Le schéma est le point de départ de la représentation, il permet d'obtenir l'étiquette d'un nœud ou d'une arête, et d'accéder à ses attributs.

— *Données :* Ce composant contient toutes les valeurs que peut prendre un attribut dans le graphe. Un attribut peut être représenté de deux façons différentes selon sa fréquence d'apparition, on distingue donc deux types d'attributs :

Attributs rares : Ce sont les attributs qui prennent généralement des valeurs différentes à chaque apparition, ils sont stockés dans des listes et indexés avec l'identifiant de l'élément.

Attributs fréquents : Ce type d'attributs est sauvegardé dans deux matrices, une pour les attributs des nœuds et l'autre pour les attributs des liens. Les matrices sont stockées sous forme d'arbres k^2-trees.

— *Relations :* C'est le dernier composant de Attk^2-tree, il stocke les relations entre les nœuds et les arêtes du graphe en utilisant un arbre k^2-tree et d'autres structures pour sauvegarder les identifiants des arêtes ainsi que les arêtes multiples. Les structures supplémentaires sont les suivantes :

Multi : Un tableau qui indique si l'arête est multiple ou non.

Firt : Un tableau qui donne l'identifiant de l'arête, ou de celui de la première dans le cas d'une arrête multiple.

Next : Un tableau qui contient les identifiants des arêtes multiples restantes.

La figure 2.12 donne la représentation Attk^2-tree des relations du graphe de la figure **??** (Álvarez-García et al., 2018) :

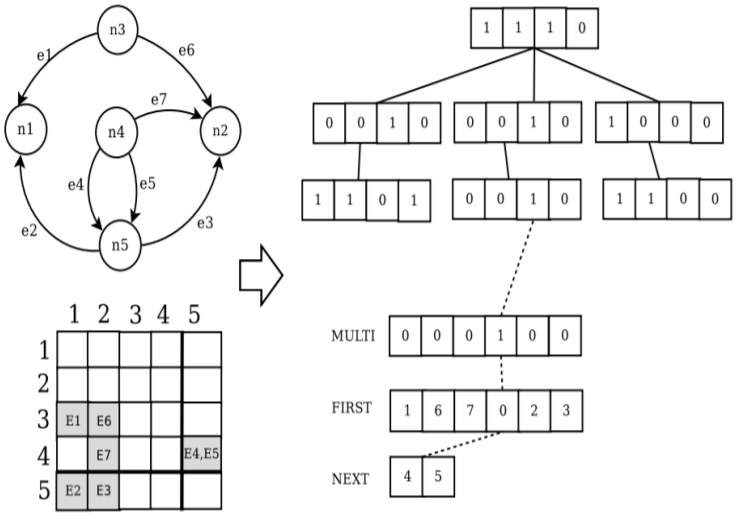


FIGURE 2.12 – Exemple d'une la représentation Attk^2-tree

Dans le même article (Álvarez-García et al., 2018), les auteurs étendent Attk^2-tree pour les graphes dynamiques. Ils proposent une nouvelle variante appelée dynAttk^2-tree qui supporte le

changement dans les attributs et les liens du graphe. Comme Attk^2-tree, dynAttk^2-tree représente le graphe avec trois composantes : Schémas, données et relations. Les composantes sont semblables à ceux de Attk^2-tree mais avec certaines amélioration vue la nature dynamique du graphe.

Structure de données :

— *Schéma :* En ce qui concerne les nœuds, leurs étiquettes sont stockées dans une liste dynamique ordonnée lexicographiquement. En outre, une séquence dynamique est utilisée pour sauvegarder le type de chaque nœud. Elle est stockée ensuite sous forme d'un arbre d'ondelettes [12] (Grossi et al., 2003). Le même principe est appliqué sur les arrêtes.

— *Données :* Les attributs rares sont stockés dans des listes dynamiques, quant aux attributs fréquents, ils sont sauvegardés avec des arbres dk^2-trees (un arbre pour chaque attribut).

— *Relations :* Le stockage des relations se fait à l'aide d'un dk^2-tree et des tableaux dynamiques pour stocker les identifiants des arêtes et les arêtes multiples.

2.3.0.1 Synthèse des méthodes de compression basée sur les K^2-trees

Nous avons présenté dans les parties précédentes les différentes méthodes de compression existantes basées sur la représentation k^2-trees, et expliqué le principe de fonctionnement de chaqu'une d'elles. Nous allons par la suite comparer entre ces méthodes et résumer nos recherches.

Une présentation synthétique de notre étude est fournie dans le tableau 2.2. Chaque ligne du tableau représente une méthode, tandis que chaque colonne représente un aspect susceptible d'être utilisé dans la méthode (type de graphe, type de compression, structure en sortie). Nous constatons que toutes les méthodes de cette classe sont des méthodes de compression sans perte qui supportent les graphes orientés et non orientés, cependant nous remarquons une variation dans l'aspect temporelle, certains méthodes sont destinées aux graphes statiques comme k^2-trees de base et k^2-trees1, tandis que d'autres sont appliquées aux graphes dynamiques comme k^n-trees et dk^2-trees. Certains méthodes se distinguent aussi par le type de graphe comme k^2-treaps qui s'applique aux graphes pondérés et Attk^2-trees qui accepte les graphes étiquetés, attribués et multiples. Nous observons aussi que toutes les méthodes donnent en sortie une représentation succincte. Le tableau résume aussi les résultats de l'application des méthodes sur des graphes de test.

12. Ou wavelet en anglais, est un arbre binaire équilibré qui contient des données compressées dans une représentation optimale.

Article	Graphe en entrée					Compression		Structure en sortie		Graphe de test	Résultat
	Orienté	Non orienté	Statique	Dynamique	Autre Propriétés	Avec perte	Sans perte	Succincte	Structurelle		
k^2-trees : Algorithme de base (Brisaboa et al., 2009)	✓	✓	✓	✗		✗	✓	✓	✗	eu-2005	5.21 bits/lien
											taux de compression : 16.88%
										cond-mat	taux de compression : 15.58%
k^2-trees : Hybridation (Brisaboa et al., 2009)	✓	✓	✓	✗		✗	✓	✓	✗	eu-2005	5.21 bits/lien
k^2-trees : Optimisation (Shi et al., 2012)	✓	✓	✓	✗		✗	✓	✓	✗	cond-mat	taux de compression : 37.96%
k^2-trees : Amélioration (Brisaboa et al., 2014b)	✓	✓	✓	✗		✗	✓	✓	✗	eu-2005	3.22 bits/lien
dlk^2-trees (Brisaboa et al., 2012)	✓	✓	✗	✓		✗	✓	✓	✗	eu-2005	6.2 bits/lien
k^n-trees (De Bernardo et al., 2013)	✓	✓	✗	✓		✗	✓	✓	✗	CommNet	taux de compression : 65.16%

TABLE 2.1 – Synthèse des méthodes de compression par k^2-trees.

Article	Graphe en entrée					Compression		Structure en sortie		Graphe de test	Résultat
	Orienté	Non orienté	Statique	Dynamique	Autre Propriétés	Avec perte	Sans perte	Succincte	Structurelle		
k^2-trees1(de Bernardo Roca, 2014)	✓	✓	✓	✗		✗	✓	✓	✗	eu-2005	taux de compression : 16.23%
Delta-k^2-trees (Zhang et al., 2014b)	✓	✓	✓	✗		✗	✓	✓	✗	eu-2005	3.24 bits/lien
k^2-treaps (Brisaboa et al., 2014a)	✓	✓	✓	✗	Pondéré	✗	✓	✓	✗	SalesDay	2.48 bits/lien
Ik^2-trees (Garcia et al., 2014)	✓	✓	✗	✓		✗	✓	✓	✗	CommNet	taux de compression : 60.43%
Diff Ik^2-trees (Alvarez-Garcia et al., 2017)	✓	✓	✗	✓		✗	✓	✓	✗	CommNet	taux de compression : 60.43%
Att k^2-trees (Álvarez-García et al., 2018)	✓	✓	✓	✗	Étiqueté Attribué Multiple	✗	✓	✓	✗	Movielen-10M	taux de compression : 89.97%
dynAtt k^2-trees (Álvarez-García et al., 2018)	✓	✓	✗	✓	Étiqueté Attribué Multiple	✗	✓	✓	✗	Movielen-10M	taux de compression : 93.75%

TABLE 2.2 – Synthèse des méthodes de compression par k^2-trees.

2.4 Compression par extraction de motifs

Les motifs fréquents sont des connaissances extraites sur des données. Leur but est de fournir à l'utilisateur des informations non triviales, implicites, présumées non connues. Ils lui offrent ainsi une meilleure appréhension des données. Dès lors, l'extraction de motifs fréquents est devenue une tâche importante dans la fouille de données et un thème très étudié par la communauté. Elle a aussi été vastement utilisée dans le domaine de compression des graphes vu qu'elle permet de ne garder que l'information utile et d'éliminer les redondances de manière efficace. En effet, nous trouvons plusieurs méthodes basées sur ce principe où nous pourrons clairement distinguer deux grandes classes : (i) les méthodes de compression basées vocabulaire (ii) les méthodes de compression basées agrégation.

Dans cette section, nous allons expliquer le principe de base de chaque classe et nous allons subdiviser chacune en plusieurs sous-classes en se basant sur ce dernier.

2.4.1 Compression basée vocabulaire

Les méthodes de compression par extraction de motifs basées vocabulaire sont des méthodes qui ont attirées l'attention des chercheurs ces dernières années car elles permettent une meilleure compréhension du graphe. Elles partent toujours d'un ensemble de structures prédéfinies qui ont été prouvées fréquentes dans les graphes réels. Deux sous classes de cette dernières peuvent être identifiées :

Les Méthodes basées sur des techniques de clustering

Les méthodes de cette classe s'appuient sur le fait qu'on ne peut pas comprendre facilement les graphes denses, alors que quelques structures simples sont beaucoup plus faciles à comprendre et souvent très utiles pour analyser le graphe. Elles se basent sur des algorithmes de détection de communautés. La question suivante peut alors se poser : pourquoi ne pas appliquer l'un des nombreux algorithmes de détection de communautés ou de partitionnement de graphes pour compresser le graphe en termes de communautés ? La réponse est que ces algorithmes ne servent pas tout à fait le même objectif que la compression. Généralement, ils détectent de nombreuses communautés sans ordre explicite, de sorte qu'une procédure de sélection des sous-graphes les plus «importants» est toujours nécessaire. En plus de cela, ces méthodes renvoient simplement les communautés découvertes, sans les caractériser (par exemple, clique, étoile) et ne permettent donc pas à l'utilisateur de mieux comprendre les propriétés du graphe.

Une première technique de compression usant de ces méthodes s'intitule Vocabulary Graph (VOG) (Koutra et al., 2015). C'est une méthode de base sur laquelle s'appuient toutes les autres méthodes de cette classe. Elle permet de compresser un graphe statique non orienté G à l'aide d'un vocabulaire de sous-structures qui apparaissent fréquemment dans les graphes réels et ayant une signification sémantique, tout en minimisant le cout du codage en utilisant le principe de longueur de description minimale (MDL) [13]. Le vocabulaire Ω utilisé est composé de six structures qui sont : clique (*fc*) et quasi-clique (*nc*), noyau bipartie (*cb*) et quasi-noyau bipartie

13. MDL est un concept de la théorie de l'information permettant de trouver le modèle ayant une longueur minimale : $min(D,M) = L(M) + L(D \mid M)$ où $L(M)$ est la longueur du modèle et $L(D \mid M)$ est la longueur en bits de la description des donnés en utilisant le modèle M.

(*nb*), étoile (*st*) et chaine (*ch*). On peut avoir un chevauchement au niveau des nœuds, les liens quand à eux sont pris selon un ordre FIFO et ne peuvent pas se chevaucher, i.e, la première structure $s \in M$ qui décrit l'arête dans A détermine sa valeur.

On note par C_x l'ensemble de tous les sous-graphes possibles de type $x \in \Omega$, et C l'union de tous ces ensembles, $C = \cup_x C_x$. La famille de modèles noté $\mathcal{M}$ représente toutes les permutations possibles des éléments de C. Par MDL, on cherche $M \in \mathcal{M}$ qui minimise le mieux le cout de stockage du modèle et de la matrice d'adjacence. En d'autre terme, VOG formule le problème de compression comme un problème d'optimisation dont la fonction objective est :

$min(D,M)$ = L(M) + L(E) avec E= A $\bigoplus$ **M** représentant l'erreur.

Pour l'encodage du modèle, on a pour chaque $M \in \mathcal{M}$:

$$\mathrm{L}(M) = L_{\mathbb{N}}(|\mathrm{M}|+1) + \log \begin{pmatrix} |M| + |\Omega| - 1 \\ |\Omega| - 1 \end{pmatrix} + \sum_{s \in M} (-logPr(x(s)|M) + L(s))$$

Le premier terme représente le nombre de structures dans le modèle , le second terme encode le nombre de structures par type $x \in \Omega$ tant dis que le troisième terme permet pour chaque structure $s \in M$, d'encoder son type x(s) avec un code de préfixe optimal et d'encoder sa structure. Le codage des structures se fait selon leurs types :

Clique : Pour l'encodage d'une clique, on calcule le nombre de nœuds de celle-ci, et on encode leurs ids : L(fc) = $L_{\mathbb{N}}(|fc|) + \log \binom{n}{|f_c|}$

Quasi-Clique : Les quasi-cliques sont encodées comme des cliques complètes, tout en identifiant les arêtes ajoutées dont le nombre est $||nc||$ et manquantes dont le nombre est noté $||nc||$' en utilisant des codes de préfixe optimaux : L(nc) = $L_{\mathbb{N}}(|nc|) + \log \binom{n}{|n_c|} + \log(|nc|)) + ||nc||l_1 + ||nc||'l_0$ où l_1 = - log($||nc||/(||nc||+||nc||')$) et analogique à l_0 sont les longueurs des codes de préfixe optimaux des arêtes ajoutées et manquantes.

Noyau biparti : notant par A et B les deux ensembles du noyau bipartie. On encode leurs tailles, ainsi que les identifiants de leurs sommets : L(fb) = $L_{\mathbb{N}}(|A|) + L_{\mathbb{N}}(|B|) + \log \binom{n}{|A|} + \log \binom{n-|A|}{|B|}$.

Quasi-Noyau biparti : Comme les quasi-cliques, les quasi-noyaux bipartie sont codés comme suit : L(nb) = $L_{\mathbb{N}}(|A|) + L_{\mathbb{N}}(|B|) + \log \binom{n}{|A|} + \log \binom{n-|A|}{|B|}$+log(|area($nb$)|) + $||nb||l_1 + ||nb||'l_0$.

Étoile : L'étoile est un cas particulier d'un noyau bipartie. D'abord on calcule le nombre de nœuds des extrémités de l'étoile, ensuite on identifie le nœud central parmi les n sommets et les nœuds des extrémités parmi les n-1 restants. L(st) = $L_{\mathbb{N}}(|st|\text{-}1)$ + log n + log $\binom{n-1}{|st|-1}$.

Chaine : On calcule d'abord le nombre d'éléments de la chaine, ensuite on encode les identifiants des nœuds selon leurs ordre dans la chaine : L(ch) = $L_{\mathbb{N}}(|ch|\text{ - }1) + \sum_{i=0}^{|ch|}(n-i)$

Matrice d'erreur : la matrice d'erreur E est encodée en deux parties E^+ et E^-. E^+ correspond à la partie de A que M modélise en rajoutant des liens non existants contrairement à E^- qui représente les parties de A que M ne modélise pas. Notons que les quasi-cliques et les quasi-noyaux bipartie ne sont pas inclut dans la matrice d'erreur puisqu'ils sont encodés avec exactitude. Le codage de E^+ et E^- est similaire à celui des quasi-cliques, on a :

$$\mathrm{L}(E^+) = \log(|E^+|) + ||E^+||l_1 + ||E^+||'l_0$$
$$\mathrm{L}(E^-) = \log(|E^-|) + ||E^-||l_1 + ||E^-||'l_0$$

Pour la recherche du meilleur modèle $M \in \mathcal{M}$, VOG procède sur trois étapes :

1. **Génération des sous-structures** : Dans cette phase, Les méthodes de détection de communautés et de clustering sont utilisées pour décomposer le graphe en sous-graphes pas forcément disjoints. La méthode de décomposition utilisée dans VOG est SlashBurn.
2. **Étiquetage des sous-graphes** : L'algorithme cherche pour chaque sous-graphe généré dans l'étape précédente la structure $x \in \Omega$ qui le décrit le mieux, en tolérant un certain seuil d'erreur.
   a **Étiquetage des structures parfaites** : Tout d'abord, le sous-graphe est testé pour une similarité sans erreur par rapport aux structures complètes du vocabulaire :
      - si tous les sommets d'un sous graphe d'ordre n ont un degré égale à n-1, il s'agit alors d'une clique.
      - si tous les sommets ont un degré de 2 sauf deux sommets ayant le degré 1, le sous-graphe est une chaine.
      - si les amplitudes de ses valeurs propres maximales et minimales sont égales, le sous-graphe est un noyau bipartie où les sommet de chaque ensemble du noyau sont identifiés à travers un parcours Breadth First Search (BFS) [14] avec coloration des sommets.
      - Quant à l'étoile, elle est considérée comme un cas particulier d'un noyau bipartie, il suffit donc que l'un des ensembles soit composé d'un seule sommet.

   b **Étiquetage des structures approximatives** : Si le sous graphe ne correspond pas à une structure complète, on cherche la structure qui l'approxime le mieux en terme du principe MDL.

   Après avoir représenté le sous graphe sous forme d'une structure, on l'ajoute à l'ensemble des structures candidates C, en l'associant à son cout.
3. **Assemblage du modèle** : Dans cette dernière étape, une sélection d'un ensemble de structures parmi ceux de C est réalisée. Des heuristiques de sélections sont utilisées car le nombre de permutations est très grand ce qui implique des calculs exhaustifs. Les heuristiques permettent d'avoir des résultats approximatifs et rapides, parmi les heuristiques utilisées dans VOG on trouve :
   — PLAIN : Cette heuristique retourne toutes les structures candidates, e.i, $M = C$.
   — TOP-K : Cette heuristique sélectionne les k meilleurs candidats en fonction de leur gain en bits.
   — GREEDY'N FORGET(GNF) : Parcourt structure par structure dans l'ensemble C ordonné selon la qualité (gain en bits), ajoute la structure au modèle tant qu'elle n'augmente pas le cout total de la représentation, sinon l'ignore.

Comme nous l'avons déjà précisé, VOG formule le problème de compression de graphe en tant que problème d'optimisation basé sur la théorie de l'information, l'objectif étant de rechercher les sous structures qui minimisent la longueur de description globale du graphe. Un

14. BFS : est un algorithme de parcours en largeur.

élément clé de VOG est la méthode de décomposition utilisée qui peut donner en sortie des sous-graphes ayant des nœuds et/ou des arêtes en commun et dont VOG (Koutra et al., 2015) ne suppose que le premiers cas. En partant de ce constat, les auteurs de (Liu et al., 2015) proposent VoG-overlapp, une extension de VOG prenant en compte les chevauchements des structures sous forme d'une étude expérimentale de l'effet de diverses méthodes de décomposition sur la qualité de la compression.

L'idée de base de VoG-overlapp est d'inclure une pénalité pour les chevauchements importants dans la fonction objective ce qui oriente le processus de sélection des structures vers la sortie souhaitée. Elle devient alors :

$$min\ L(G, M) = min\ \{L(M) + L(E) + \mathbf{L(O)}\}$$

Le principe de calcul de $L(M)$ *et* $L(E)$ demeure le même, avec O une matrice cumulant le nombre de fois que chacune des arêtes a été couverte par le modèle. Le cout du codage de la matrice des chevauchements est donné par la formule (2.1).

$$L(O) = log(|O|) \ + \ ||O||\ l_1 \ + \ ||O||'\ l_0 \ + \ \sum_{o \in \varepsilon(O)} L_N(|o|) \tag{2.1}$$

Où :

- $|O|$ est le nombre d'arêtes (distinctes) qui se répètent dans le modèles M.
- $||O||$ et $||O||'$ représentent respectivement le nombre d'arêtes présentes et manquantes dans O.
- $l_1 = -log(\frac{||O||}{||O||+||O||'})$, de manière analogique l_0, sont les longueurs des codes de préfixe optimaux pour les arêtes actuelles et manquantes, respectivement.
- $\varepsilon(O)$ est l'ensemble des entrées non nulles dans la matrice O.

Durant la même année, Shah et al.(Shah et al., 2015) ont proposé une autre variation de VoG, TimeCrunch, pour le cas des graphes simples (sans boucles) non orientés dynamiques représentés par un ensemble de graphes associés chacun à un timestamp. En d'autres termes, ils considèrent les graphes $G = \bigcup_{t_i} G_{t_i}(V, E_{t_i}) \ \ 1 \leq i \leq t$ où G_{t_i} = G a l'instant t_i. Un nouveau vocabulaire est proposé pour décrire proprement l'évolution des sous-structures dans le temps. En effet, ils partent du même vocabulaire de structures statiques $\Omega = \{st\ (etoile),\ fc(clique),\ nc(quasi - clique),\ bc(bipartie),\ nb(quasi - bipartie),\ ch(chaine)\}$ dont ils affectent une signature temporelle $\delta \in \Delta$ où : Δ = {o(oneshot), r(ranged), p(periodique), f(flickering), c(constante) }.

Comme les éléments du modèle sont modifiés, son cout est alors aussi modifié pour inclure pour chaque structure s non seulement sa connectivité $c(s)$ correspondant aux arêtes des zones induites par s mais aussi sa présence temporelle $u(s)$ correspondant aux timestamps dans lesquels s apparait dans le graphe G.

$L(M) = L_N(|M| + 1) + \log\binom{|M|+|\Phi|-1}{|\Phi|-1} + \sum_{s \in M}(logP(v(s)|M) + L(c(s)) + \mathbf{L(u(s))})$

Le cout de l'encodage de la présence temporelle diffère selon ses caractéristiques. Nous présenterons dans ce qui suit la formule correspondant à chaque signature.

- **Oneshot** : cette signature décrit les sous-structures qui apparaissent dans un seul timestamp, i.e, $|u(s)| = 1$. Donc le cout de l'encodage se réduit aux nombre de bits nécessaires pour sauvegarder le timestamp : $L(u(s)) = log(t)$.
- **Ranged** : dans ce cas la sous-structure apparait dans tous les graphes se trouvant entre deux timestamps t_{debut} et t_{fin}. Le cout englobe le nombre de timestamps dans lesquels elle apparait ainsi que les identifiants des deux timestamp t_{debut} et t_{fin} : $L(u(s)) = L_N(|u(s)|) + log\binom{t}{2}$.
- **Periodic** : cette catégorie est une extension de la précédente d'où : $L(p) = L(r)$. En effet, la périodicité peut être déduite à partir des marqueurs début et de fin ainsi que du nombre de pas de temps $|u(s)|$ entre chaque deux timestamps, permettant ainsi de reconstruire $u(s)$
- **Flickering** : ce type décrit les structures qui apparaissent dans n timestamps de manière aléatoire. Le coût doit englober donc le nombre de timestamps ainsi que leurs identifiants d'où : $L(u(s)) = L_N(|u(s)|) + log\binom{t}{|u(s)|}$.
- **Constant** : dans ce cas la sous-structure apparait dans tout les timestamps et donc elle ne dépend pas du temps d'où L(c)=0.

Nous notons que décrire $u(s)$ est encore un autre problème de sélection de modèle pour lequel les auteurs tirent parti du principe MDL . En effet, juste comme pour le codage de la connectivité, il peut ne pas être précis avec une signature temporelle donnée. Toutefois, toute approximation entraînera des coûts supplémentaires pour l'encodage de l'erreur qui englobent dans ce cas l'erreur de l'encodage de la connectivité ainsi que l'erreur de l'encodage de la signature temporelle.

Algorithme 1 : TIMECRUNCH

1: **Génération des sous-structures candidates :** Génération de sous-graphes pour chaque G_{t_i} en utilisant un des algorithmes de décomposition de graphes statiques.
2: **Étiquetage des sous-structures candidates :** Associer chaque sous-structure à une étiquette $x \in \Omega$ minimisant son MDL .
3: **Assemblage des sous-structures candidates temporelles :** Assembler les sous-structures des graphes G_{t_i} pour former des structures temporelles avec un comportement de connectivité cohérent et les étiqueter conformément en minimisant le coût de codage de la présence temporelle. Enregistrer le jeu de candidats $C_x \in C$.
4: **Composition du graphe compressé :** Composition du modèle M d'importantes structures temporelles non redondantes qui résument G à l'aide des méthodes heuristiques VANILLA, TOP-10, TOP-100 et STEPWISE. Choisir M associé à l'heuristique qui génère le coût de codage total le plus faible.

Une dernière variante, s'intitulant CONditional Diversified Network Summarization (CONDENSE), a été présentée par Liu et al. (Liu et al., 2018b) où ils abordent efficacement trois contraintes principales des méthodes précédentes : (i) leurs dépendance à la méthode d'extraction de motifs (ii) l'incapacité de certaines à gérer les motifs qui se chevauchent (iii) leur dépendance vis-à-vis de l'ordre dans lequel les structures candidates sont considérées lors de la phase d'assemblage. En effet, pour résoudre le premier problème, ils combinent plusieurs méthodes d'extraction de

motifs ce qui améliore la qualité des structures candidates en dépit du temps d'exécution. Tant dis que pour répondre à la deuxième contrainte ils utilisent la fonction objective proposée dans (Liu et al., 2015). Arrivant à la dernière phase de l'algorithme, ils proposent quatre nouvelles heuristiques : (1) STEP : choisie les K meilleures structures, (2) STEP-P : partitionne le graphe et affecte chaque motif à la partition ayant un chevauchement maximal de nœuds avec lui. Ces partitions sont parcourues parallèlement pour ne prendre que la meilleure de toutes les structures dans chacune des partitions, (3) STEP-PA : amélioration de STEP-P en désignant chaque partition du graphe comme étant active, puis si une partition échoue x fois pour trouver une structure qui réduit le coût MDL , cette partition est déclarée inactive et n'est pas visitée dans les prochaines itérations, (4) K-STEP : combinaison des trois premières heuristiques. Il transforme par la suite chaque motif trouvé en un super-nœud.

Les méthodes basées sur les propriétés de la matrice d'adjacence

Les graphes peuvent avoir différentes représentations. Chacune des structures de données présente des avantages et des inconvénients en ce qui concerne la quantité de mémoire nécessaire pour stocker les données et la facilité d'accès aux données. Selon les besoins, il est parfois utile de stocker les données dans des structures de données plus grandes, qui nécessitent plus d'espace mais offrent un accès efficace aux données. En se basant sur ce constat plusieurs méthodes ont été proposées dans la littérature pour compresser la matrice d'adjacence en exploitant les propriétés des graphes réels pour trouver les motifs les plus fréquents dans cette dernière.

(Asano et al., 2008) ont exploité les propriétés du graphe du web pour présenter une nouvelle méthode de compression, appelée Efficient Compression of web graph (ECWG), sans perte permettant d'extraire les motifs à partir de la matrice d'adjacence. Ils proposent un vocabulaire composé de six types de blocs (Motifs) : un bloc horizontal de 1, un bloc vertical de 1, un bloc diagonal de 1, un rectangle de 1, un bloc de 1 sous forme de L et le singleton 1. Avant de procéder à l'extraction des motifs, la liste d'adjacence du graphe est partitionnée selon les domaines (ex : esi.dz, usthb.dz, ...). Une nouvelle matrice d'adjacence est donc construite pour chaque hôte(domaine) contenant les liens existants entre ses pages, auxquels les liens inter-hôtes sont concaténés. Les blocs B sont détectés par la suite et chacun est représenté par un quadruplets (i, j, type(B), dim(B)) où i,j représentent les coordonnées du premier élément du bloc dans la matrice d'adjacence de l'hôte, type(B) représente le type du bloc et dim(B) représente les dimensions du bloc (omis dans le cas du singleton).

Dans une méthode toute récente s'intitulant Graph Compression Using Pattern Matching (GCUPMT), Shah et Rushabh (Shah, 2018) partitionnent les lignes de la matrice d'adjacence en plusieurs blocs ayant la même taille des motifs qui sont dans ce cas sous forme de vecteurs prédéfinies. Les blocs sont comparés avec l'ensemble des motifs ce qui entraîne , en cas de correspondance , le remplacement du bloc par un indicateur du motif précédé par un 1 indiquant que les bits suivants appartiennent à un indicateur de motif. Dans le cas contraire, les données brutes sont stockées directement précédées par un 0.

Source	Destinations
0	1, 2, 315
1	0, 2, 3, 315, 316
2	3
3	1
4	314
...	...
314	4
315	0, 316, 317
316	1, 317
317	316

Source	Intra	Inter
315	316, 317	0
316	317	1
317	316	-

Source	Destinations
0	1, 2
1	2
2	1

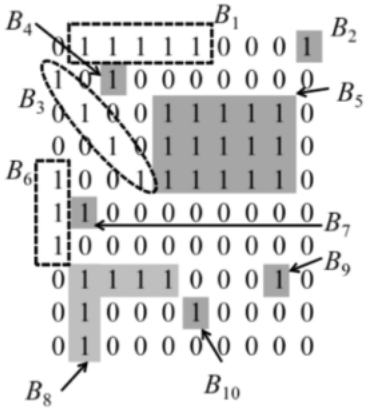


FIGURE 2.13 – Exemple illustrant le principe de fonctionnement (Asano et al., 2008)

2.4.2 Compression basée sur l'agrégation des motifs

Les méthodes de compression par extraction de motifs basées sur l'agrégation sont des méthodes qui agrègent plusieurs nœuds ou liens d'un motif en un seul nœud, appelé super-nœud. Le graphe en sortie, dit super-graphe, devient dès lors plus simple et moins complexe offrant ainsi une aisance et une facilité de traitement, d'exploration et de visualisation. Nous présenterons dans ce qui suit les deux sous-classes de cette classe qui se distinguent selon que l'agrégation concerne les nœuds ou les liens.

a. 2.4.2.1 Les méthodes de compression basées l'agrégation de nœuds

Les techniques de compression basées sur l'agrégation des nœuds des motifs sont des méthodes qui ont existé depuis plusieurs décennies offrant plusieurs avantages. Elles visent à résumer le graphe initial en agrégeant les nœuds des motifs découvert dans le but de diminuer le nombre de nœuds existants et d'offrir une meilleure visibilité et analyse du graphe.

Une première méthode de cette classe s'intitule Subdue (Ketkar et al., 2005). Elle effectue une recherche *Branch&Bound* qui commence à partir des sous-structures composées de tous les sommets avec des étiquettes uniques. Les sous-structures sont prolongées de toutes les manières possibles par un sommet et une arête ou par une arête afin de générer des sous-structures candidates. Subdue conserve les instances des sous-structures et utilise l'isomorphisme de graphe pour déterminer les instances de la sous-structure candidate. Les sous-structures sont ensuite évaluées en fonction de leur compression de la longueur de description (DL) du jeu de données. Cette procédure se répète jusqu'à ce que toutes les sous-structures soient prises en compte ou que les contraintes imposées par l'utilisateur ne soient plus vérifiées. A la fin de la procédure, Subdue indique les meilleures sous-structures de compression. Le système Subdue fournit également la possibilité d'utiliser la meilleure sous-structure trouvée lors d'une étape de découverte pour compresser le graphe d'entrée en remplaçant les instances de la sous-structure par un seul sommet et en effectuant le processus de découverte sur le compressé. Cette fonctionna-

lité génère une description hiérarchique du jeu de données du graphe à différents niveaux d'abstraction en termes de sous-structures découvertes.

Dans (Rossi and Zhou, 2018), les auteurs partent de l'observation que les graphes réels sont formés souvent de nombreuses cliques de grande taille. En utilisant ceci comme base, GraphZip décompose le graphe en un ensemble de grandes cliques, qui est ensuite utilisé pour compresser et représenter le graphe de manière succincte.

b. **2.4.2.2 Les méthodes de compression basées sur l'agrégation de liens**

Les méthodes de compression par extraction de motifs basées sur l'agrégation de liens sont parmi les méthode les plus populaires. Leur objectif est de produire un graphe compressé à partir du graphe initial en remplaçant les liens denses du graphe par un nouveau super-nœud. Elles se divisent selon le principe en deux grandes classes : celles utilisant les règles de grammaire et celles utilisant des heuristiques de clustering. Nous détaillerons dans ce qui suit ces deux classes.

b.1. **Les méthodes de compression basées sur les règles de grammaire** La classe des méthodes de compression basées sur les règles de grammaire est une généralisation d'une méthode de compression des dictionnaires s'intitulant Re-pair. Son principe de base consiste en la recherche, à chaque itération, de la paire de symboles la plus fréquente dans une séquence de caractères et de la remplacer par un nouveau symbole, jusqu'à ce qu'il ne soit plus commode de les remplacer. Nous notons que dans ce cas le motif est sous forme de deux arêtes ayant un sommet en commun, nommé *digraph*.

Une première méthode suivant ce principe a été proposée dans (Claude and Navarro, 2010b) baptisée *Approximate Re-pair*. Dans cette méthode un graphe G=(V,E) est représenté sous forme d'une sequence de caractères T :

$$\text{T=T(G)= } \overline{v_1}\ v_{1,1}\ \ldots\ v_{1,a}\ \overline{v_2}\ v_{2,1}\ \ldots\ v_{2,a_2}\ldots\overline{v_n}\ v_{n,1}\ \ldots\ v_{n,a_n}$$

où $\overline{v_i}$ représente l'identificateur du sommet v_i. Elle procède en trois étapes essentielles expliquer dans l'algorithme **2** . Lorsqu'il n'y a plus de paires à remplacer,

Algorithme 2 : Approximate Re-pair

1: **Calcule des fréquences :** T est parcourue séquentiellement et chaque pair $t_i t_{i+1}$ est ajoutée à un tableau de hachage H avec leur nombre d'occurrences.
2: **Recherche des k meilleurs paires :** H est parcourue et les k paires les plus fréquentes sont retenues, en utilisant k pointeurs vers les cellules de H.
3: **Le remplacement simultané :** les k paires identifiées dans l'étape précédente sont simultanément remplacées par un nouveau identifiant et une règle de production est ajoutée.

Approximate Re-pair s'arrête donnant comme résultat un représentation compacte C de la chaine T. Pour finaliser le processus, tous les indicateurs de nœuds $\overline{v_i}$ seront

supprimés de C. De plus, l'algorithme crée une table qui contiendra des pointeurs vers le début de la liste d'adjacence de chaque nœud dans C. Grâce à cette table l'algorithme pourra répondre aux requêtes de recherche de successeurs en un temps optimal.

Dans un travail ultérieur (Claude and Navarro, 2010a) , les même auteurs s'intéressent aux requêtes de recherche des nœuds prédécesseurs et successeurs à partir du graphe compressé de *Approximate Re-pair* directement. Il proposent alors de combiner leur méthode avec une représentation basée sur les relations binaires de (Barbay et al., 2006). En effet, ce dernier consiste à représenter les listes d'adjacences à l'aide d'une représentation séquentielle permettant de rechercher les occurrences d'un symbole puis de rechercher les voisins inverses à l'aide de cette primitive.

Claude et Ladra (Claude and Ladra, 2011) partagaient les mêmes préoccupations des auteurs de la méthode précédente et ont proposé comme solution de combiner la méthode Re-pair avec la représentation k2-tree. Ils obtiennent alors une compression de 2,27 (pbe) sur le graphe UK2002, tout en conservant la possibilité d'interroger les voisins entrants et sortants (Maneth and Peternek, 2015).

Une dernière méthode de cette classe s'instituant gRepair a été proposée dans (Maneth and Peternek, 2018). Ce nouveau algorithme de compression détecte de manière récursive des sous-structures répétées et les représente via des règles de grammaire. Des requêtes spécifiques telles que l'accessibilité entre deux nœuds ou des requêtes de chemin normal peuvent ainsi être évaluées en temps linéaire (ou en temps quadratique, respectivement), sur la grammaire, permettant ainsi des accélérations proportionnelles au taux de compression. La figure 2.14 présente le résultat de cette méthode sur un exemple.

b.2. **Les méthodes de compression basées sur des heuristiques de clustering**

Les méthodes de compression appartenant à cette classe sont des méthodes basées sur la recherche des sous-graphes denses (ayant des nœuds fortement connectés). Elles sont destinées principalement aux graphes du Web et les graphes des réseaux sociaux dans le but de faciliter leur exploration et analyse.

(Buehrer and Chellapilla, 2008) ont exploité l'existence de plusieurs ensembles de pages web qui ont les mêmes liens sortants. S'intitulant Virtual Node Miner (VNM), leur approche est basée sur la réduction du nombre de liens en créant des nouveaux sommets virtuels qui sont ajoutés au graphe. Soit G = (V,E) un graphe orienté, l'algorithme proposé se compose de deux phases essentielles :

i. **Phase de Clustering :**

Le but de cette première étape est de contourner la tâche presque impossible d'extraction simultanée de centaines de millions de points de données en groupant d'abord les sommets similaires dans le graphes dans des clusters. Pour cela k fonctions de hachage indépendantes sont utilisées pour obtenir une matrice de taille V * k. Par la suite, les lignes de la matrice sont triées lexicographiquement et elle est parcourue colonne par colonne en regroupant les lignes ayant la même

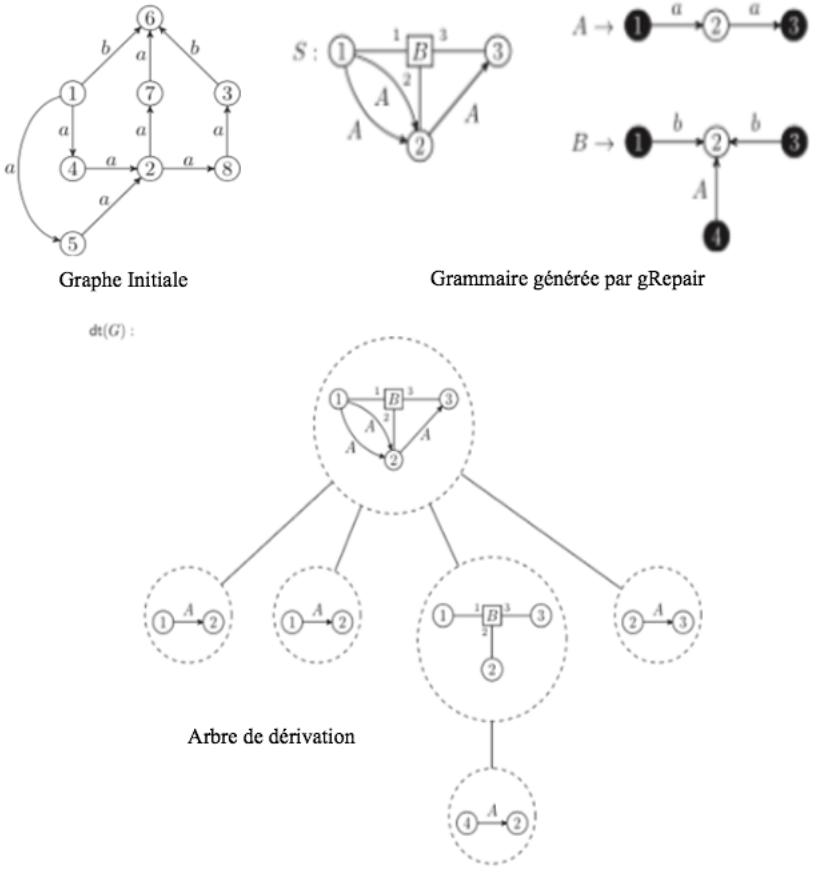


FIGURE 2.14 – Exemple d'exécution de gRepair sur G.

valeur. Lorsque le nombre total de lignes chute au-dessous d'un seuil ou que le bord de la matrice de hachage est atteint, les identifiants des sommets associés aux lignes sont renvoyés au processus d'extraction (Phase 02).

ii. **Phase d'Extraction de Motifs :** Le but de cette étape est de localiser des sous-ensembles communs de liens sortants dans les sommets donnés. Ainsi les ensembles plus grands et fréquents présentent un intérêt, car ils peuvent représenter des motifs plus pertinents et une meilleure compression. En effet, les performances de compression d'un motif sont calculés en fonction de sa fréquence dans la liste d'adjacence, et de sa taille qui est le nombre de liens qu'il contient (Formule 2.2).

$$Compression(P) = (P.frquence - 1)(P.taille - 1) - 1 \tag{2.2}$$

Afin d'extraire ces motifs, VNM utilise une heuristique gloutonne. Cette heuristique procède comme suit :

A. Extraire un histogramme des identifiants de liaison sortante à partir de la liste d'adjacence des sommets données.

B. Les listes sont réorganisées dans l'ordre décroissant des fréquences des liens sortants en éliminant ceux qui apparaissent une seul fois uniquement.

C. Chaque lien sortant est ajouté à un arbre de préfixes avec l'ensemble trié de ces extrémités initiales selon leurs identifiants.

D. L'arbre est par la suite parcouru afin d'identifier les motifs qui maximisent la formule de performance de la compression (Formule 2.2). Ces motifs sont ensuite convertis en nœuds virtuels et les identificateurs de sommets de leurs listes sont remplacés par les ids des nœuds virtuels dans la liste d'adjacence.

Une variante de VNM , Dense SubGraph Mining (DSM), a été proposée par Hernandez et Navarro (Hernández and Navarro, 2014). Comme première contribution, ils augmentent les types de structures découvertes dans la phase de clustering pour englober aussi : les cliques, les bi-cliques. L'extraction de motifs cette fois-ci n'est pas basée sur un parcours des feuilles vers la racine mais l'inverse où l'ensemble des sommets finaux des liens du motifs est constitué des étiquettes des nœuds de l'arbre inclues dans le chemin de la racine vers la feuille et les sommets initiaux sont la liste des sommets inclus dans le nœud feuille. Leur deuxième contribution consiste en une hybridation dans le but de représenter le graphe en sortie à l'aide de structures compactes. Une première approche proposée est d'utiliser les k2-trees (Brisaboa et al., 2009) qui donnent la représentation la plus compacte. La deuxième hybridation consiste en une nouvelle structure proposée par les auteurs.

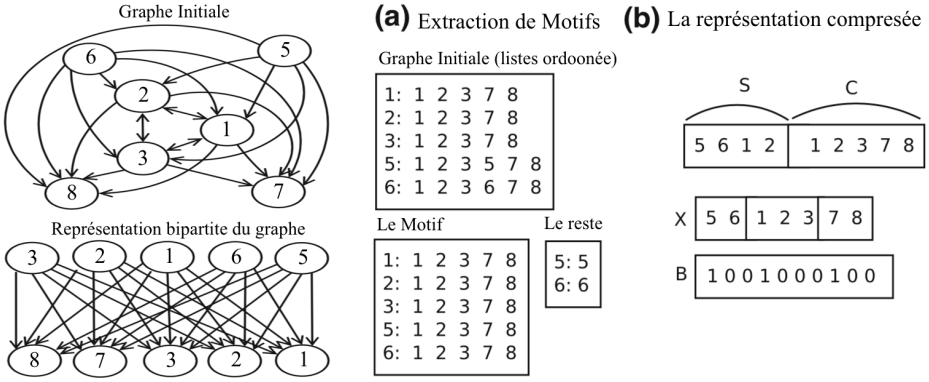


FIGURE 2.15 – Exemple d'exécution de DSM

2.4.3 Synthèse des méthodes de compression par extraction de motifs

Durant les sections précédentes nous avons expliqué de manière générale les fondements de base de chaque sous-classe de la classe des méthodes de compression par extraction de motifs ainsi que le principe de fonctionnement de leurs méthodes.

Nous regroupant dans le tableau 2.3 les différentes caractéristiques des méthodes de ces sous-classes. Nous observons qu'elles sont toutes sans perte et destinées aux graphes statiques, à l'exception de la classe des méthodes de compression Basées vocabulaire en utilisant les méthodes de clustering qui contiennent des méthodes de compression supportant les graphes dynamiques. On remarque aussi que la majorité des méthodes de compression par extraction de motifs fournissent en sortie une représentation succincte. Le tableau 2.3 contient aussi un résumé sur les résultats de l'application de ces méthodes sur des graphes réels.

Classe	Méthode	Graphe en entrée				Compression		Structure en sortie		Graphe de test	Résultat
		Orienté	Non orienté	Statique	Dynamique	Avec perte	Sans perte	Succincte	Structurelle		
Basée vocabulaire en utilisant les méthodes de clustering	VoG (Koutra et al., 2015)	✗	✓	✓	✗	✗	✓	✓	✗	Enron	75%
	VoG-Overlapp (Liu et al., 2015)	✗	✓	✓	✗	✗	✓	✓	✗	Enron	75%
	TimeCrunch (Shah et al., 2015)	✗	✗	✓	✓	✗	✓	✓	✗	Enron	74%
	CanDenSe (Liu et al., 2018b)	✗	✓	✗	✓	✗	✓	✓	✓	Enron	78%
Basée vocabulaire en utilisant les propriétés de la matrice d'adjacence	ECWG (Asano et al., 2008)	✓	✓	✓	✗	✗	✓	✓	✗	uk-2002	76.1%
	GCUPMT (Shah, 2018)	✓	✓	✓	✗	✗	✓	✓	✗	graphes 8192 nœuds	70%
Basée Agrégation de nœuds des motifs	Subdue (Ketkar et al., 2005)	✗	✓	✓	✗	✗	✓	✗	✓	Graphe des composantes chimiques	16%
	GraphZip (Rossi and Zhou, 2018)	✗	✓	✓	✗	✗	✓	✗	✓	Web-Google	19%
Basée Agrégation de liens des motifs en utilisant les règles de grammaire	Approximate Re-pair (Claude and Navarro, 2010b)	✓	✗	✓	✗	✗	✓	✗	✓	uk-2002	4.23
	Approximate Re-pair (Claude and Navarro, 2010a)	✓	✗	✓	✗	✗	✓	✗	✓	uk-2002	3.98 bpe
	gRe-pair (Maneth and Peternek, 2018)	✓	✓	✓	✗	✗	✓	✗	✓	NotreDame	4.84 bpe
Basée Agrégation de liens des motifs et les méthodes de clustering	VNM (Buehrer and Chellapilla, 2008)	✓	✗	✓	✗	✗	✓	✗	✓	uk-2002	1.95 bpe
	DSM (Hernández and Navarro, 2014)	✓	✗	✓	✗	✗	✓	✓	✓	uk-2002	1.53 bpe

TABLE 2.3 – Synthèse des méthodes de compression par extraction de motifs.

2.5 Conclusion

De nos jours, les graphes sont omniprésents. Cependant, leur taille présente un obstacle presque insurmontable à la compréhension du caractère essentiel des données. D'où la nécessité de la compression qui permet de réduire la taille des graphes tout en gardant le caractère utile de l'information incluse dans ces derniers.

Nous avons étudié dans ce chapitre différentes méthodes de compression de graphes dans le but d'établir une classification des méthodes basée sur deux approches : l'extraction de motifs et les arbres k2-trees. Nous nous sommes appuyer pour cela sur le principe de fonctionnement de chacune d'elles. Nous proposons six classes de méthodes :

i Les méthodes de compression par les k2-trees

ii Les méthodes de compression par extraction de motifs basées vocabulaires en utilisant des méthodes de clustering.

iii Les méthodes de compression par extraction de motifs basées vocabulaires exploitant les propriétés de la matrice d'adjacence.

iv Les méthodes de compression par extraction de motifs basées agrégation de nœuds.

v Les méthodes de compression par extraction de motifs basées agrégation de liens en utilisant des règles de grammaire.

vi Les méthodes de compression par extraction de motifs basées agrégation de liens en utilisant des heuristiques de clustering.

Nous avons présenté le principe de fonctionnement des méthodes relatif à chaque classe tout en comparant leurs caractéristiques. Ces classes diffèrent les unes des autres dans leurs fondements théoriques, le type de compression considéré, la complexité des algorithmes, les objectifs et les domaines d'application. Dans le tableau B.1 , nous allons essayer de synthétiser les principales différences et similitudes entre les deux approches, compression par extraction de motifs et compression par les arbres k2-trees, que nous avons pu constater à travers notre recherche bibliographique. Ils ont un fort impact sur le choix de la méthode de compression du moment qu'ils ont une influence directe sur les performances.

	k^2-trees	**Extraction de motifs**
Type de compression	toujours sans perte	toujours sans perte
Structure en sortie	Toujours succincte	Peut être succincte où structurelle où les deux en même temps
technique utilisée	exploitation de la matrice d'adjacence du graphe	exploitation des motifs fréquents dans le graphe
Dépendance	Dépend du paramètre k	Dépend selon la méthode de l'algorithme de clustering ou du vocabulaire de motifs utilisé
Objectif	Compression, Réduire l'espace de stockage et le temps de parcours	- Compression, - Réduire l'espace de stockage et le temps de parcours, - Extraire les informations pertinentes, - Visualisation
Domaine d'application	tous les domaines	tous les domaines
Type de graphes supporté	- Statique orienté, - Attribué, - Dynamique,	- Statique (orienté et non orienté), - étiqueté, - Dynamique,

TABLE 2.4 – Comparaison entre les méthodes basées surk^2-trees et basées sur l'extraction de motifs.

Deuxième partie

Contribution

Chapitre 3

Conception

3.1 Introduction

La réalisation de l'étude bibliographique dans le précédent chapitre, nous a initié au domaine de compression en général, et nous a permis d'approfondir nos connaissances dans le domaine de compression des graphes. Nous avons pu voir en détails les caractéristiques de plusieurs méthodes de compression s'inscrivant dans les deux classes de méthodes de compression : celles basées sur l'extraction de motifs et celles basées sur les arbres k^2-trees. La vocation de ce travail est d'enrichir un projet existant en l'étendant avec deux moteurs de compression qui regroupent différentes stratégies des deux classes étudiées. La réalisation de ces deux moteurs permettra de comparer entre ces stratégies et d'avoir une idée plus claire sur leurs performances dans différents scénarios.

Dans ce chapitre, nous présenterons les détails de la conception des deux moteurs. Nous allons, dans un premier temps, présenter leur principe de fonctionnement tout en mettant l'accent sur les différents modules qui les constituent. Nous expliquerons juste après notre deuxième contribution qui consiste en une nouvelle méthode de compression destinée aux graphes dynamiques.

3.2 k^2-GraCE :

Dans cette section, nous présenterons notre premier moteur baptisé k^2-GraCE. Il consiste en un moteur de compression des graphes par les arbres k^2-trees qui exploitent les propriétés de localité et de similarité dans les graphes du web.

3.2.1 Principe de fonctionnement :

Le moteur k^2-GraCE a été conçu pour permettre la compression de graphes statiques ou dynamiques et orientés ou non orientés. Il se base sur les travaux de Brisaboa et al. (Brisaboa et al., 2009). Il offre la possibilité de construire le graphe compressé à partir de la matrice d'adjacence, de la liste d'adjacence ou du graphe directement. Il permet aussi dans le cas des graphes dynamiques de réduire davantage la taille de l'arbre en le construisant non pas à partir

de la matrice d'adjacence initiale mais en calculant une matrice de différence entre les instants t_i.

k^2-GraCE est un moteur de compression sans perte de données. En effet, il construit en sortie un arbre k^2-tree incluant toute information présente dans le graphe initial. Cette information est représentée sous forme de deux chaines binaires. Le processus de compression d'un graphe de données par le moteur k^2-GraCE passe par les étapes suivantes :

1. Lecture et structuration du graphe de données en entrée
2. Pré-traitement : cette étape est optionnelle, elle ne figure pas dans l'algorithme de base.
3. Construction récursive de l'arbre k2-tree à partir de la matrice d'adjacence (ou de la liste d'adjacence ou du graphe directement) et la concaténation des différents niveaux de l'arbre dans une première chaine T, à l'exception du dernier niveau qui sera stocké dans une deuxième chaine L.
4. Écriture du graphe compressé sur le fichier de sortie.

Nous illustrons ces phases par la figure 3.1 qui donne une vue globale sur le principe de fonctionnement de ce moteur.

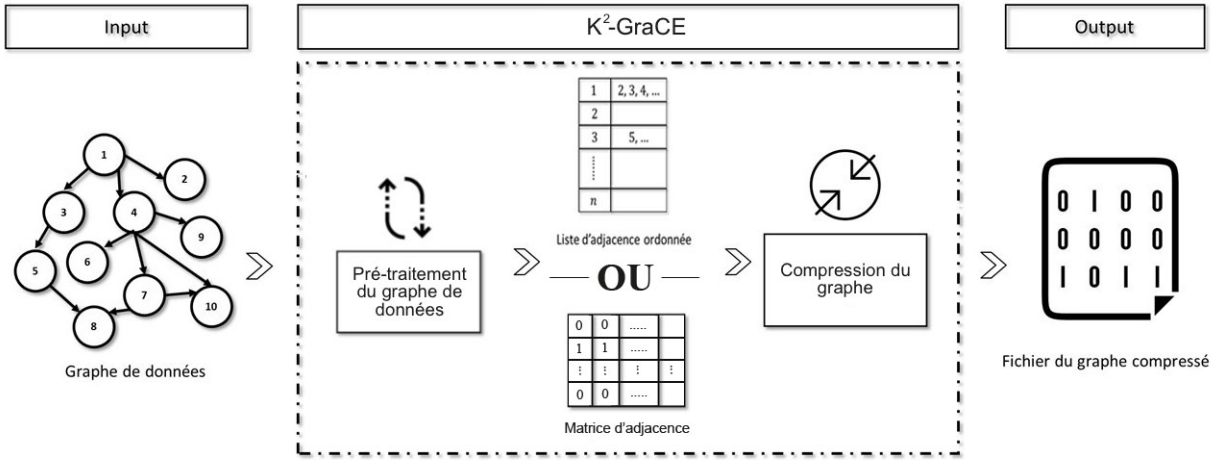


FIGURE 3.1 – Principe de fonctionnement du moteur k^2-GraCE.

Après avoir expliquer de manière générale le principe de fonctionnement du moteur k^2-GraCE, nous présenterons dans ce qui suit les détails des deux phases : la phase de pré-traitement et la phase de compression (construction de l'arbre). Nous allons tout d'abords commencer par présenter les notations et opérations de base qui seront utilisées par la suite. Nous enchainerons par les différents algorithmes de construction selon le type de graphe en entrée.

3.2.2 Paramètre et notations :

Paramètre	Signification
G	Graphe de données
M	Matrice d'adjacence du graphe G
$List$	Liste d'adjacence du graphe G
A	L'arbre k^2-tree
T	Le nombre d'instants dans lesquels le graphe a été capturé
h	La hauteur de l'arbre k^2-tree
N	Nombre de nœuds dans le graphe
k	Paramètre déterminant le nombre de fils dans l'arbre k^2-tree
:	opérateur de concaténation
rank(T, i)	Fonction calculant le nombre de 1 existant dans le tableau binaire T dans l'intervalle des indices $[1, i]$

TABLE 3.1 – Tableau des notations et paramètres du moteur k^2-GraCE.

3.2.3 Conception Modulaire :

Dans cette partie, nous détaillerons chaque phase du moteur k^2-GraCE. Nous commencerons par présenter les différentes techniques de pré-traitement que nous voulons utiliser avec notre moteur. Nous expliquerons par la suite le processus de compression pour chaque type de graphe supporté par le moteur k^2-GraCE.

3.2.3.1 Pré-traitement du graphe de données :

Durant cette première phase, le moteur k^2-GraCE utilise des stratégies qui permettrons d'aboutir à une meilleur compression. Il offre une alternative pour chaque type de graphe qu'il supporte.

Dans le cas des graphes statiques orientés, k^2-GraCE offre la possibilité de ré-ordonnoncer les nœuds du graphe de données G. Comme notre travail est une suite d'un travail d'étudiants de l'année précédente (W. GUERMAH, 2018), cette phase a été déjà conçue et implémentée. Nous rappellerons uniquement dans cette section le principe de base de chaque méthode.

- **Ordre Lexicographique :** Les nœuds sont ordonnés selon leurs listes de successeurs. Les listes de successeurs seront donc triées selon un ordre croissant des identifiants et par la suite ordonnées.
- **Ordre Gray :** Les nœuds sont permutés de telle sorte que deux nœuds dont l'ordre est successif diffèrent dans au plus un voisin.
- **Ordre DFS :** Les nœuds sont ordonnés selon leurs positions dans le parcours en largeur (DFS) du graphe.
- **Ordre BFS :** Les nœuds sont ordonnés selon leurs positions dans le parcours en profondeur (BFS) du graphe.
- **Ordre Aléatoire :** Des permutations aléatoires des nœuds sont établies.

Le deuxième type de graphe supporté par le moteur k^2-GraCE correspond aux graphes statiques non orientés. Dans ce cas, nous obtenons une matrice d'adjacence symétrique. De ce fait, nous proposons de ne considérer que la partie triangulaire supérieure pour enlever la redondance porté par la symétrie. L'arbre construit ainsi offre toujours la possibilité d'extraire le voisinage d'un nœuds sans avoir recours à une décompression. En effet, extraire les voisins d'un nœud dans le graphe initial revient à extraire les voisins directs et inverses dans la nouvelle matrice construite en utilisant les mêmes algorithmes de l'annexe A. Si nous prenons l'exemple du nœuds 2 dans la figure 3.2, ses voisins sont représentés par les 1 de la deuxième ligne ou de la deuxième colonne dans la matrice d'origine (matrice gauche) et par l'union des cellules ayant un 1 de la deuxième ligne et de la deuxième colonne dans la matrice droite ce qui nous donne : $v(2) = \{1, 2, 5, 6\}$.

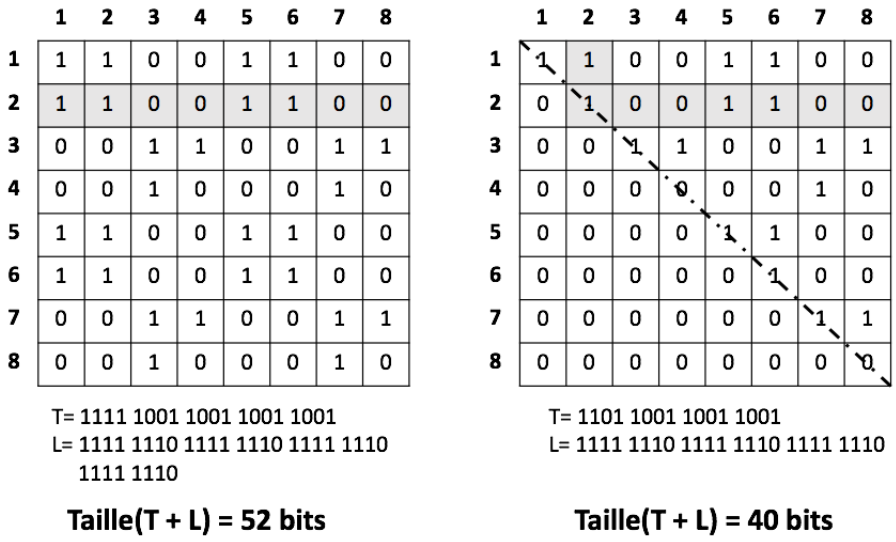


FIGURE 3.2 – Exemple d'arbre k^2-tree (k=2) pour un graphe non orienté.

k^2-GraCE supporte aussi les graphes dynamiques. Dans le cas de ce type de graphe, il offre la possibilité de calculer une matrice différence à partir de la matrice initiale. Le but de cette fonctionnalité est de maximiser les zones nulles dans la matrice afin de réduire la taille de l'arbre. À t=0, on garde la même matrice bidimensionnelle du graphe initial. Pour les instants restants ($t > 0$), nous comparons M_{pq} à l'instant t avec M_{pq} à l'instant *t-1*, si égalité alors la nouvelle matrice contiendra un 0 dans la cellule pq à l'instant t sinon elle contiendra un 1. La nouvelle matrice d'adjacence contiendra ainsi uniquement les changements qui occurrent entre les instants. La figure 3.3 montre une matrice d'adjacence d'un graphe dynamique capturée dans trois instants différents avec sa matrice de différence et la représentation k^2-tree dans les deux cas.

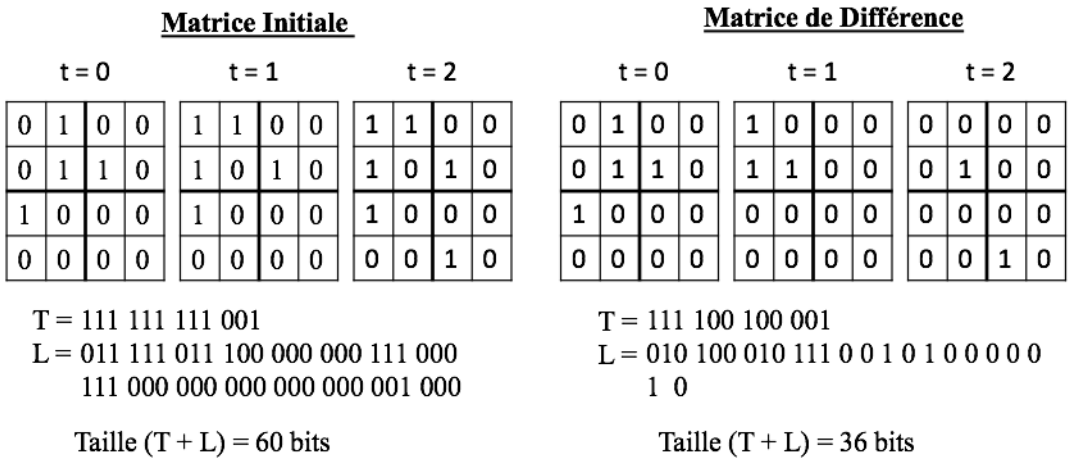


FIGURE 3.3 – Exemple d'arbre k^2-tree (k=2) pour la matrice de différence.

L'inconvénient de cette fonctionnalité réside dans le fait qu'une reconstruction de la matrice initiale est nécessaire dans le cas d'interrogation du graphe. Un autre inconvénient de cette technique est qu'elle dégrade les performances dans le cas où les différentes captures ne possèdent pas des arêtes en commun. Dans le cas de disparition de liens par exemple nous obtiendrons des zéros (0) qui vont être remplacés par des uns (1) engendrant ainsi une matrice plus dense. Le gain apporté par cette technique est fortement dépendant des caractéristiques du graphe. Nous fournissons ci-après l'algorithme de construction de la matrice de différence :

Algorithme 3 : ConstructDiffMatrice

Entrée :

- M[][][] : la matrice d'adjacence

Sortie : La matrice de Différence

```
1: MatriceDiff[][][]
2: MatriceDiff[1, :, :] = M[1, :, :]
3: pour i = 2...T faire
4:    pour j = 1...N faire
5:       pour k = 1...N faire
6:          si M[i][j][k] = M[i-1][j][k] alors
7:             MatriceDiff[i][j][k] = 0
8:          sinon
9:             MatriceDiff[i][j][k] = 1
10: Retourner MatriceDiff
```

Nous notons que l'utilisation de ce module dans le processus de construction de l'arbre k^2-tree n'est pas obligatoire. Cette phase ne figure pas dans l'algorithme de base. Nous voulons, à travers son intégration dans le moteur k^2-GraCE, rassembler davantage les nœuds ayant des voisins communs et donc les zones homogènes dans la matrice d'adjacence (zones plaines de 0 ou zones plaines de 1) ou augmenter le nombre de cellules nulles dans la matrice d'adjacence.

3.2.3.2 Construction de l'arbre k^2-tree

k^2-tree est une représentation compacte de la matrice d'adjacence qui exploite ses propriétés de similarité et de localité. Elle était destinée au départ pour les graphes du web et généralisée par la suite pour différents cas. La représentation est conçue pour compresser les grandes zones nulles de la matrice d'adjacence en les représentant avec un nombre réduit de bits. Plusieurs alternatives existent pour construire l'arbre k^2-tree : l'utilisation de la matrice d'adjacence, l'utilisation de la liste d'adjacence ou l'utilisation du graphe directement. Dans tous les cas, nous obtenons une représentation sous forme d'un arbre ayant une hauteur h= $log_k(N)$ où chaque nœud possède k^2 fils. Pour représenter l'arbre de manière concise, deux structures seront utilisées :

— **T** : Un tableau qui stocke tous les bits du k^2-tree sauf ceux du dernier niveau. Les bits de l'arbre k^2-tree sont placés après une traversée horizontale de l'arbre. k^2-GraCE représente d'abord les k^2 valeurs binaires des fils du nœud racine, puis les valeurs du deuxième niveau, ...etc.

— **L** : Un tableau stockant le dernier niveau de l'arbre. Ainsi, il représente la valeur des (ou de certaines) cellules de la matrice d'adjacence du graphe initial.

L'algorithme 4 résume les différentes étapes de construction de l'arbre k^2-tree dans le cas de la construction à partir de la liste d'adjacence. Nous dotons pour cela la liste d'adjacence de n curseurs, un par ligne, de sorte que chaque fois que nous devons accéder à M_{pq}, nous comparons le curseur actuel de la ligne p à la valeur q. S'ils sont égaux, alors on aura $M_{pq} = 1$ et nous devons avancer le curseur vers le nœud suivant de la liste de la ligne p. Sinon, nous saurons que $M_{pq} = 0$.

Nous supposons que la liste d'adjacence *List*, le nombre de fils k ainsi que l'arbre k^2-tree A sont des variables globales. Pour construire l'arbre à partir de la racine, elle sera invoquée comme suit : ConstructK2Tree(N,1,0,0). Après avoir construit l'arbre A sous forme d'un tableau de niveau, nous procèderons à la construction des deux structures selon les deux formules suivantes :

- T = $A_1 : A_2 : ... : A_{h-1}$
- L = A_h

Des versions itératives de l'algorithme peuvent être établies. Cependant, elles dégradent les performances car elles nécessitent soit l'utilisation de structures supplémentaires ou plus d'accès mémoire. Nous avons privilégié la version récursive car elle permet de construire l'arbre k^2-tree en un seul parcours de la liste d'adjacence *List*.

Algorithme 4 : ConstructK2Tree

Entrée :

- n : la taille de la sous-matrice
- l : le niveau de l'arbre
- p : l'indice ligne de début de la sous-matrice
- q : l'indice colonne de début de la sous-matrice

Sortie : La valeur du nœud de la sous-matrice

```
1:  C = ∅
2:  pour i = 1...k faire
3:      pour i = 1...k faire
4:          si l = log_k(N) alors
5:              // Condition selon le type de représentation en entrée
6:              // Matrice d'adjacence : M[p+i,q+j] = 1
7:              // Graphe : e_{p+i,q+j} ∈ E
8:              si List[p+i].courrant() = q+j alors
9:                  C = C : 1
10:                 List[p+i].avancer()
11:             sinon
12:                 C = C : 0
13:         sinon
14:             C = C : ConstructK2Tree ( n/k, l + 1, p + i ∗ (n/k), q + j ∗ (n/k))
15: si C est un vecteur nul alors
16:     Retourner 0
17: A[l] = A[l] : C
18: Retourner 1
```

Une fois construites, les deux structures (T et L) permettent d'extraire les voisins directs et inverses d'un nœud directement sans décompression. Nous fournissons dans l'annexe A les détails de ces algorithmes d'extraction de voisinage.

Un autre type de graphe supporté par notre moteur est le type de graphes dynamiques. Ces graphes sont représentés par un ensemble de graphes statiques chacun capturé à un instant t_i. De ce fait, l'algorithme de construction peut être facilement adapté avec chaque nœud dans l'arbre contenant, cette fois-ci, un vecteur de bits chacun faisant référence à un instant t_i. Nous fournissons ci-après (algorithme 5) l'algorithme de construction de l'arbre k^2-tree à partir de la matrice d'adjacence tridimensionnelle.

Algorithme 5 : DynK2Tree

Entrée :

- n : la taille de la sous-matrice
- l : le niveau de l'arbre
- p : l'indice ligne de début de la sous-matrice
- q : l'indice colonne de début de la sous-matrice

Sortie : La valeur du nœud de la sous-matrice

```
1: C = ∅
2: Creturn = ∅
3: pour i = 1...k faire
4:    pour j = 1...k faire
5:       si l = log_k(N) alors
6:          pour m = 1...T faire
7:             C[m] = C[m] : M [p+i][q+j][m]
8:       sinon
9:          Ctmp =∅
10:         Ctmp = DynK2Tree ( n/k, l + 1, p + i ∗ (n/k), q + j ∗ (n/k))
11:         pour m = 1...taille(Ctmp) faire
12:            C[m] = C[m] : Ctmp[m]
13:            Creturn [m] = Creturn [m] ou Ctmp [m]
14: si C est un vecteur nul alors
15:    Retourner 0^|T| // retourner un vecteur nul de taille T.
16: pour i = 1...k ∗ k faire
17:    pour j = 1...T faire
18:       si C[i] n'est pas un vecteur nul alors
19:          A[l] = A[l] : C [i][j]
20: Retourner Creturn
```

3.3 P-GraCE :

Notre deuxième moteur P-GraCE est un moteur de compression par extraction de motifs. Il englobe quatre approches de compression différentes mais qui partagent toutes le même schéma global de compression. Nous présenterons dans cette partie, en détails, leur principe de fonctionnement et les différents phases qui les constituent.

3.3.1 Principe de fonctionnement :

Les méthodes de compression par extraction de motifs sont des méthodes qui essayent de représenter le graphe de données à travers ses motifs, autrement dit ses sous-graphes portant les

informations les plus importantes.

P-GraCE est un moteur de compression comportant plusieurs méthodes qui peuvent être avec ou sans perte de données. En effet, le modèle produit en sortie (résultat de la compression) est parfois accompagné avec une matrice d'erreur qui sera représentée dans ce cas avec une structure d'arbre k^2-tree. Le processus de compression d'un graphe de données par le moteur P-GraCE passe par les étapes suivantes :

1. Lecture et structuration du graphe de données en entrée
2. Extraction et évaluation des motifs : Durant cette phase, une détection des motifs les plus denses ou les plus fréquents est réalisée. Leur évaluation dans cette phase permet de ne garder que les structures (motifs) susceptibles de donner une meilleur compression.
3. Traitement des motifs : Ce module permet d'encoder ou d'agréger, dans certains cas, les motifs déjà découverts dans le but de minimiser davantage la taille du graphe en sortie. Dans d'autres cas, le module retourne une liste de structures sélectionnées parmi les motifs précédemment identifiés en utilisant des heuristiques dans le but de ne garder que les motifs importants du graphe.
4. Écriture du graphe compressé sur le fichier de sortie.

La figure 3.4 permet d'illustrer le principe de fonctionnement globale du moteur P-GraCE

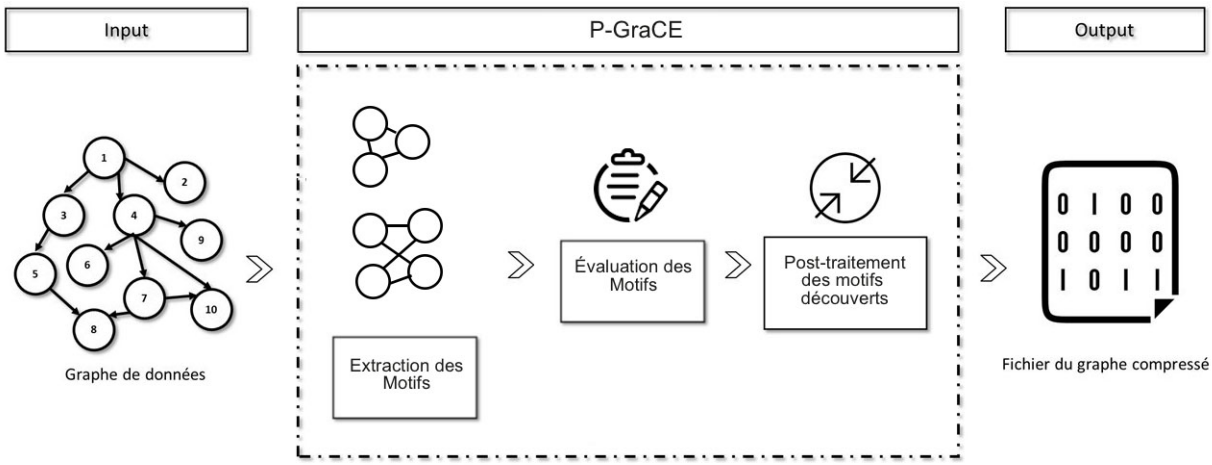


FIGURE 3.4 – Vue globale sur le fonctionnement du moteur P-GraCE.

Nous présenterons dans ce qui suit les trois phases de chaque approche incluse dans le moteur P-GraCE. Nous précèderons cela par expliquer les différents paramètres et notations que nous avons adoptés durant la conception de notre deuxième moteur.

3.3.2 Paramètre et notations :

Paramètre	Signification
G	Graphe de données
V	L'ensemble des nœuds de G
E	L'ensemble des arêtes de G
L	L'ensemble des étiquettes de G
A	Matrice d'adjacence du graphe G
M	Le modèle produit par la compression
h	Heuristique d'évaluation
N	Nombre de nœuds dans le graphe
k	Paramètre déterminant le nombre de fils à considérer dans le beam-search

TABLE 3.2 – Tableau des notations et paramètres du moteur k^2-GraCE.

3.3.3 Conception modulaire :

Durant cette section, nous allons présenter les différentes approches qui sont offertes par le moteur P-GracE. Nous commencerons par expliquer les méthodes d'extraction de motifs utilisées dans chacune. Nous enchainerons avec les techniques d'évaluation employées par ces méthodes, pour finir par expliquer comment ces motifs seront traités et utilisés pour compresser le graphe.

3.3.3.1 Extraction des motifs :

La phase d'extraction de motifs est une phase très importante dans le processus de compression. Elle permet de trouver les composantes les plus denses qui représentent en général l'information utile dans un graphe de données. Plusieurs techniques existent pour réaliser cette tâche. Elles diffèrent dans la qualité des sous-structures découvertes selon le domaine d'application et le type de graphe en entrée. Nous présenterons ci-dessous les quatre méthodes offertes par le moteur P-GraCE.

1. **Beam search :**
   Le beam search est une méthode de recherche locale gloutonne. C'est une version améliorée de l'algorithme de recherche en largeur BFS. Elle permet de n'explorer que les k meilleurs fils dans chaque niveau à travers des heuristiques d'évaluation.

   Dans le cas d'extraction de motifs dans un graphe, le beam search commence par considérer que chaque nœud du graphe est un motif. Il utilise pour cela un arbre de recherche où ses nœuds sont des sous-structures. À chaque itération les motifs sont étendus par une arête et un nœud donnant ainsi un ensemble de sous-structures. Par la suite, les k meilleurs sous-structures sont choisies. Les fils restants sont donc élagués. Cette alternative sera employée pour le cas des graphes étiquetés. Nous fournissons ci-après l'algorithme du beam

search que nous avons utilisé.

Algorithme 6 : Beam-Search

Entrée :

- G : le graphe de donnée
- k : le nombre de fils à considérer
- limit : limite de la profondeur de l'arbre

Sortie : retourne la meilleur sous-structure

1: C = {v | v est un nœud ayant une unique étiquette dans G}
2: *meilleurStruct* = la première sous-structure de C
3: **répéter**
4: nouvelC= ∅
5: **pour** chaque S dans C **faire**
6: nouvelStruct = Etendre(S)
7: Evaluer(nouvelStruct)
8: nouvelC = nouvelC ∪ { les k meilleur sous-structures de nouvelStruct }
9: limit = limit -1
10: **si** h(meilleur sous-structure de C) >= h(*meilleurStruct*) **alors**
11: *meilleurStruct* = meilleur sous-structure de C
12: C = nouvelC
13: **jusqu'à** $C = \emptyset$ ou $limit \leq 0$

Afin de mieux illustrer le fonctionnement de cet algorithme, nous proposons l'exemple du graphe de la figure 3.5. Nous considérons dans cet exemple qu'un sommet est représenté sur 8 bits, un lien est représenté sur 8 bits et que chaque pointeur est représenté sur 4 bits.

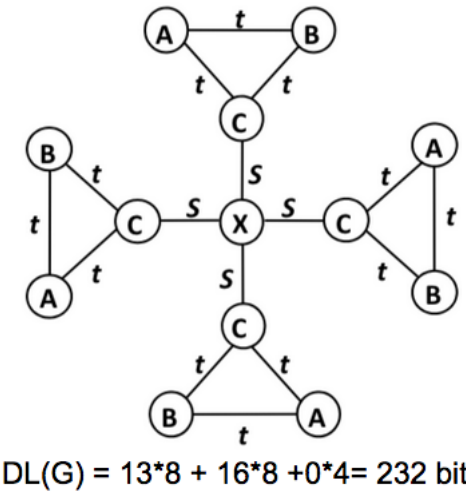


FIGURE 3.5 – Exemple de Graphe Étiqueté

Au départ chaque nœud est considéré comme une sous-structure, nous obtenons donc quartes sous structures : A, B, C et X. Par la suite, chacune des sous structures est étendues

par un sommet et une arête tout en estimant le gain obtenu par chaque extension. Nous obtenons donc encore quatre sous structures : A-B, A-C, B-C et X-C. Les trois premières sous-structures donnent le même gain de 24 bits. Tant dis que la quatrième donne une configuration non permise à cause des chevauchements entre ses instances qui possèdent le nœud x en commun. La dernière étape donne ainsi le motif A-B-C apportant un gain de 64 bits et qui représente la meilleur sous-structure.

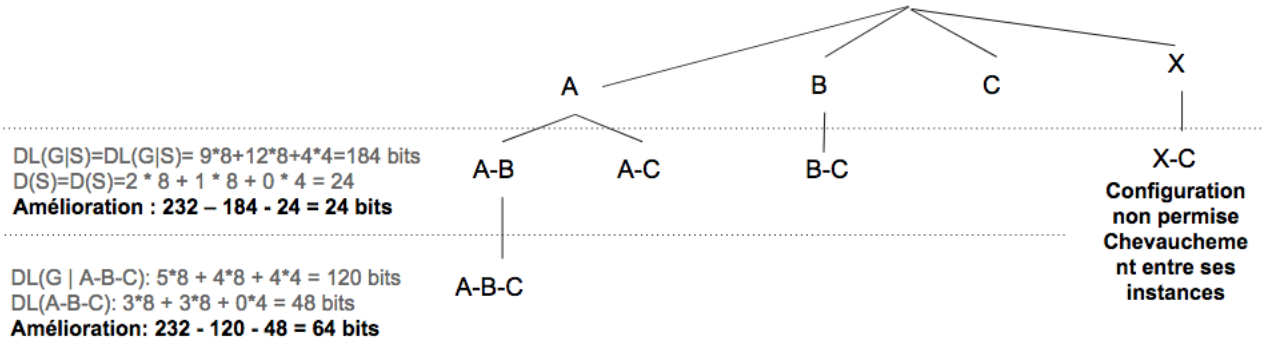


FIGURE 3.6 – Arbre de recherche (k= ∞ , limit = ∞)

2. **Méthodes de clustering :**

   Plusieurs méthodes de clustering existent dans la littérature. P-GraCE utilise un ensemble de méthodes de clustering destinées pour compresser un graphe de données en entrée. Nous les détaillons dans ce qui suit.

   SlashBurn : Cette technique part de l'observation que les graphes du monde réel sont facilement décomposables en supprimant leurs nœuds concentrateurs qui sont définies comme les nœuds ayant un degré maximale dans le graphe G.

   À chaque itération le nœud concentrateur est supprimé et le graphe est décomposé en un nœud concentrateur (hub), un Composant Connecté Géant (CCG) et des rayons restants que nous définissons comme étant le composant connecté non géant connecté aux anciens CCG. Le nœud concentrateur obtient ainsi le plus petit identifiant , les nœuds des rayons (spokes) reçoivent les identifiants les plus élevés dans l'ordre décroissant de la taille du composant connecté auquel ils appartiennent, et le nouveau CCG prend les identifiants restants. Le même processus s'applique aux nœuds du nouveau CCG, de manière récursive.

   Nous illustrons le principe de fonctionnement de cette méthode de clustering sur le graphe de la figure 3.7. Le nœud concentrateur (8) obtient le plus petit identifiant (1), les nœuds des rayons reçoivent les identifiants les plus élevés (9-16), et le CCG prend les identifiants restants (2-8). La prochaine itération considère le nouveau CCG. Nous fournissons l'algorithme détaillé en annexe 13.

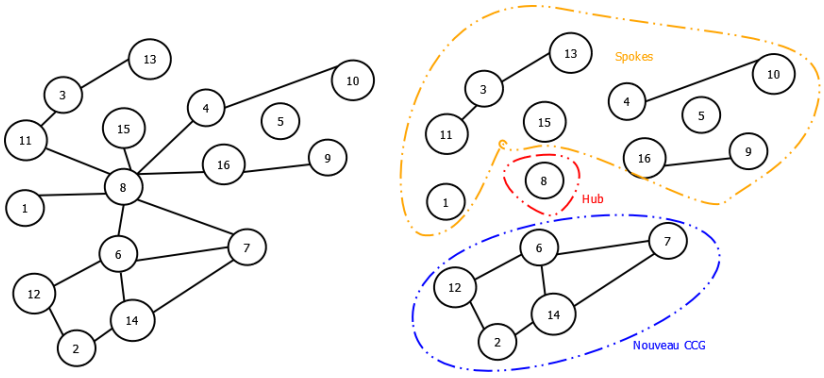


FIGURE 3.7 – Exemple d'application de l'algorithme SlashBurn

K-Cores : Cette technique se base sur le principe de dégénérescence [1]. Soit G un graphe non orienté, nous notons par Δ(G) le degré maximal des nœuds de G. Un k-core de G est un sous graphe H de taille maximale tel que $\Delta(H) \leq k$ avec k entier positif.
Pour décomposer le graphe, la méthode passe par quatre étapes. D'abord, le degré k de chaque nœud du graphe est calculé. Ensuite, pour chaque degré k allant du plus grand au plus petit le graphe est décomposé en supprimant tous les nœuds dont le degré est inférieure à k, l'opération s'arrête si la décomposition donne un ensemble non vide et le k est choisit comme k_{max} sinon elle se termine quand k_{max}=1. Après la décomposition, chaque composant est identifié comme étant une structure. Enfin, les arêtes entre les structures précédemment identifiées sont supprimées. L'algorithme de la méthode est donné en Annexe (14)

Spectral : Cette méthode utilise les résultats d'algèbre linéaire afin de trouver une partition du graphe. L'approche générale consiste à utiliser une méthode de classification standard (généralement k-means) sur les premiers vecteurs propres (vecteurs de Fiedler) de la matrice Laplacienne du graphe. La matrice Laplacienne est définie comme suit :

$$M_{Lap} := \begin{cases} \sum_{k=1}^{n} poids(i,k) & si \quad i = j \\ -poids(i,j) & sinon \end{cases}$$

Chaque vecteur de Fiedler permet d'obtenir une partition du graphe en deux parties. Pour partitionner le graphe, nous devons minimiser le coût de coupe de la bissection, cela revient à trouver un vecteur $x \in \{-1, 1\}^n$ qui minimise $x^T M_{Lap} x$. Pour cela, nous résolvons le système linéaire $M_{Lap} x = \lambda x$.
L'algorithme prend comme entrée la matrice d'adjacence que nous notons A, une autre matrice diagonal notée D est crée tel que : D :

$$d_i := \sum_{(v_i, v_j) \in E_m} e(v_i, v_j)$$

1. : La dégénérescence d'un graphe G est le plus petit nombre k de telle sorte que chaque sous-graphe $S \in G$ contient un sommet de degré au plus k

L'algorithme calcule ensuite le deuxième plus petit vecteur propre *y* de la matrice Laplacienne Q = D - A. Ce vecteur propre (vecteur de Fiedler) contient une valeur pour chaque nœud du graphe, à partir de cette valeur le nœud est affecté à l'une des deux partition. Soit r le médian pondéré des valeurs de *y*. Chaque valeur y_j du vecteur propre est comparée à r et le nœud est affecté à une partition selon le résultat de la comparaison.

3. **Le MinHash :**

Dans cette alternative, nous allons partitionner le graphe en des motifs denses dont les nœuds sont fortement connectés. Nous utiliserons pour cela une approximation de *la similarité de Jaccard* qui est une métrique permettant de mesurer la similarité entre deux ensembles S_1 et S_2 (voir formule 3.1).

$$SIM(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{3.1}$$

L'objectif de cette méthode est de remplacer les listes d'adjacence des nœuds par des représentations beaucoup plus petites appelées « signatures ». La propriété importante de ces signatures est que leur comparaison permet d'estimer la similarité de Jaccard des listes d'adjacence sans avoir recours à les comparer deux à deux.

Avant d'expliquer comment il est possible de construire de petites signatures à partir des listes d'adjacence, il est utile de les visualiser en tant que matrice caractéristique. Les colonnes de cette matrice correspondent aux listes d'adjacence et les lignes correspondent aux nœuds. Nous rappelons que la matrice caractéristique est peu susceptible d'être la façon dont les données sont stockées, mais elle est utile pour visualiser les données (voir figure 3.8).

Dans un premier temps, k-permutations aléatoires des nœuds devront être choisies. L'utilisation des k-permutations se base sur le fait que :

$$P(\pi_i(S_1) = \pi_j(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = SIM(S_1, S_2) \tag{3.2}$$

Nous construisons la matrice de signature en considérant chaque ligne dans leur ordre donné. Soit $SIG(i, c)$ l'élément de la matrice de signature pour la i^{me} fonction de hachage et la colonne c. SIG (i, c) est initialisée à ∞ pour tout i et c. Nous traitons une ligne r en procédant comme suit :

(a) nous calculons $h_1(r), h_2(r), ..., h_k(r)$.

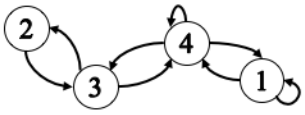
(b) Pour chaque colonne, nous procédons comme suit :

— Si c = 0 dans la ligne r, rien à faire.

— Cependant, si c = 1 dans la ligne r, alors pour chaque i = 1,2, ..., k
$SIG(i, c) = \min(\ SIG(i, c)\ ,\ h_i(r)\ )$.

Une fois les signatures sont construites, les nœuds ayant les mêmes valeurs seront regroupés. Nous obtiendrons ainsi en sortie la liste des motifs contenant les nœuds ayant

un voisinage similaire dans le graphe de données. Nous illustrons le principe de fonctionnement de cette stratégie sur un graphe G dans la figure 3.8. Le tableau gauche de la figure 3.8 montre une matrice caractéristique des listes d'adjacence des différents nœuds du graphe G ainsi que deux permutations aléatoires h_1, h_2. Tant dis que le tableau à droite de la même figure montre les étapes de calcul des signatures en utilisant ces deux fonctions. La matrice finale des signatures montre bien que les nœuds 1 et 4 sont les plus similaires ce qui est justifié par leurs listes d'adjacence qui ne diffèrent que dans un seul élément.



x	Liste d'adjacence
1	S1={1, 4}
2	S2={3}
3	S3={2, 4}
4	S4={1, 3, 4}

	S1	S2	S3	S4	h1	h2
1	1	0	0	1	**2**	**2**
2	0	0	1	0	**3**	**1**
3	0	1	0	1	**4**	**3**
4	1	0	1	1	**1**	**2**

Matrice Caractéristique

		S1	S2	S3	S4
initialisation	**h1**	∞	∞	∞	∞
	h2	∞	∞	∞	∞
Itération 01	**h1**	2	∞	∞	2
	h2	2	∞	∞	2
Itération 02	**h1**	2	∞	3	2
	h2	2	∞	1	2
Itération 03	**h1**	2	4	3	2
	h2	2	3	1	2
Itération 04	**h1**	1	4	1	1
	h2	2	3	1	2

Étape de calcul de la Matrice des Signatures

FIGURE 3.8 – Exemple de calcul de la matrice des signatures.

Après l'étape de clustering, une arborescence est ensuite construite pour chaque ensemble de sommets ayant les même signatures et qui permettra d'extraire les sous-structures (motifs) denses du graphe à travers un parcours de la racine vers les feuilles.

4. **Extraction de Motifs de la matrice d'adjacence :**

   La dernière méthode d'extraction de motifs offerte par le moteur P-GraCE se base sur les blocs les plus fréquents dans la matrice d'adjacence. Cette méthode est destinée aux graphes ayant une matrice d'adjacence contenant de larges zones nulles. L'ensemble des motifs à découvrir est prédéfinies au départ et consiste en trois types de motifs qui sont paramétrés par la taille du motif :

   (a) **Type 01 :** Ce premier ensemble de motifs consiste en les vecteurs ayant uniquement deux '1', le premier dans le bit de poids fort et un autre dans l'une des position restante, et le vecteur nul. Pour une taille de motif de 2^n bits, nous obtenons un ensemble de taille 2^n.

   (b) **Type 02 :** Le deuxième ensemble de motifs consiste en les vecteurs ayant uniquement un seul '1'. La taille de cet ensemble est de 2^n motifs pour des motifs de taille 2^n bits.

(c) **Type 03 :** Ce dernier ensemble de motifs consiste en l'union des deux ensembles précédents. Dans le cas de cet ensemble, nous obtenons 2^{n+1} motifs pour une taille de motifs de 2^n.

Chaque ligne de la matrice d'adjacence est donc divisée en plusieurs blocs ayant la même taille que les motifs. Ces blocs seront comparés avec les motifs selon le type choisi et dans le cas de correspondance ils seront remplacés par l'identifiant du motif (0...n-1) qui est de taille n pour des motifs de taille 2^n.

3.3.3.2 Évaluation des motifs :

Dans cette deuxième phase, P-GraCE cherche à obtenir la meilleur compression en filtrant les sous-structures et en ne gardant que celles qui offrent un bon MDL.

1. Comme nous l'avons déjà précisé P-GraCE utilise le beam-search pour le cas des graphes étiquetés. Il est fondamentalement guidé par le principe MDL. L'évaluation heuristique effectuée suppose que la meilleure sous-structure est celle qui minimise le MDL du graphe en entrée lorsqu'il est compressé par cette sous-structure(S). Notons la longueur de description de S par *DL(S)*, la longueur de description du graphe d'entrée G par *DL(G)* et la longueur de description du graphe après compression par *DL(G|S)*. L'heuristique est alors rien d'autre que le ratio de compression et peut être définie ainsi :

$$\text{compression} = \frac{DL(G)}{DL(S)+DL(G|S)}$$

2. Dans le cas de l'utilisation des méthodes de clustering, nous obtenons une liste des structures les plus fréquentes. Pour chaque structure identifiée nous cherchons le motif qui la décrit le mieux en se basant sur un ensemble prédéfini de motifs (étoile, clique, noyau bipartie, semi clique, semi noyau bipartie). Afin de trouver le motif approximatif de la structure, nous utiliserons les formules proposées dans (Koutra et al., 2015) (voir section 2.4.1).
3. Dans le cas de l'utilisation de la technique du MinHashing, nous utilisons le gain apporté par la compression des motifs comme métrique d'évaluation. Nous utilisons pour cela la même formule proposée dans (Buehrer and Chellapilla, 2008). En effet, la qualité de compression d'un motif est calculée en fonction de sa fréquence dans la liste d'adjacence, et de sa taille qui est le nombre de liens qu'il contient (Formule 3.3).

$$Compression(P) = (P.frequence - 1)(P.taille - 1) - 1 \tag{3.3}$$

4. Dans le cas de l'extraction de motifs à partir de la matrice d'adjacence aucune évaluation n'est nécessaire car tout les motifs offrent le même gain d'espace mémoire.

3.3.3.3 Traitement des motifs :

Nous proposons pour traiter les motifs dans le moteur P-GraCE plusieurs alternatives, cela dépend de la méthode utilisé.

La première consiste à faire une agrégation des nœuds du meilleur motif. Les trois étapes seront donc répétées jusqu'à ce que la taille du graphe en sortie ne peut plus être optimisée. Elle est appliquée dans le cas de l'approche beam-search.

La deuxième alternative consiste à considérer que la liste des sous-structures de l'étape 2 représentent le compressé du graphe de données. Nous effectuons une sélection à partir de l'ensemble des structures initiales pour obtenir un modèle (ensemble de structures) avec un MDL optimal. Cette option sera employée dans l'approche basée sur les méthodes de clustering . La sélection se fait à travers plusieurs algorithmes. Nous présentons ci dessous les méthodes que nous allons utiliser durant cette phase :

— Top10 : Cette méthode commence par ordonner les sous structures selon le gain de codage local apporté par chacune d'elles. Le gain de codage local est définit par L(g,∅) - L(g,ω) où L(g,∅) représente le codage de g (g est le sous graphe contenant la structure) en tant qu'erreur (un modèle vide) et L(g,ω) est le cout de codage de g avec la structure ω. Elle sélectionne ensuite les k meilleurs structures ayant un gain local élevé.

— Greedy'nForget (GNF) : Au lieu de prendre en compte toutes les combinaisons possibles des structures, l'heuristique GNF considère les structures selon un ordre décroissant de leur gain d'encodage local. Elle parcourt structure par structure et inclut dans le modèle celle qui réduit le MDL. L'algorithme de GNF est présenté dans ce qui suit (7)

Algorithme 7 : Geedy'nForget

Entrée :

- C : Listes des structures évaluées.

Sortie : ensemble des structures sélectionnées $\mathcal{M}$

```
1: MDL-optimal = MDL(G,∅)
2: Ordonner(C,"descendant")
3: pour s ∈ C faire
4:     M = M ∪ s
5:     si MDL(G,M) < MDL-optimal alors
6:         MDL-optimal = MDL(G,M)
7:     sinon
8:         M = M \ s                ► Enlever la structure de l'ensemble
```

— STEP : Cette méthode parcourt de manière itérative toutes les structures évaluées précédemment dans un ordre quelconque et choisit la structure qui réduit le plus le MDL du modèle courant pour l'ajouter au modèle jusqu'à ce qu'aucune structure n'améliore le cout. Nous présentons l'algorithme détaillé ci-dessous :

Algorithme 8 : STEP

Entrée :

- C : Listes des structure évaluées.

Sortie : ensemble de structures sélectionnées $\mathcal{M}$

```
1: Best-Structure = -1
2: MDL-optimal = MDL(G,∅)
3: tantque C != ∅ faire
4:    pour  s ∈ C  faire
5:       M = M ∪ s
6:       si  MDL(G,M) < MDL-optimal alors
7:          MDL-optimal = MDL(G,M)
8:          Best-structure = s
9:       M = M \ s //Enlever la structure de l'ensemble
10:   si Best-Structure != -1 alors
11:      M = M ∪ Best − structure
12:      Best-Structure = -1
13:   sinon
14:      STOP
15:   C = C \ Best − Structure
```

— Pour la méthode du minhashing, nous offrons la possibilité d'utiliser le codage proposé dans (Liu et al., 2018b). Il représente cette liste de motifs de manière concise tout en permettant les requêtes d'extraction de voisinage.

3.4 Notre méthode : Dynamic Dense Subgraph Mining (DDSM)

Comme on vient de voir dans les chapitres précédents, de nombreux phénomènes dans des contextes très variés peuvent être représentés par des graphes. Citons en particuliers les réseaux sociaux qui nous attirent vu leur importance et leur fort impact sur la vie réelle. En effet, trois quarts des personnes connectées à Internet utilisent les réseaux sociaux. Ils rassemblent aujourd'hui 3,169 milliards d'utilisateurs actifs, voir 40% de la population sur terre, avec 11 personnes s'inscrivant chaque seconde sur Facebook, Twitter et autres [2]. De ce fait, deux constats s'imposent : le premier est l'incapacité des graphes statiques à décrire certaines situations de la vie réelle et leur évolution dans le temps, le deuxième est la montée en flèche de la quantité et l'hétérogénéité des informations modélisées par ces réseaux et qui rendent presque impossible tout traitement. D'où le besoin croissant de méthodes de compression de graphes dynamiques.

Nous nous sommes aussi intéressées dans ce travail à un autre enjeu important qui est la recherche de motifs et particulièrement les cliques et les noyaux bipartis qui sont fréquents dans les réseaux faisant abstraction à des situations réelles importantes dans le processus de prise de

2. https ://bfmbusiness.bfmtv.com/hightech/la-moitie-de-la-population-mondiale-est-connectee-a-internet-1361933.html

décision. Nous citons par exemple, les attaquants de réseaux de botnet formant un noyau biparti avec leurs victimes pendant la durée d'une attaque, les membres de la famille se liant comme une clique au cours d'une période difficile, ou les collaborations de recherche s'étant créées et qui disparaissent au cours des années.

Nous allons dans cette partie proposer une nouvelle méthode DDSM basée sur deux approches existantes. Notre but principal est de développer une technique de compression compétitive pour les graphes des réseaux d'interactions dans un contexte dynamique en se basant sur l'extraction de motifs pour filtrer les informations les plus importantes et les arbres k^2-trees pour aboutir à une compression sans perte. Pour accomplir ce travail nous nous sommes appuyées sur les deux techniques suivantes :

— Exploiter la structure de données proposée dans DSM (Hernández and Navarro, 2014) qui permet de représenter les sous-graphes denses comme les cliques, les quasi-cliques, noyaux bipartis de manière concise. Nous avons opté pour cette structure car elle donne de bons résultats en matière d'espace et de temps de requêtes.

— Adopter les signatures temporelles utilisées dans TimeCrunch (Shah et al., 2015) pour décrire le comportement des sous-graphes dans le temps. Nous avons utilisé ces signatures car elles englobent tous les types de comportements temporels d'un motif dans un graphe dynamique.

3.4.1 Formulation du problème

Dans cette section, nous allons définir le problème de base auquel notre méthode convient tout en définissant le cadre dans lequel elle peut être appliquée.

Nous considérons les graphes orientés dynamiques $G = \bigcup_{t_i} G_{t_i}(V, E_{t_i})$ $1 \preceq i \preceq t$ où $G_{t_i} = \mathrm{G}$ à l'instant t_i et un ensemble de signatures temporelles. En d'autre termes, nous considérons des captures du graphe à des instants différents t_i et un ensemble de descripteurs de comportement temporel des sous-graphes des différents G_{t_i}. Le problème peut ainsi être formulé :

Problème : *Trouver, à partir d'un graphe dynamique G et un lexique de signatures temporelles, la plus petite description du graphe initial en termes de ses sous structures les plus denses et leur comportement temporel, en offrant la possibilité de manipuler le graphe, d'extraire les voisins directs d'un nœud à un instant donné et d'extraire les sous-graphes au besoin.*

3.4.2 Principe général

Notre méthode s'intitule DDSM pour *Dynamic Dense Substructure Mining*. Elle représente une généralisation du travail de Hernandez et Navarro (Hernández and Navarro, 2014) pour le cas des graphes dynamiques qui sont de nos jours omniprésents.

La codification du graphe en sortie doit respecter les contraintes du problème tout en réduisant un maximum d'espace mémoire. Pour cela, nous proposons d'étendre la codification proposée dans (Hernández and Navarro, 2014) pour le cas des graphes dynamiques en rajoutant une information temporelle. Une fois les sous-structures identifiées, Hernandez et al. (Hernández and Navarro, 2014) proposent de représenter chacune d'elles avec trois composantes : la

première contenant les sommets ayant uniquement des arêtes sortantes, la deuxième contenant les sommets ayant des arêtes entrantes et sortantes et la troisième contenant les sommets ayant uniquement des arêtes entrantes. Pour pouvoir identifier les différentes composantes, ils associent un vecteur binaire à cette représentation marquant par un 1 le début de chacune des trois composantes (voir figure 2.15). Notre première contribution consiste en l'extension de cette structure. En effet, nous suggérons de représenter chaque sous structure avec non pas trois composantes mais quarte composantes. Les trois premières étant les même que dans (Hernández and Navarro, 2014), la quatrième représente l'information temporelle. Cette dernière peut avoir cinq formats (voir section 2.4.1) dont nous résumons la représentation proposée pour chacune dans le tableau 3.3.

Signature temporelle	Représentation
constante	0
OneShot	1 t_1
ranged	2 t_1 t_2
periodic	3 T
flikering	4 t_1 t_2 ... t_n

TABLE 3.3 – Les types de signatures temporelles et leurs représentations, t_i représentent les timestamps et T représente la période.

Nous représentons G donc comme un ensemble de sous-graphes temporels denses. Cependant, pour obtenir une compression sans perte, nous devons aussi garder l'erreur modélisant l'ensemble d'arêtes restantes dans chaque G_{t_i}. Nous proposons pour cela d'utiliser une des structures dynamiques des k2-trees qui nous permettra de garder trace de cette dernière dans un format compacte. Nous ferons appel à cette étape au moteur k^2-GraCE.

Nous visons à travers la méthode que nous proposons d'exprimer le graphe en entrée avec ses sous-structures les plus denses et leur comportement temporel réduisant ainsi sa taille et offrant la possibilité d'effectuer les traitements dans un temps meilleur. Nous avons structuré notre algorithme sous forme de trois (03) étapes essentielles, chacune servant d'entrée pour l'étape suivante.

En premier lieu, nous appliquons la découverte des sous-graphes les plus denses. Pour effectuer cela de manière efficace, nous suggérons d'utiliser la technique du MinHashing du moteur P-GraCE parallèlement sur chaque capture G_{t_i} de G.

Dans une deuxième phase, nous effectuons une comparaison entre les sous-structures. Nous utilisons pour cela le principe du MinHashing encore une fois pour grouper les sous-structures de différents timestamps ayant un nombre élevé de nœuds en commun. Ayant associé à chaque sous-structure ses différents timestamps, ces derniers seront à leur tour compressés en utilisant les descripteurs de comportement temporel précédemment décrits. L'algorithme 9 représente les étapes de compression de l'ensemble des timestamps d'une structure.

Algorithme 9 : getTemporalSignature

Entrée :

- timeseries[] : un tableau de timestamps
- size : la taille du tableau timeseries

Sortie : la signature temporelle

```
1:  si |timeseries| = T alors
2:      return 'c'                                              ► Constante
3:  sinon
4:      si |timeseries| = 1 alors
5:          return 'o'                                          ► OneShot
6:      sinon
7:          si t_{i+1} − t_i =1 ∀t_i ∈ timeseries alors
8:              return 'r' : timeseries[0] :timeseries[size-1]  ► Ranged
9:          sinon
10:             si t_{i+1} − t_i = Constante ∀t_i ∈ timeseries alors
11:                 retourner 'p' : timeseries[1] - timeseries [0]   ► periodic
12:             sinon
13:                 return 'f' :timeseries[0] :... :timeseries[size-1]   ► flickering
```

Une dernière phase consiste en la codification du graphe en utilisant le codage proposé pour chaque sous-structure de la phase (02). Nous concaténons par la suite ces représentations pour obtenir une seule représentation concise du graphe initial en terme de ses structures les plus denses. Nous schématisons ces différentes étapes dans la figure 3.9.

L'algorithme 10 représente l'algorithme de compression globale.

Algorithme 10 : DDSM

1: **Génération des sous structures candidates :** Génération de sous-graphes, principalement les bicliques et les cliques.
2: **Étiquetage de sous-structures :** Associer chaque sous-structure à une signature temporelle décrivant son comportement.
3: **Codification du graphe compressé :** Codifier les sous-structures étiquetées de l'étape (02) en utilisant les structures de la section précédente.

Pour les algorithmes de parcours et d'extraction de voisins, tous les algorithmes proposés dans (Hernández and Navarro, 2014) peuvent être généralisés sur notre structure. En effet, le seul changement consiste en la prise en compte de l'information temporelle dans l'incrémentation des indices de parcours. De ce fait, nous pensons que notre méthode peut offrir un très bon compromis entre l'espace mémoire et le temps d'accès des traitements dans le cas des graphes dynamiques.

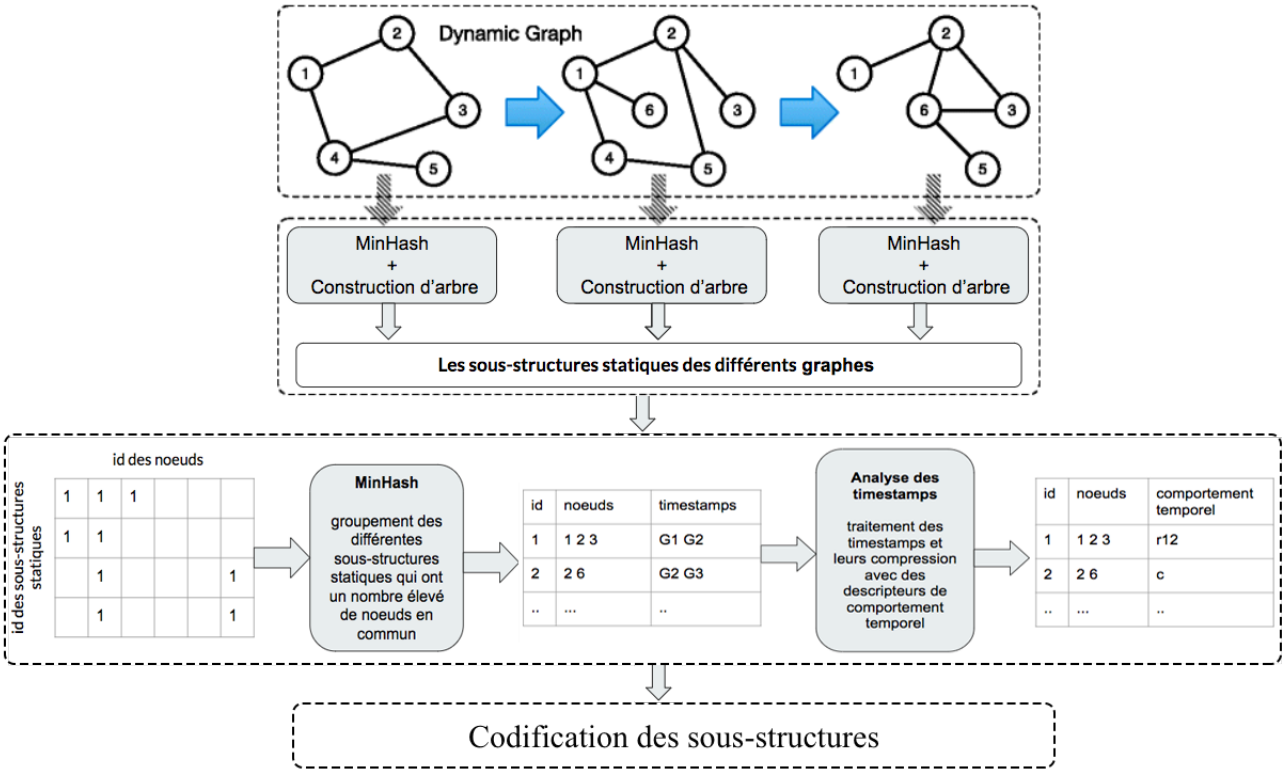


FIGURE 3.9 – Les différentes phases de DDSM

3.5 Conclusion

Dans ce chapitre, nous avons expliqué la partie conceptuelle de nos deux moteurs : k^2-GraCE et P-GraCE, qui représentent les deux classes qui font l'objet de notre recherche. Nous avons présenté leur différents modules et fonctionnalités tout en clarifiant leurs paramètres ainsi que leur principe de fonctionnement. Nous avons aussi proposé une nouvelle méthode de compression destinée pour les graphes dynamiques utilisant principalement la technique du Minhashing.

Dans le chapitres suivant, nous présenterons les choix d'implémentation que nous avons effectués. Nous décrirons l'environnement de développement ainsi que l'architecture globale de notre projet.

Chapitre 4

Implémentation

4.1 Introduction

Notre projet s'intègre dans un projet de recherche scientifique. De ce fait, une implémentation efficace et optimisée de notre conception est nécessaire. Nous commencerons ce chapitre par présenter l'architecture existante et comment nous l'avons étendue avec nos deux moteurs tout en détaillant les différentes couches de cette architecture. Nous conclurons par présenter l'environnement de développement.

4.2 Architecture globale

L'architecture existante est une architecture en pipeline de 3-tiers. Nous adopterons cette même architecture. Elle se compose de trois couches logicielles que nous avons enrichies avec nos deux moteurs. Nous les présenterons ci-dessous :

1. **La couche données ou persistance :** cette couche est responsable de la gestion des données en entrée et les fonctions d'accès et de stockage. Les données en entrée ainsi que les résultats de compression sont sous forme de fichiers.
2. **La couche traitement :** c'est le noyau des moteurs de compression de notre projet. Elle inclue tout les algorithmes de compression et de manipulation des graphes.
3. **La couche présentation :** Deux types de résultats (fichiers) sont produits : le premier est le graphe compressé, le deuxième représente le fichier log des performances.

FIGURE 4.1 – Architecture du globale

4.3 Données

Les données manipulées par les moteurs de compression sont tous sous format de fichiers texte. La structure de ces fichiers diffère selon le type du graphe en entrée. Comme nous l'avons déjà implicitement mentionné, l'implémentation proposée prend en charge les types de graphes suivants : graphe dynamique orienté, graphe statique (orienté et non orienté) et finalement graphe étiqueté. Nous présenterons dans ce qui suit leurs structures :

- **Graphe statique :**

```
# Directed graph (each unordered pair of nodes is saved once): web-Stanford.txt
# Stanford web graph from 2002
# Nodes: 281903 Edges: 2312497
# FromNodeId    ToNodeId
1       6548
1       15409
6548    57031
15409   13102
2       17794
2       25202
2       53625
2       54582
2       64930
2       73764
2       84477
2       98628
```

FIGURE 4.2 – Structure d'un fichier de graphe statique

Le fichier représentant un graphe statique est composé de plusieurs lignes chacune faisant référence à un des liens du graphe de données. Elles incluent l'identifiant de la source suivi de l'identifiant de la destination. Les lignes dont le premier caractère est un "#" sont ignorées.

- **Graphe dynamique :**

```
# Dynamic Graph
# Nodes : 1899  edges : 59835    Time_span : 193 days
# SrcID DstID   UNIXTS
#
1 2 1082040961
3 4 1082155839
5 2 1082414391
6 7 1082439619
8 7 1082439756
9 10 1082440403
9 11 1082440453
12 13 1082441188
9 14 1082441754
9 15 1082441824
9 16 1082441895
9 17 1082442153
9 14 1082442328
9 18 1082442560
```

FIGURE 4.3 – Structure d'un fichier de graphe dynamique

Les liens dans un graphe dynamique sont représentés par trois composantes (u,v,t) signifiant qu'un lien entre les nœuds u et v appartient à la capture de l'instant t. Chaque lien représente une ligne dans le fichier de données.

- **Graphe étiqueté :**

```
v 1 object
v 2 object
v 3 object
v 4 object
v 5 object
v 6 object
v 7 object
e 1 11 shape
e 2 12 shape
e 3 13 shape
e 4 14 shape
e 5 15 shape
e 6 16 shape
```

FIGURE 4.4 – Structure d'un fichier de graphe étiqueté

Le fichier de données d'un graphe étiqueté est composé de deux types de lignes : les lignes commençant avec un "*v*" représentent les identifiants des sommets suivis de leurs étiquettes et les lignes commençant avec un "*e*" représentent les sommets source et destination des liens suivis de leurs étiquettes.

4.4 Traitement

Dans cette section, nous détaillerons les principales structures utilisées pour représenter les graphes en mémoire ainsi que les fonctions et méthodes décrivant les interfaces de nos deux moteurs de compression.

4.4.1 Les structures utilisées

Nous nous intéressons dans notre projet à trouver un bon compromis entre les performances de la compression (taux de compression et le nombre de bits par nœud) et le temps d'exécution. Pour cela nous avons essayé de choisir des implémentations de structures de données qui répondent à cette contrainte.

(a) **TNGraph :** Cette structure est l'une des structures les plus performantes offertes par la bibliothèque Stanford Network Analysis Package (SNAP) (voir section 4.6.2). Elle représente un graphe orienté et est implémenté à l'aide de fonctions de hachage offrant ainsi un temps d'accès très rapide. Elle offre aussi une panoplie d'opérations facilitant la manipulation du graphe de données.

(b) **TUNGraph :** Cette structure est aussi une des structures de la bibliothèque SNAP. Elle représente un graphe non orienté et offre les mêmes avantages que la structure précédente.

(c) **std : :vector<Boost : :dynamicbitset<> > :** Nous avons adopté cette structure pour représenter la matrice d'adjacence en mémoire. Les lignes de la matrice d'adjacence sont représentées par un vecteur de bits de la bibliothèque Boost donnant un accès rapide

comparé aux implémentations disponibles (Pieterse et al., 2010). Les tableaux de lignes seront stockés dans un tableau de la bibliothèque standard du langage c++ permettant d'exploiter la localité du cache.

(d) **std : :vector<std : :pair<std : :vector<unsigned int>,unsigned int> > :** Cette structure représente la liste d'adjacence d'un graphe. Comme nous l'avons déjà expliqué nous dotons chacune des listes par un pointeur indiquant la position du dernier élément visité. Les paires (listes, pointeurs) seront stockées dans un tableau indexé par les identifiants des nœuds du graphe (0...N).

(e) **LabeledGraph :** Pour les graphes étiquetés, nous proposons d'utiliser notre propre structure intitulée « *LabeledGraph* ».

(f) **std : :map<unsigned int,TNGraph> :** La dernière structure modélise les graphes dynamiques orientés qui sont représentés par les paires (t_i , G_i) où G_i représente une capture du graphe de données à l'instant t_i.

4.4.2 Les méthodes et fonctions

Nous présenterons ci-dessous les principales fonctions et méthodes décrivant le schéma global de fonctionnement de nos deux moteurs.

(a) **LoadGraph :** Cette fonction permet de charger le graphe de données en mémoire à partir d'un fichier texte. Elle prend en paramètres le fichier de données ainsi que le type de graphe dans le cas du moteur P-GraCE. Dans le cas du moteur k^2-GraCE, elle prend en paramètres le fichier de données, le type de graphe, le type représentation à utiliser pour compresser le graphe ainsi que le choix d'activer ou non le module de pré-traitement.

(b) **CompressK2 :** Cette fonction consiste en le processus de compression du graphe en utilisant les arbres k^2-trees. Elle reçoit en entrée le graphe de données et son type et donne en sortie les deux listes T et L représentant l'arbre k^2-trees.

(c) **CompressPattern :** Cette fonction permet de compresser le graphe de données en utilisant ses sous-structures les plus importantes. Elle prend en paramètres le graphe de données ainsi que le choix de l'approche de compression à utiliser et ses paramètres.

(d) **SaveCompressed :** Cette fonction effectue la sauvegarde du résultat de compression dans un fichier texte.

Le schéma de la figure 4.5 montre le schéma globale de fonctionnement des deux moteurs :

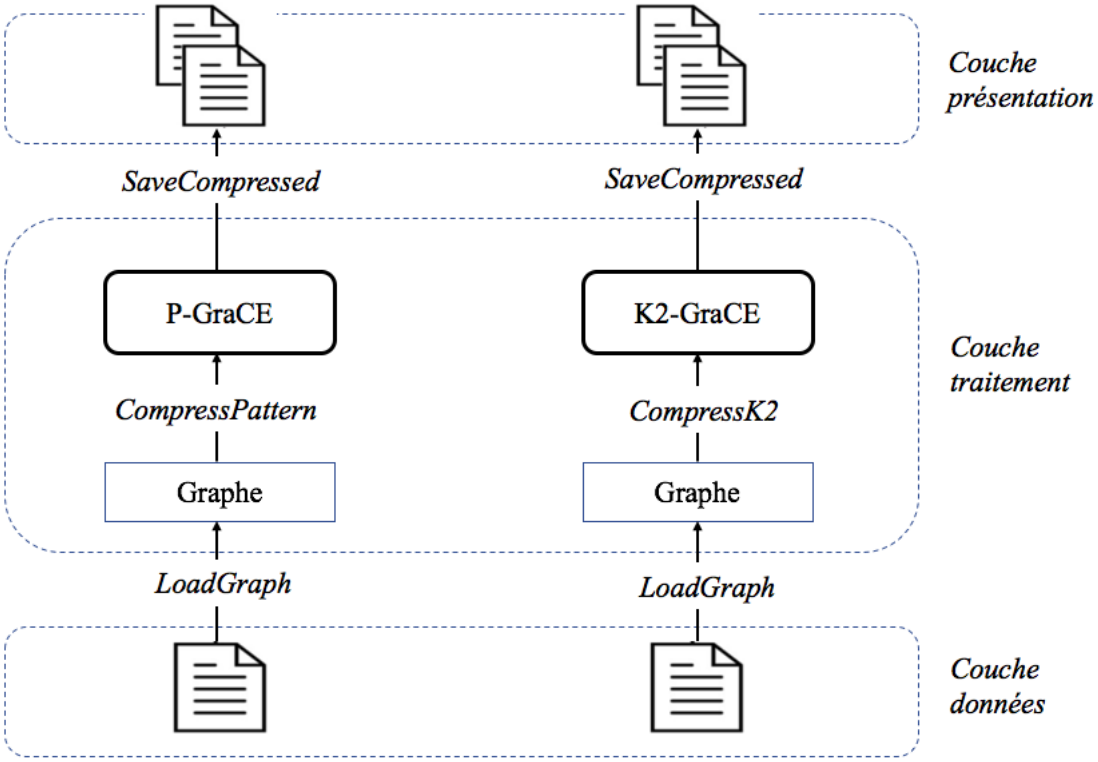


FIGURE 4.5 – Schéma globale du fonctionnement des deux moteurs

4.5 Présentation

La couche présentation est responsable de donner accès à toutes les fonctionnalités offertes par notre solution, entre autre le choix des paramètres pour différentes méthodes, ainsi que de visualiser les résultats. En plus du fichier de sortie, elle fournit les différentes mesures de performances. Elle offre ainsi aux chercheurs la possibilité de comparer les performances de différentes méthodes sur un même graphe.

4.6 Environnement de développement

Le choix des outils de développement dans tout projet informatique est très important vu leur fort impact sur les performances du produit final. Comme notre solution fait partie d'un projet de recherche, il faut aussi prendre en compte la flexibilité et la capacité de la solution à s'interfacer avec d'autre outils dans des systèmes qui peuvent être hétérogènes.

4.6.1 Langage de Programmation

Nous avons adopté le langage de programmation avec lequel la première version de notre solution a été développée : le C++. En effet, il représente un langage très performant pour les calculs lourds. Il permet d'avoir des exécutions très rapides, ce qui en fait un langage de choix pour les applications critiques qui ont besoin de performances. Il permet aussi d'avoir un code

portable : un même code source peut être facilement transformé en exécutable sous Windows, Mac OS ou Linux. Un autre aspect du langage c++ est sa richesse de bibliothèques optimisées pour le traitement et le stockage des grands graphes en mémoire.

Nous avons choisi Visual Studio 2015 comme environnement de développement (IDE). Notre choix a été influencé par le fait que Visual Studio s'ouvre à toutes les tendances du moment et permet facilement de travailler en équipe sur le même projet.

4.6.2 Bibliothèque Snap

Afin de faciliter la manipulation des graphes et d'améliorer les performances de notre solution, nous avons offert la possibilité d'exécuter les différents algorithmes de compression en utilisant une des plus performantes bibliothèques de manipulation de graphes en c++ : SNAP. SNAP est une bibliothèque d'analyse et d'exploration de graphes à usage général qui s'adapte facilement à des graphes massifs. Elle présente aussi l'avantage d'être efficace et facilement extensible.

4.6.3 Bibliothèque Boost

Boost est un ensemble de bibliothèques pour le langage de programmation C ++ qui prend en charge des tâches et des structures telles que l'algèbre linéaire, la génération de nombres pseudo-aléatoires, le multithreading, les expressions régulières et les tests unitaires. Elle contient plus de quatre vingt bibliothèques individuelles.

Nous l'avons utilisée dans notre projet pour l'implémentation des arbres k^2-trees. En effet, elle contient une implémentation des tableaux de bits en c++ qui donne un temps d'exécution optimal comparé aux autres implémentations existantes (Pieterse et al., 2010).

4.7 Conclusion

Durant ce chapitre, nous avons présenté notre implémentation où nous avons essayé d'utiliser différentes bibliothèques permettant d'avoir les meilleurs performances. L'indépendance des différentes couches de l'architecture existante nous a facilité la tache d'implémentation. Nous avons ainsi essayé de respecter cette indépendance afin de faciliter tout autre contribution future dans le projet.

Nous évaluerons dans le chapitre suivant les différentes méthodes de nos deux moteurs en vue de l'obtention d'une étude comparative plus objective et plus claire de leur performance.

Chapitre 5

Test

5.1 Introduction

Dans ce chapitre, nous exposerons les différents résultats des mesures de performances obtenus par nos deux moteurs k^2-GraCe et P-GraCe en utilisant plusieurs datasets dans différentes configurations. Nous commencerons par présenter l'environnement de test et les conditions d'expérimentation ainsi que les graphes utilisés. Nous allons par la suite présenter les résultats obtenus pour chaque méthode séparément et comparer par la suite entre les deux moteurs k2-GraCE et P-GraCE.

5.2 Environnement de Test

L'environnement de test compte parmi les éléments sur lesquels repose l'optimisation de tout système. C'est un environnement permettant de tester les différentes configurations et modules. Nous présenterons dans cette partie les caractéristiques matérielles et logicielles de l'environnement de test utilisé durant nos expériences.

- **Nombre de processeurs :** 1
- **Processeur :** Intel Core i7 (2,2 GHz)
- **Nombre total de cœurs :** 2
- **Cache de niveau 2 (par cœur) :** 256 Ko
- **Cache de niveau 3 :** 4 Mo
- **RAM :** 8 Go 1600 MHz DDR3
- **Système d'exploitation :** Windows 10

5.3 Présentation des graphes de test

Le tableau 5.1 regroupe les différents graphes de test utilisés pour l'évaluation de nos deux moteurs k^2-Grace et P-Grace. La première colonne classe les graphes selon leur type : Statique non orienté étiqueté, Statique non orienté non étiqueté, Statique orienté et Dynamique

orienté. Cette diversité revient à la particularité des graphes pris en entrées et leur caractéristiques qui diffèrent d'une méthode à une autre. Nous fournissons également une présentation des domaines d'application à partir desquels les graphes sont issus. Le tableau donne également les caractéristiques de chaque graphe représenté par le nombre de nœuds et le nombre de liens.

Type de graphe			Graphe	Nb. noeuds	Nb. liens	Description
Statique	Non Orienté	Étiqueté	vote-r (sub, 2011)	1266	6451	Représentation graphique d'une base de données présente dans le référentiel UCI
			diabetes (sub, 2011)	4500	4000	Représentation graphique d'une base de données présente dans le référentiel UCI
			ttt-win (sub, 2011)	5634	10016	Base de données Tic Tac Toe générée de manière exhaustive
			credit (sub, 2011)	14700	14000	Représentation graphique d'une base de données présente dans le référentiel UCI
		Non Étiqueté	ca-netscience (Rossi and Ahmed, 2015a)	379	914	Réseaux de collaboration
			Choclate (Koutra et al., 2015)	2899	5467	Graphe co-authorship
			Caida (cai, 2017)	26475	53381	Réseaux routier
	Orienté		bio (Leskovec and Krevl, 2014)	490	4598	Graphes de Biologie
			web-edu	3031	6474	Graphe du web
			web-polblogs (Rossi and Ahmed, 2015b)	644	2280	Graphe du Web
			wiki-Vote (Leskovec and Krevl, 2014)	7115	103689	Graphes des réseaux sociaux
Dynamique	Orienté		aves-weaver-social (Rossi and Ahmed, 2015b)	441		Réseaux sociaux
			reptilia-tortoise-network-bsv (Rossi and Ahmed, 2015b)	360		Réseaux d'interaction des animaux
			reptilia-tortoise-network-fi (Rossi and Ahmed, 2015b)	784		Réseaux d'interaction des animaux

TABLE 5.1 – Description des Graphes de Tests

Nous fournissons ci-après plus de détails sur la description des graphes et leurs domaines d'application :

— **Graphes de Biologie :** sont des graphes représentant les interactions entre les molécules.

— **Graphes du web :** Ces graphes modélisent les pages du web. Les nœuds de ce graphe représentent des pages html et ses arcs représentent l'existence d'un hyperlien d'une page vers une autre.

— **Graphes de réseaux sociaux :** Ce type de graphe représente les interactions entre individus dans différentes plateformes. Ils peuvent modéliser des relations symétriques entre eux comme l'amitié, avec des graphes non orientés, ou des relations asymétriques, comme l'envoie de messages, avec des graphes orientés.

— **Graphes de collaboration :** Ces réseaux représentent les individus travaillant sur un projet. Un lien entre deux individus dans ce cas signifie qu'ils travaillent ensemble sur une ou plusieurs parties du projet.

— **Graphes des réseaux routiers :** Les nœuds dans ces graphes représentent les villes et les liens représentent les routes existantes entre elles.

— **Graphes des réseaux d'interaction des animaux :** Ensembles de données du réseau d'interaction animale dans le monde réel. Ces données proviennent d'études publiées sur des animaux sauvages, captifs et domestiques.

5.4 Évaluation du moteur k^2-GraCE

Pour évaluer le moteur K^2-Grace, nous commencerons par étudier l'impact du type de représentation utilisée dans la construction de l'arbre sur le temps d'exécution. Nous nous intéresserons par la suite à l'étude de l'influence du paramètre K sur les performances de compression. Finalement, nous analyserons l'impact du module pré-traitement sur la qualité de compression.

5.4.1 Étude de l'effet de la représentation du graphe en entrée

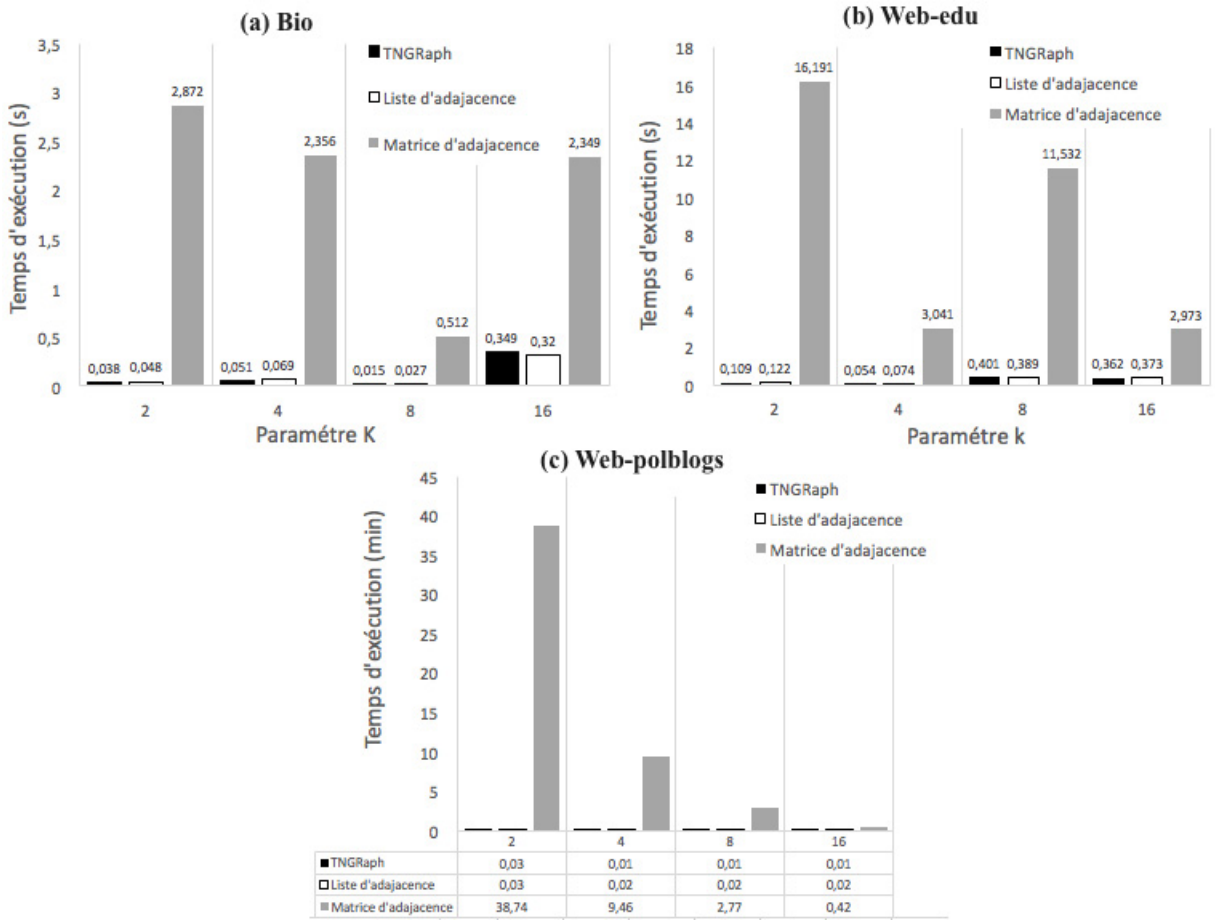


FIGURE 5.1 – Résultats de tests du moteur K^2-Grace selon les différents types de représentations du graphe en entrée.

Les graphiques de la figure 5.1 représentent la variation du temps d'exécution de la méthode k^2-trees selon les différentes représentations du graphe en entrée sur trois graphes de tests : Bio, web-edu et web-polblogs. Cette compression a été effectuée avec un ordre initial et avec différentes valeurs de k.

Nous constatons d'après les trois figures que l'utilisation des trois représentations donne des temps d'exécution totalement différents. Nous remarquons que si l'écart existant dans le temps d'exécution entre l'utilisation de la matrice d'adjacence et les deux autres structures peut être acceptable pour les petits graphes comme bio qui comptent 490 nœuds, il ne peut être toléré pour le cas des grands graphes comme web-edu et web-polblogs. En effet, nous avons obtenu, dans le cas du graphe bio, un temps d'exécution entre 2 et 3 secondes dans le cas de l'utilisation de la matrice d'adjacence et un temps d'exécution entre 0,015 et 0,4 seconde dans les deux autres cas. Cependant, le cas du graphe web-polblogs donne un temps d'exécution de 38 minutes pour k=2 lorsque l'arbre est construit en utilisant la matrice d'adjacence et un temps d'exécution de 0,03 secondes dans le cas des deux autres structures. Cette différence est due au temps que prend la machine pour charger la matrice d'adjacence en mémoire centrale qui devient de plus

en plus important lorsque la taille du graphe est grande. De ce fait, nous utiliserons uniquement la structure TNGraph pour la suite des tests.

5.4.2 Analyse de l'impact du paramètre k

Les figures 5.3 et 5.2 représentent respectivement l'évolution du nombre de bits par nœud et du ratio de compression en fonction des différentes valeurs du paramètre K. La compression a été appliquée sur différents graphes de test statiques issus de plusieurs domaines et avec différentes caractéristiques pour avoir une meilleure analyse. Les graphes utilisés dans ce test sont : bio, web-edu, web-polblogs, wiki et soc-Epinions. Cette compression a été effectuée avec un ordre initial et sans aucun pré-traitement. Nous avons varié le paramètre K entre 2 et 16.

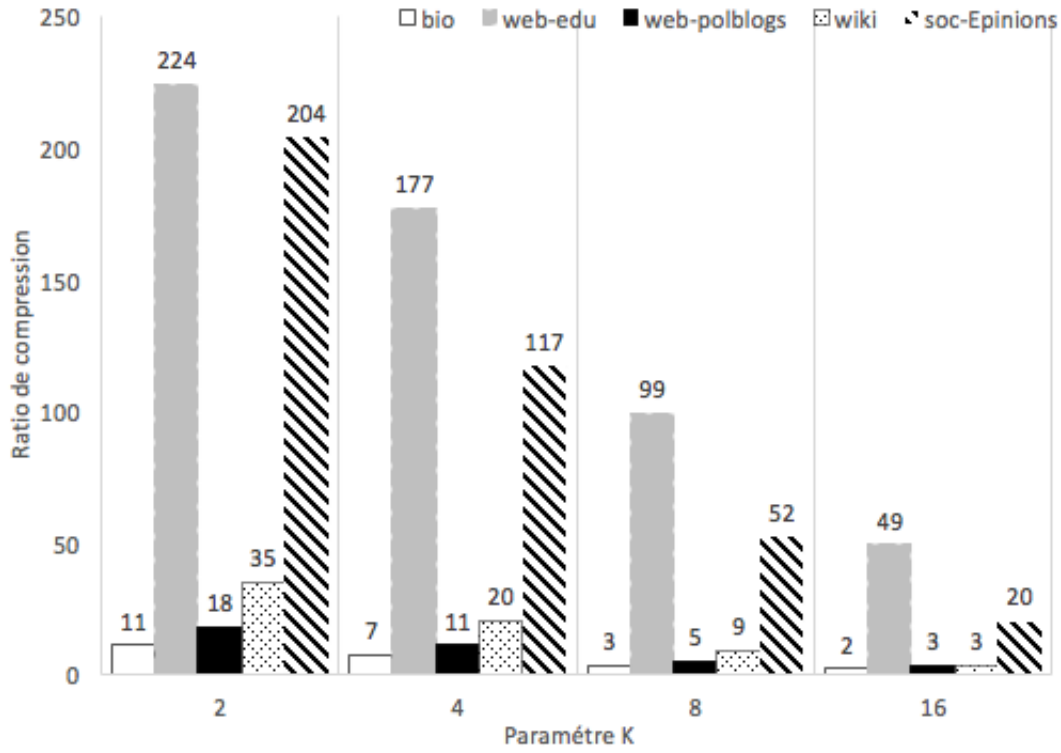
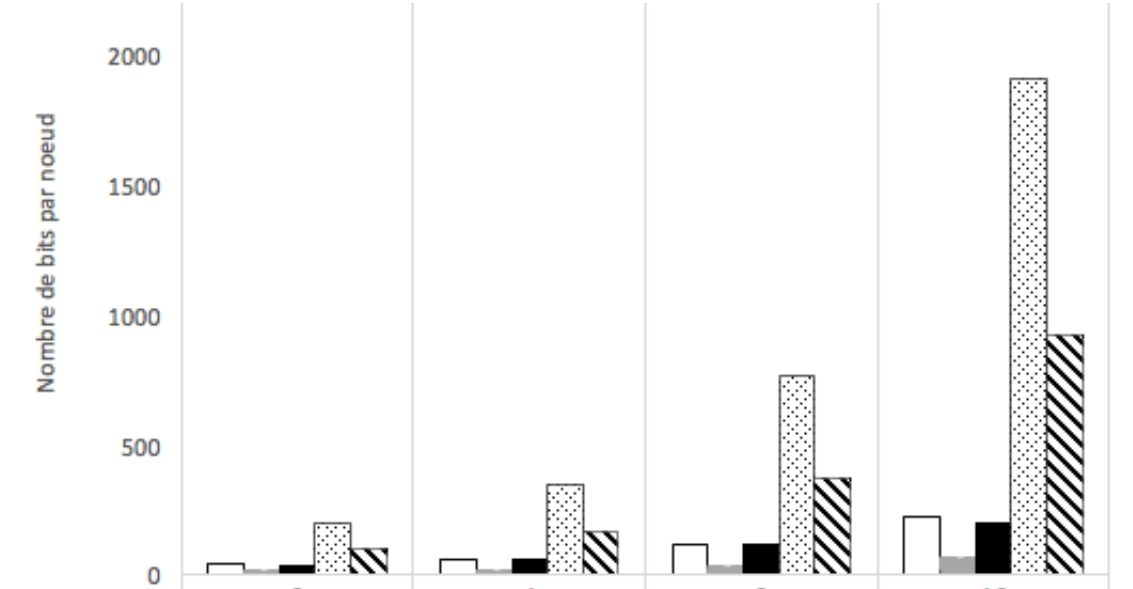


FIGURE 5.2 – Résultats de compression de K^2-Grace : Ratio de compression du moteur K^2-Grace en fonction du paramètre K (cas statique)

D'après les résultats obtenues dans la figure 5.2, nous remarquons que le ratio de compression diminue quand k prend des valeurs plus grandes et atteint sa valeur minimale (exemple : 20 pour soc-Epinions) quand k=16. Ainsi, nous constatons que plus le k est petit plus le ratio de compression est optimal. En effet, pour des petites valeurs de k, l'arbre a plus de niveaux, mais il est moins large et le nombre de ses feuilles est plus petit (petites sous matrices finales) ce qui réduit la taille de l'arbre et donne une meilleur qualité de compression. Nous avons pu aussi confirmer cette hypothèse avec le nombre de bits par nœuds obtenu dans les différents cas et qui est représenté dans la figure 5.3. Nous observons que le nombre de bits par nœud augmente de manière exponentielle en fonction de k et atteint sa valeur maximale (exemple 917 938 bits

par nœud dans le cas du graphe wiki) quand k est à 16. Le nombre de bits augmente avec les grandes valeurs de k reflétant ainsi une augmentation dans la taille du graphe en sortie.



	2	4	8	16
bio	37,91	59,13	115,14	223,22
web-edu	12,39	17,07	30,41	60,81
web-polblogs	35,34	57,58	116,55	203,05
wiki	197,67	343,51	765,74	1907,64
soc-Epinions	94,93	163,43	370,57	917,94

FIGURE 5.3 – Résultats de compression de K^2-Grace : Nombre de bits par nœuds en fonction du paramètre K

Nous nous sommes aussi intéressées à l'étude de l'effet du paramètre k sur les performances de compression dans le cas des graphes dynamiques. Nous présentons dans la figure 5.4 les résultats obtenus pour ce type de graphes. Les graphes utilisés dans ce test sont : aves-weaver-social, reptilia-tortoise-network-bsv et reptilia-tortoise-network-fi. Ces graphes sont représentés avec leur captures initiales (sans aucun pré-traitement).

A partir de la figure 5.4, nous remarquons une dégradation des performances dans le cas général. Cependant, nous obtenons les meilleurs performances pour k=8 dans le cas des deux graphes aves-weaver-social et reptilia-tortoise-network-bsv avec un 1,87 bits par nœud pour le premier graphe et 1,78 bits par nœud pour le deuxième graphe. De ce fait, l'hypothèse que nous avons avancée dans le cas des graphes statiques ne peut être généralisée dans le cas des graphes dynamiques où nous avons obtenu différentes valeurs optimales de k dans les différents graphes.

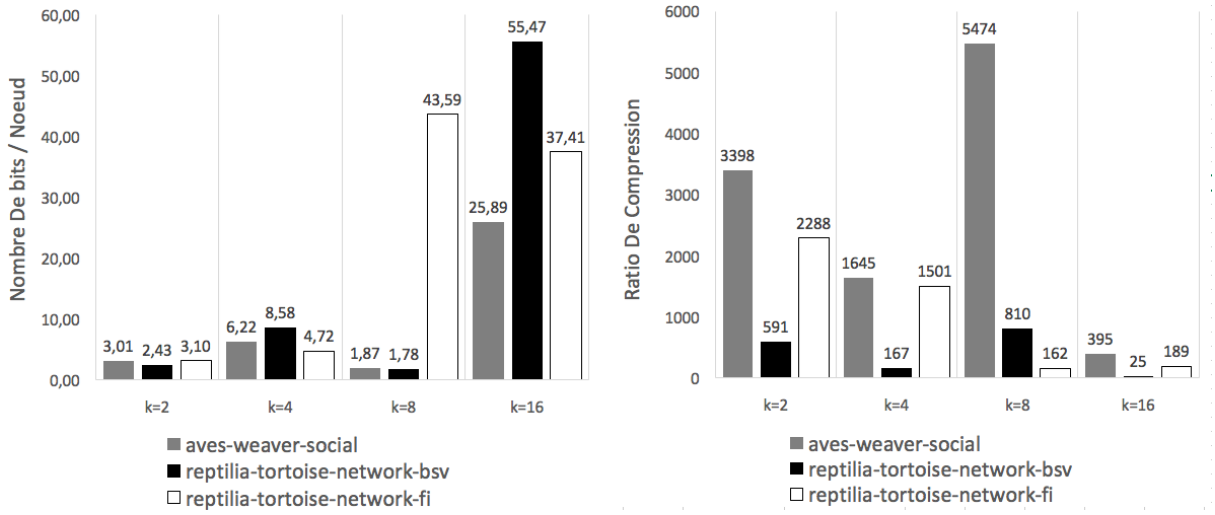


FIGURE 5.4 – Résultats de compression de K^2-Grace : Nombre de bits par nœud et Ratio de compression du moteur en fonction du paramètre K (cas dynamique)

La figure 5.5 représente l'évolution du temps d'exécution en fonction du paramètre k. Dans le cas des graphes statiques, nous notons que le temps de compression varie de manière aléatoire et ne dépend pas de la valeur de k. En effet, dans le cas des graphes bio et soc-Epinions nous obtenons des temps d'exécution qui changent de manière aléatoire. Tant dis que dans le cas du graphe web-edu nous obtenons un temps d'exécution qui diminue en fonction de k contrairement au graphe wiki-Vote dans lequel le temps d'exécution augmente en fonction du paramètre k. Dans le cas des graphes dynamiques, nous remarquons que le temps d'exécution diminue jusqu'à k=8 où il atteint sa valeur minimale pour k=8 et commence à augmenter à partir de cette valeur.

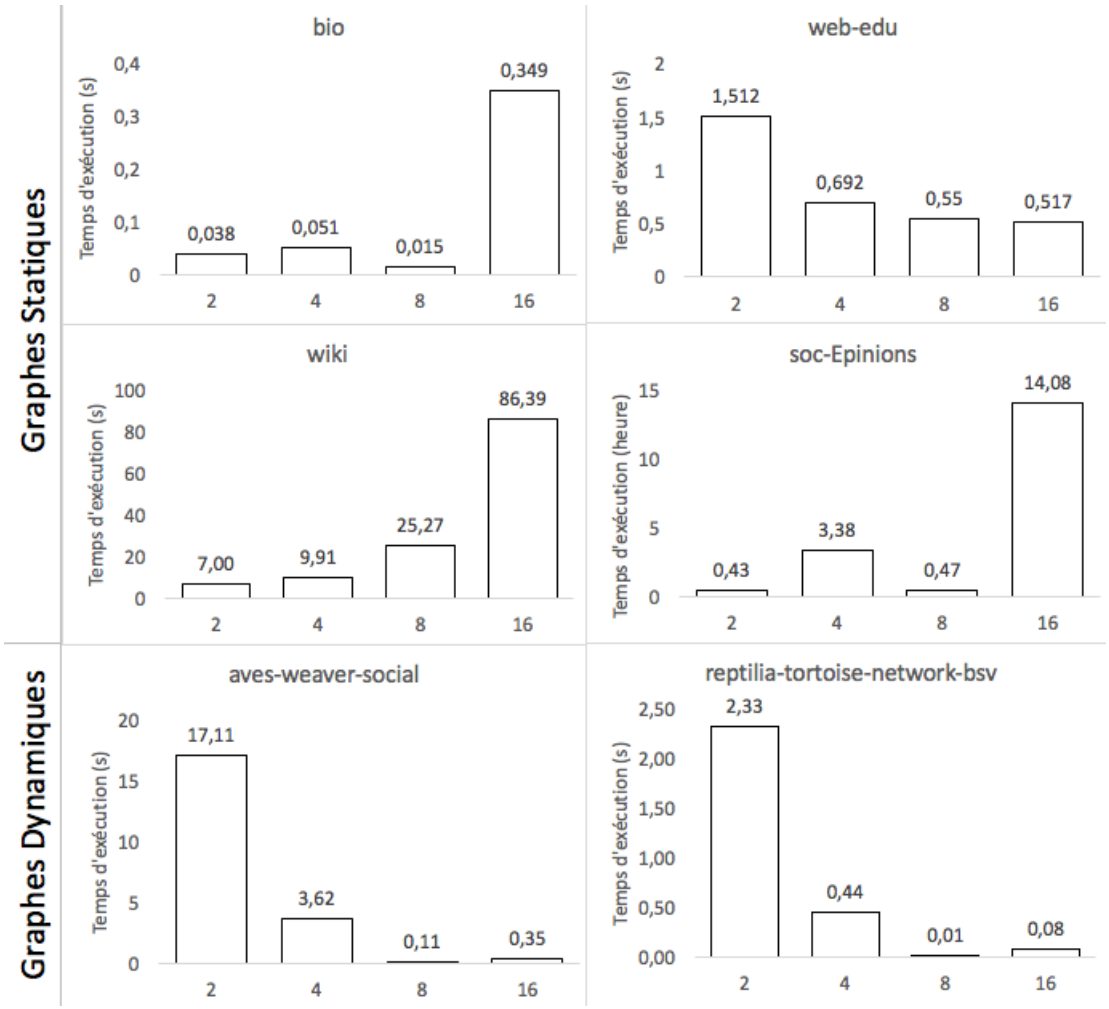
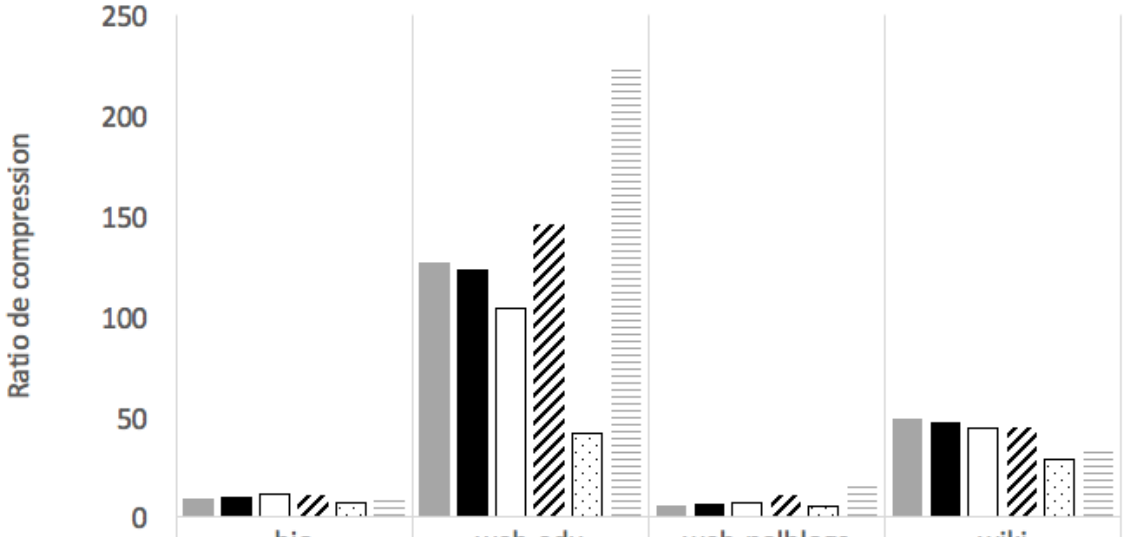


FIGURE 5.5 – Résultats de compression de K^2-Grace : Temps de compression en fonction de K

5.4.3 Étude de l'apport de la phase de pré-traitement

Durant cette partie de l'évaluation du moteur k^2-GraCE, nous nous focaliserons sur l'étude de l'effet de l'utilisation des techniques du module « pré-traitement » du moteur k^2-GraCE. Nous commencerons par étudier l'effet des différents ordres des nœuds dans le cas des graphes orientés. Nous enchainerons par la mesure du gain apporté par la considération de la partie supérieure uniquement de la matrice d'adjacence dans la construction de l'arbre k^2-tree pour le cas des graphes non orientés. Finalement, nous présenterons les tests relatifs à l'utilisation d'une matrice de différence dans le cas des graphes dynamiques.

5.4.3.1 Étude de l'effet de l'ordre



	bio	web-edu	web-polblogs	wiki
BFS	9	127	6	49
DFS	10	123	7	47
Gray	11	104	7	44
Lexical	11	146	11	45
Random	7	42	5	29
initiale	11	224	18	35

FIGURE 5.6 – Résultats de compression de K^2-Grace :Ratio de compression selon l'ordre des nœuds

La figure 5.6 représente l'évolution du ratio de compression du moteur k^2-Grace sur quatre graphes de test : bio, web-edu, web-polblogs et wiki. Cette compression a été effectuée en utilisant cinq ordres différents : BFS, DFS, Gray, Lexical et Random avec comme paramètre k=2. Nous remarquons que dans le cas des deux graphes web-edu et web-polblogs qui sont des graphes du web , nous avons obtenu une détérioration des performances de compression pour les différents ordres. Nous justifions cela par le fait que les arbres k^2-trees ont été au départ conçus pour exploiter les propriétés de l'ordre initial des nœuds dans la matrice d'adjacence. Cependant, nous remarquons que dans le cas du graphe wiki-Vote, qui est un graphe issu du domaine des réseaux sociaux, nous avons obtenu une amélioration du ratio initial d'un facteur de 1,4 avec les ordres : DFS, BFS, Gray et Lexical. Pour ce graphe, nous avons obtenu un ratio de compression optimal de 49 pour l'ordre BFS. Dans le cas du graphe bio, nous remarquons que seuls les ordres Lexical et gray ont permis de garder les même performances que l'ordre initial. Nous constatons donc que ces ordres donnent de bonnes performances avec les graphes qui ne sont pas issus du domaine du web. En effet, comme nous l'avons déjà noté s'ils ne permettent pas d'améliorer les performances ils ne les détériorent pas.

5.4.3.2 Étude de l'effet de l'utilisation de la partie supérieure uniquement de la matrice d'adjacence

Le tableau 5.7 donne les résultats de la compression du moteur k^2-Grace appliqué sans et avec pré-traitement sur des graphes non orientés avec un k=2 et un ordre initial des nœuds. Les graphes utilisés sont ChocWIki, ca-netscience, Caida.

Graphe	Avec Pre-traitement			Sans Pre-traitement		
	Ratio	Bits\nœud	Temps d'exe	Ratio	Bits\nœud	Temps d'exe
ChocWiki	85	33,6089	2,439	42	66,9653	1,796
ca-netscience	21	17,5092	0,037	11	32,1055	0,036
Caida	879	29,5004	103,562	454	58,2812	107,399

FIGURE 5.7 – Résultats de compression de K^2-Grace avec/sans pré-traitement (cas non orienté)

Nous observons que le ratio de compression et le nombre de bits par nœud sont nettement meilleurs avec le pré-traitement. Le pré-traitement appliqué sur le graphe non orienté maximise les zones vides dans la matrice d'adjacence en diminuant le nombre de 1 de moitié ce qui réduit la taille de l'arbre et produit une meilleure qualité de compression. Nous remarquons cela dans le ratio de compression qui double pour les trois graphes. Le temps d'exécution est presque le même dans les deux exécutions car nous construisons la matrice triangulaire supérieure au moment de la lecture directement. Le léger écart noté entre les deux exécutions est du aux comparaisons des indices sources et destinations effectuées pour construire la matrice.

La figure 5.8 présente le gain obtenu après l'application du pré-traitement sur les graphes non orientés, nous remarquons que le gain est aux alentours du double pour tous les graphes. Cela prouve que le pré-traitement améliore clairement la qualité de compression tout en offrant la possibilité d'extraire toutes les informations incluses dans le graphe original.

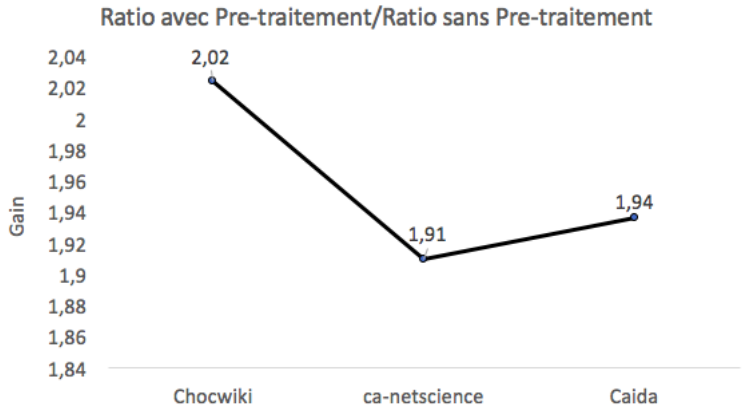


FIGURE 5.8 – Gain de compression de K^2-Grace avec le pré-traitement

5.4.3.3 Étude de l'effet de l'utilisation d'une matrice de différence

Dans cette partie, nous évaluerons l'effet de l'utilisation de la matrice de différence pour la construction de l'arbre sur les performances de compression. Comme les tests précédents ont montré qu'il n'existe pas de valeur optimale pour k qui améliore ces derniers, nous effectuerons les tests de cette phase pour différentes valeurs de k. Le tableau de la figure 5.9 donne les valeurs des trois (03) métriques considérées en fonction des valeurs de k.

Graphe	Avec Pre-traitement			Sans Pre-traitement		
	Ratio	Bits\nœud	Temps d'exécution (s)	Ratio	Bits\nœud	Temps d'exécution (s)
			K=2			
Aves-weaver-social	3398	3,0114	17,112	1735	5,89663	16,525
Reptilia-tortoise-network-bsv	591	2,433	2,327	591	3,12	2,341
Reptilia-tortoise-network-fi	2288	3,0953	48,4	2288	5,0721	49,276
			K=4			
Aves-weaver-social	1645	6,2202	3,62	844	12,119	3,672
Reptilia-tortoise-network-bsv	167	8,5778	0,443	132	10,88	0,448
Reptilia-tortoise-network-fi	1501	4,7166	3,834	1027	6,8919	3,842
			K=8			
Aves-weaver-social	5474	1,86966	0,105	3094	3,307	0,129
Reptilia-tortoise-network-bsv	810	1,778	0,012	810	17,78	0,041
Reptilia-tortoise-network-fi	162	43,5883	3,699	105	67,3342	3,741
			K=16			
Aves-weaver-social	395	25,8876	0,353	222	46,0225	0,369
Reptilia-tortoise-network-bsv	25	55,4667	0,082	24	59,733	0,097
Reptilia-tortoise-network-fi	189	37,4079	0,318	123	57,5756	0,35

FIGURE 5.9 – Résultats de compression de K^2-Grace avec/sans pré-traitement (cas dynamique)

Nous remarquons que nous obtenons dans tous les cas un temps d'exécution presque égal. Le léger écart noté dans le cas de l'utilisation du pré-traitement représente le temps nécessaire pour le calcule de la matrice de différence. Pour le graphe aves-weaver-social, nous remarquons que le ratio de compression (resp. le nombre de bits par nœud) diminue (resp. augmente) de moitié (resp. du double) pour toute les valeurs de k. Pour le deuxième graphe reptilia-tortoise-network-bsv, nous observons bien que les performances (ratio et nombre de bits par lien) restent les même dans les deux cas sans et avec pré-traitement. Tant dis que dans le cas du troisième graphe, nous remarquons que nous obtenons les mêmes performances pour k=2 et que nous perdons en performances quand k prend des valeurs plus grandes. Pour mieux illustrer ces observations nous fournissons dans la figure 5.10 la courbe représentant le rapport entre le ratio de compression avant et après l'utilisation du pré-traitement.

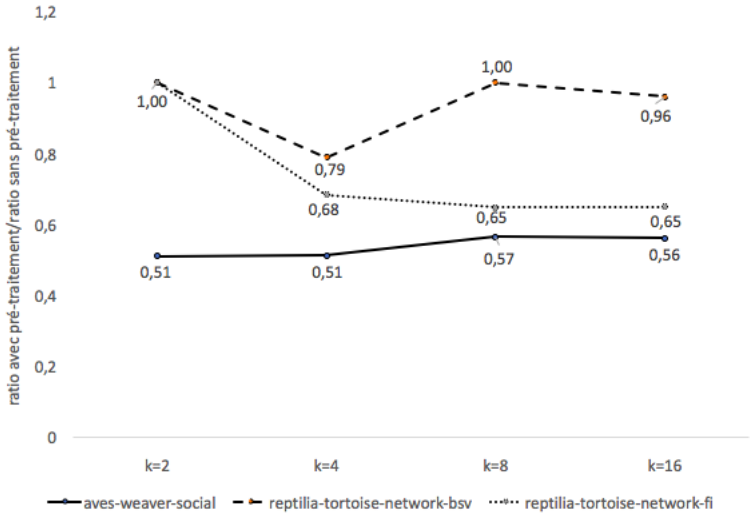


FIGURE 5.10 – Ratio de avec pré-traitement/ Ratio de sans pré-traitement

Nous constatons donc que l'utilisation de ce module n'a pas amélioré les performances de compression dans le cas des trois graphes. Cela est du à la nature de leurs captures successives qui ne partagent presque aucun lien en commun engendrant ainsi soit la même matrice initiale ou une matrice de différence plus dense que celle du graphe original.

5.5 Évaluation du moteur P-GraCE

Nous évaluerons, dans cette partie, les quatre approches de compression par extraction de motifs incluses dans le moteur P-GraCE. Dans un premier temps, nous étudierons aussi le rendement du beam-search en fonction du niveau d'agrégation. Nous enchainerons par la suite avec l'évaluation des performances de compression par extraction de motifs en utilisant des méthodes de détection de communautés. Nous aborderons par la suite l'approche basée sur les propriétés de la matrice d'adjacence où nous allons étudier l'influence du type et de la taille des motifs sur les mesures de performances. Finalement, nous présenterons les différents tests relatifs à la méthode de compression que nous avons proposée : DDSM qui est destinée aux graphes dynamiques.

5.5.1 Évaluation de l'algorithme du beam-search :

Durant cette section, nous considérons le cas des graphes non orientés et étiquetés. Nous évaluerons le rendement de compression de l'algorithme du beam-search sur ces derniers tout en prêtant attention au temps d'exécution. Le tableau ci-dessous résume les résultats obtenus.

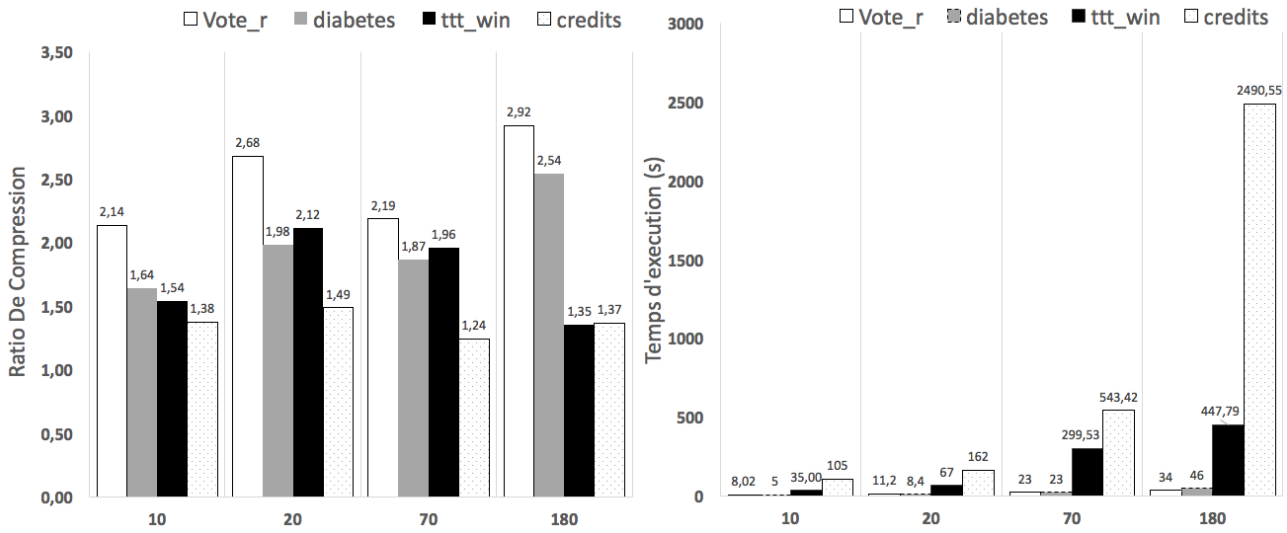


FIGURE 5.11 – Résultats des tests du beam-search.

La figure 5.11 illustre les résultats de tests obtenus pour différents graphes en fonction du nombre d'itérations de l'algorithme (le niveau d'agrégation). Nous constatons une amélioration du ratio de compression dans la majorité des cas. En contre partie, nous remarquons que le temps d'exécution augmente surtout dans le cas des grands graphes (exemple du graphe credits). Le choix du nombre d'itérations (niveau de la hiérarchie d'agrégation) dépend des performances voulues. Cependant, nous pensons qu'un nombre d'itération égale à 70 offre déjà un bon compromis entre le temps d'exécution et le ratio de compression.

5.5.2 Évaluation des techniques basées sur les méthodes de clustering :

Dans cette partie, nous proposons de tester les méthodes de compression basées sur les techniques de clustering en faisant varier différents paramètres. Dans un premier temps, nous fixerons Slashburn comme méthode d'extraction de motifs et nous évaluerons les performances de compression obtenus avec la méthode de sélection *topk* en considérant à chaque fois une portion uniquement des structures découvertes (k=25%,k= 33,33%, k=50%, k=100%). Par la suite, nous étudierons l'effet du choix de la méthode de sélection sur la qualité de compression en utilisant la même méthode d'extraction de motifs. Finalement, nous analyserons l'effet de la méthode de clustering sur les résultats en fixant cette fois la méthode *Step* comme méthode de sélection.

5.5.2.1 Étude de l'influence du nombre de structures sélectionnées :

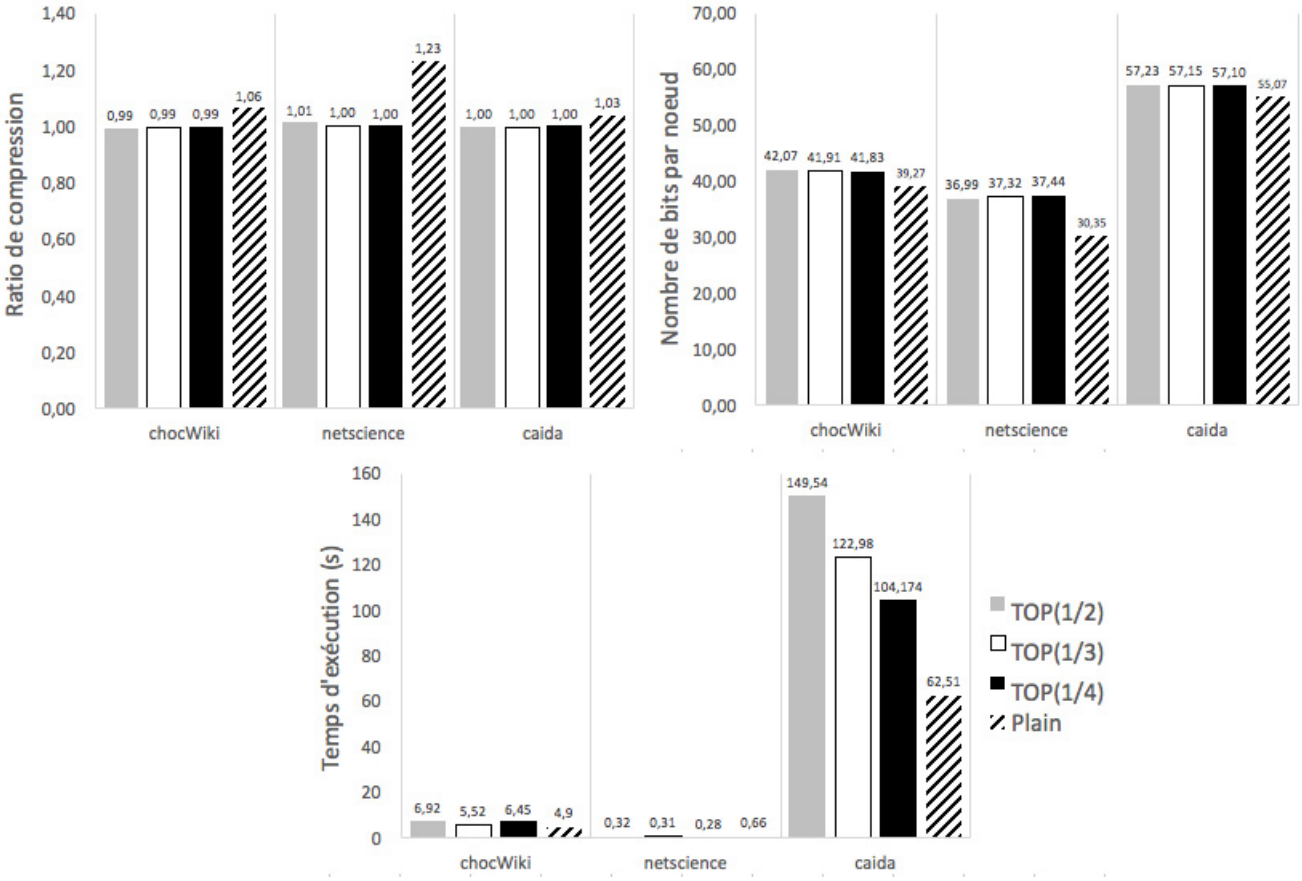


FIGURE 5.12 – Résultats des tests de l'influence du nombre de structures sélectionnées.

La figure 5.12 illustre les mesures des trois métriques : ratio de compression, nombre de bits par nœud et le temps d'exécution en considérant différentes portions des sous-structures découvertes. Nous remarquons que le ratio de compression est presque équivalent pour les différentes valeurs de k. Cependant, nous observons bien que la considération de toutes les structures donne un meilleur ratio de compression de 1,06 pour chocWiki, de 1,23 pour netscience et de 1,03 pour le graphe Caida. Cette hypothèse peut être facilement confirmée avec le nombre de bits par noeud qui atteint sa valeur minimale avec la méthode *Plain*. Quant au temps d'exécution, nous constatons que la méthode *Plain* offre toujours de meilleur performance. En effet, la méthode *topk* ordonne les sous-structures selon le gain d'encodage local avant la sélection engendrant ainsi un temps supplémentaire contrairement à la méthode *plain* qui sélectionne toutes les sous-structures sans aucun ordre.

5.5.2.2 Étude de l'influence de la méthode de sélection :

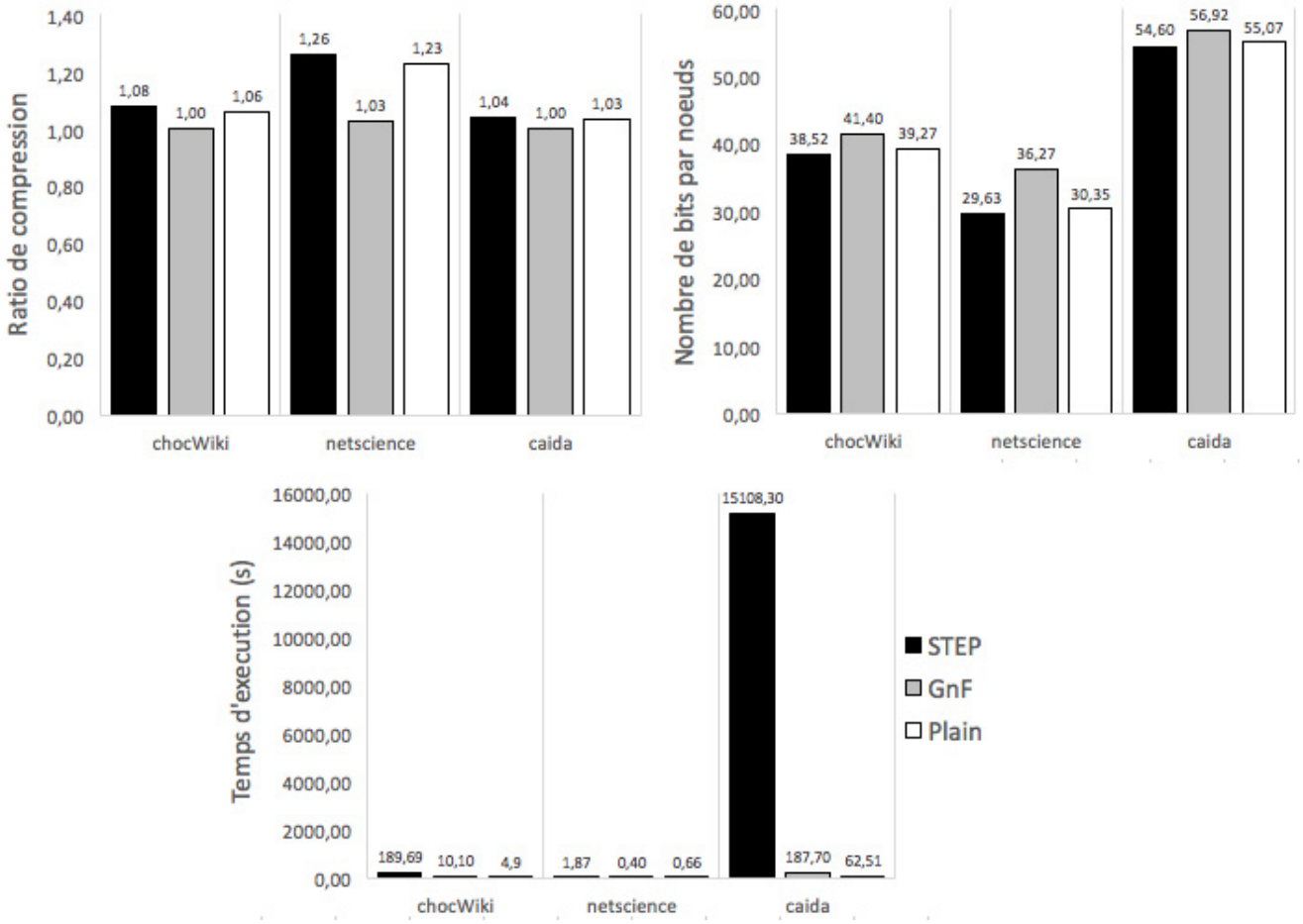


FIGURE 5.13 – Résultats des tests de l'influence de la méthode de sélection.

Dans cette partie, nous présenterons les tests relatifs à l'utilisation de trois méthodes de sélection : Step, GNF et *Plain*. La figure 5.13 résume les résultats obtenus sur trois graphes non orientés statiques.

Nous remarquons que la méthode Step donne de meilleur performances de compression avec un ratio maximale de 1,26 pour le graphe netscience qui compte 379 nœuds. En effet, si nous voulons classer ces méthodes de sélection selon les résultats du ratio et le nombre de bits par nœud nous obtenons le classement suivant : Step, *Plain* et GNF et ce pour tout les graphes utilisés dans ce test. Par contre, nous remarquons que la méthode Step donne un temps d'exécution qui est très élevé par rapport aux deux autres méthodes surtout dans le cas des grands graphes. Par exemple dans le cas du graphe Caida, nous avons obtenu un temps d'exécution de plus de quatre (04) heures avec la méthode Step contre un temps d'exécution entre une (01) et deux (02) minutes pour les deux autres méthodes pour une amélioration de 0,04 dans le ratio de compression.

5.5.2.3 Étude de l'influence de la méthode de clustering :

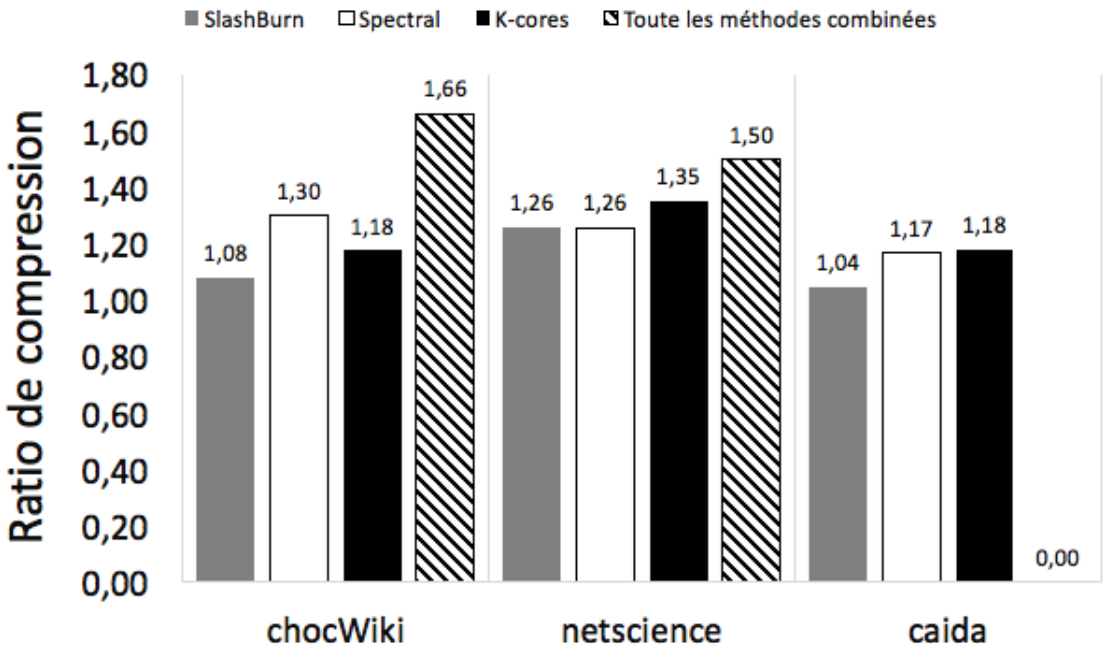


FIGURE 5.14 – Résultats des tests de l'influence de la méthode de clustering .

Dans ce dernier test, nous nous sommes intéressées à voir le gain apporté par la méthode de clustering dans le ratio de compression. La figure 5.14 illustre les résultat obtenus. Nous avons considéré trois (03) méthodes de clustering : Slashburn, Spectral et K-cores.

Selon le graphe, les deux méthodes de compression Spectral et K-Cores donnent de meilleur résultat que SLashburn pour les différents graphes. En effet, dans le cas du graphe ChocWiki la méthode spectrale donne le meilleur ratio (1,30). Tant dis que dans le cas des deux autres graphes, netscience et Caida, la méthode K-cores donne un meilleur ratio de 1,35 et 1,18 respectivement. Cependant, la combinaison des trois méthodes d'extraction de motifs donne une amélioration significative du ratio dans les trois graphes de test.

5.5.3 Étude de l'influence du type et de la taille des motifs dans les méthodes basées sur la matrice d'adjacence :

Durant cette partie, nous présenterons les résultats obtenus lors de l'évaluation de l'extraction de motifs à partir de la matrice d'adjacence. La figure 5.15 illustre l'évolution du ratio de compression en fonction de la taille du motif pour les trois classes offertes par le moteur P-GracE.

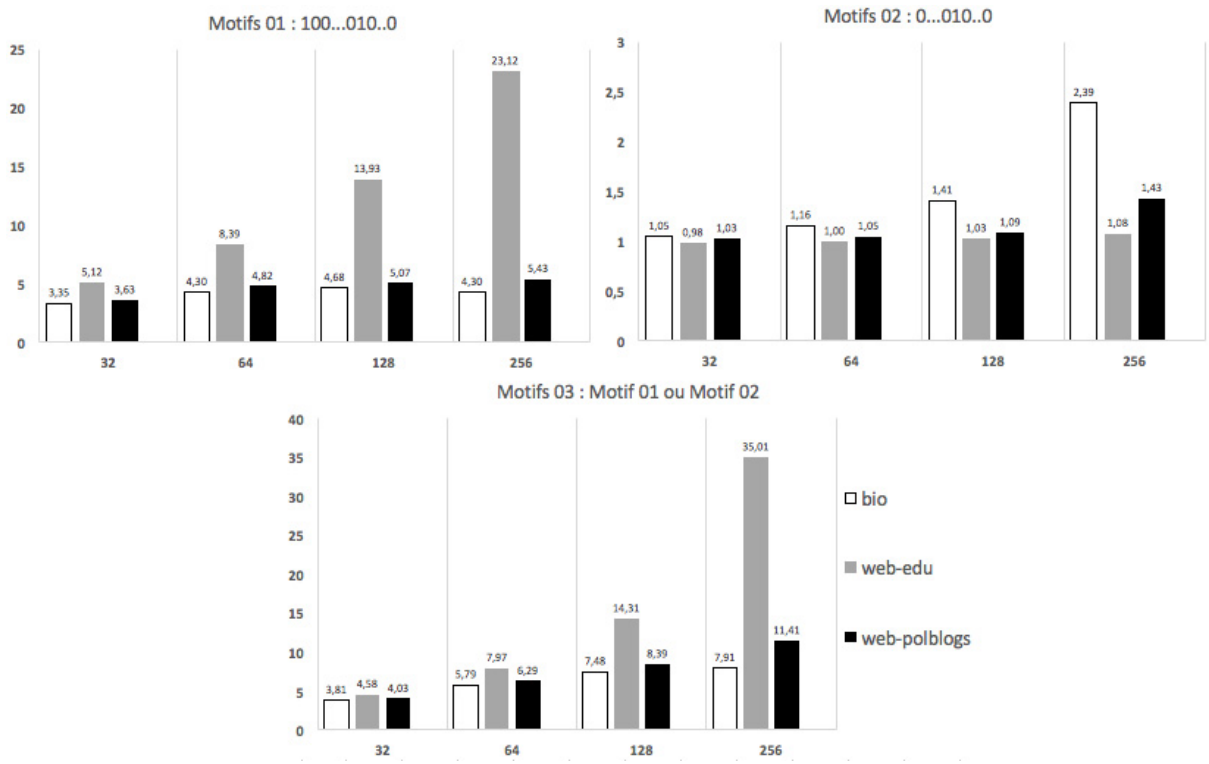


FIGURE 5.15 – Le ratio de compression en fonction de la taille du motif.

Dans le cas de la première et la troisième classe de motifs, nous remarquons une amélioration du ratio de compression lorsque leur taille prend des grandes valeurs. Cette observation est du au gain obtenu en remplaçant des motifs de taille 2^n bits par n bits pour la premier type de motfs et *n+1* bits pour le troisième, par exemple les motifs de taille 256bits seront remplacés par uniquement 8bits dans le cas de la classe (01) et par 9bits dans le cas de la classe (03). Pour la deuxième classe de motifs, le ratio de compression est resté presque constant pour différentes valeurs de la taille des motifs. Nous remarquons aussi que ce dernier au tour de 1 dans la majorité des cas signifiant ainsi que le taille du graphe en sortie est presque équivalente à la taille du graphe initial. Ce résultat ne peut être obtenu que dans le cas où peu de motifs ont été trouvés dans la matrice d'adjacence. Nous constatons donc que les motifs de cette deuxième classe ne sont pas fréquents dans les matrices d'adjacence des graphes réels.

5.5.4 Évaluation de la méthode DDSM :

Nous présenterons dans cette partie, les tests relatifs à la méthode que nous avons proposée. Nous étudierons les performances obtenues en fonction du nombre de permutations utilisées dans le processus de découverte et d'étiquetage des sous-structures.

Graphe	Ratio de compression				Temps d'exécution			
	10	50	120	250	10	50	120	250
Aves-weaver-social	45	123	129	119	0,816	0,474	0,51	0,649
Reptilia-tortoise-network-bsv	16	13	13	15	0,044	0,045	0,047	0,049
Reptilia-tortoise-network-fi	78	289	300	299	0,0787	0,653	0,719	0,754

FIGURE 5.16 – Résultats de la méthode DDSM

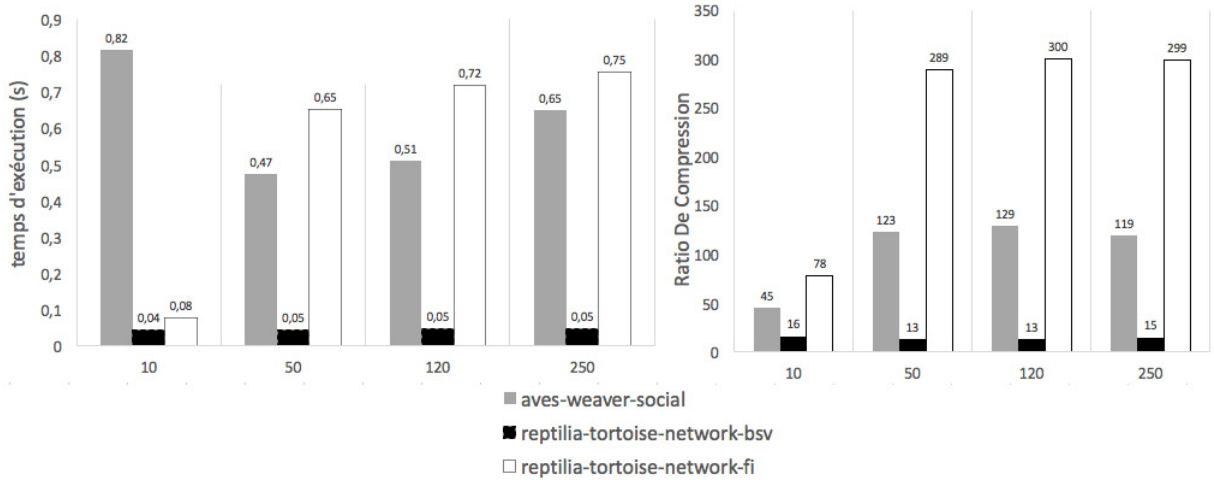


FIGURE 5.17 – Résultats des tests de la méthode DDSM.

Nous remarquons que la méthode proposée offre de très bon résultats par rapport au temps d'exécution qui ne dépasse pas une (01) seconde pour toutes les exécutions. Pour le ratio de compression, nous constatons une amélioration significative qui commence à se stabiliser à partir du moment où le nombre de permutations dépasse 50.

Comportement Temporel des Sous-structures	Aves-weaver-social			Reptilia-tortoise-network-bsv			Reptilia-tortoise-network-fi		
	Cliques	BiCliques	Autres	Cliques	BiCliques	Autres	Cliques	BiCliques	Autres
Constante	0	0	0	1	0	1	0	0	0
OneShot	0	0	0	0	0	0	0	4	0
Periodic	0	0	30	0	1	7	1	9	10
Ranged	0	0	0	0	1	5	0	11	28
Flickering	0	0	1	0	0	4	0	5	19

FIGURE 5.18 – Statistique des différentes types de sous-structures découvertes

Nous fournissons dans la figure 5.18 les différentes sous-structures découvertes pour un exemple d'exécution avec 50 permutations. Dans le cas du graphe Aves-weaver-social, nous

remarquons que nous obtenons 30 sous-structures denses qui apparaissent de manière périodique et une seule sous-strcuture qui apparait de manière aléatoire. Les sous-structures périodiques représentent donc des utilisateurs du réseaux qui s'envoient des messages périodiquement (exemple : réunion dans un contexte de télé-travail, des groupes d'étude, ...). Dans les deux autres réseaux qui font parties des réseaux d'interaction des animaux nous remarquons des sous-structures constantes qui représentent peut être des animaux appartenant à un même troupeau. Nous observons aussi des sous-strcutures périodiques qui peuvent être interprétées comme des chasses d'animaux pour la nourriture. Un autre type sont les sous-structures qui apparaissent dans une seule période représentant en générale le phénomène de couplage chez les animaux dans des périodes bien précises.

DDSM offre ainsi non pas uniquement la compression du graphe initial mais aussi la possibilité de l'analyser et d'interpréter les informations les plus pertinentes incluses dans ce dernier.

5.6 Comparaison entre les différentes méthodes :

Après avoir évaluer les différentes configurations possibles des deux moteurs : k^2-GraCE et P-GraCE, nous nous intéresserons dans cette partie à établir une étude comparative entre les méthodes implémentées dans les deux moteurs ainsi que les classes auxquelles elles appartiennent. Cette partie sera donc organisée en trois sous-sections chacune englobant les méthodes de compression acceptant le même type de graphe en entrée.

5.6.1 Évaluation des méthodes destinées aux graphes orientés statiques :

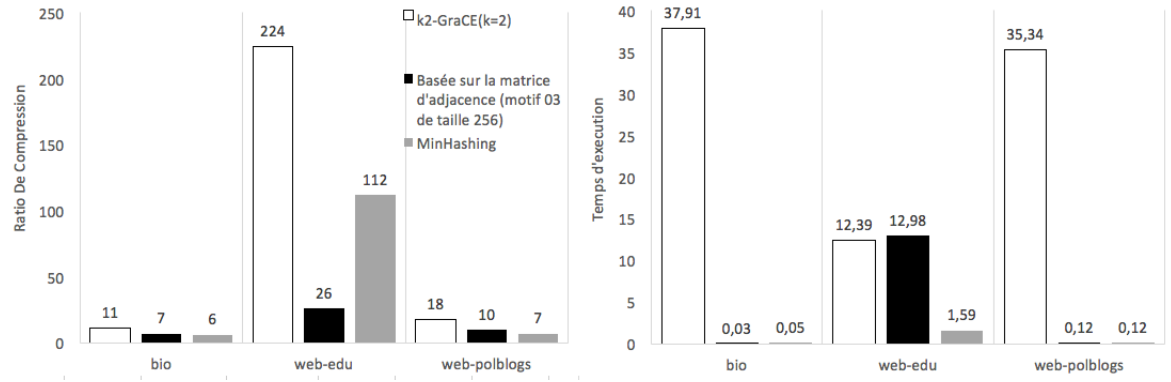


FIGURE 5.19 – Comparaison entre les méthodes destinées aux graphes statiques orientés.

Nous considérons dans cette partie les graphes orientés : bio, web-edu et web-polblogs, pour comparer entre trois méthodes de compression destinées pour ce type de graphe : k^2-GraCE, l'approche du Minhashing et enfin l'approche basée sur la matrice d'adjacence. Nous utilisons pour cela deux métriques : le ratio de compression (figure à gauche) et le temps d'exécution (figure à droite). Nous remarquons que le moteur k^2-GraCE donne les meilleurs résultats en terme d'espace mémoire (le meilleur ratio). Cependant, il donne le plus grand temps d'exécution.

Nous observons aussi que les deux autres méthodes donnent des résultats presque équivalents dans le des graphes bio et web-polblogs, et que la technique du Minhashing donne un meilleur résultat dans le cas du graphe web-edu avec un ratio de 112 vs un ratio de 26 pour la méthode basée sur la matrice d'adjacence.

5.6.2 Évaluation des méthodes destinées aux graphes non orientés statiques :

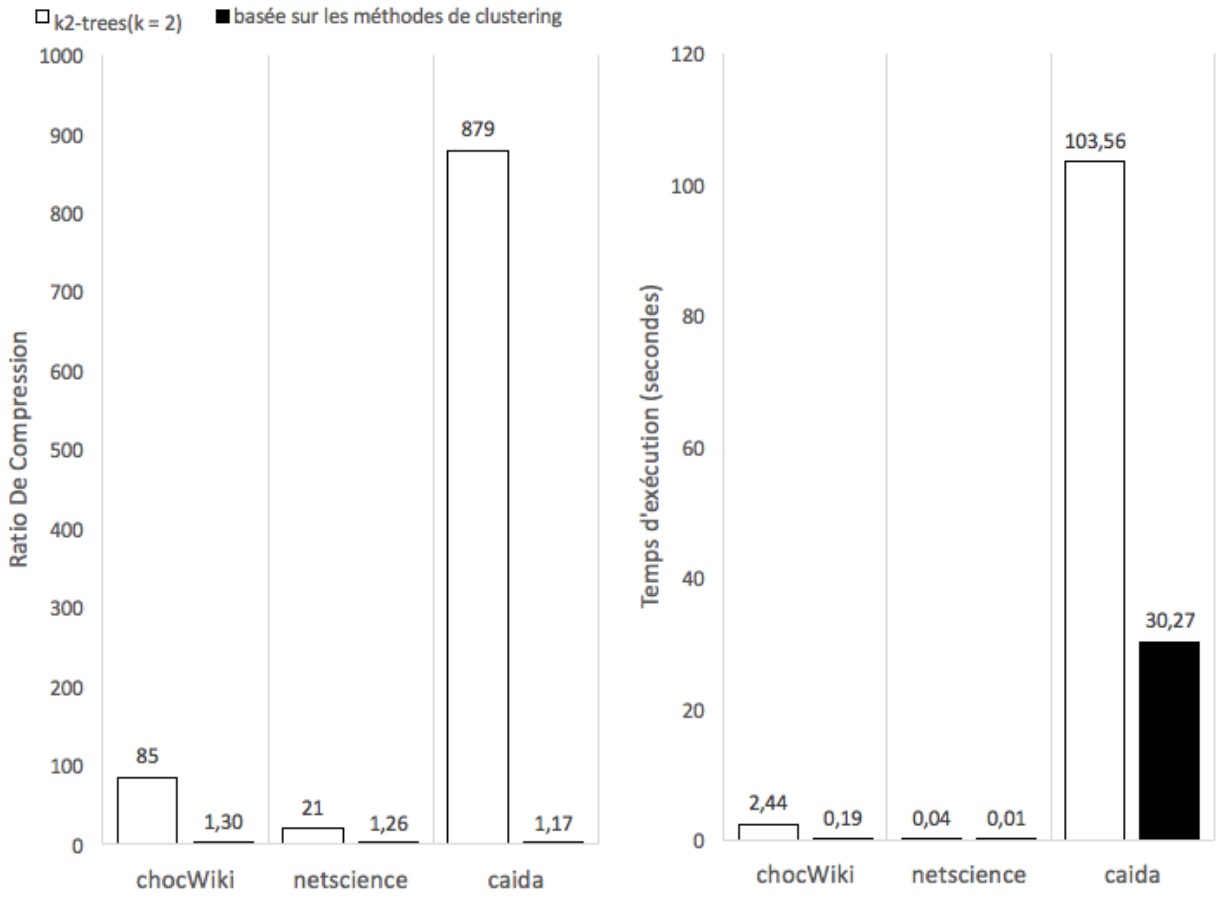


FIGURE 5.20 – Comparaison entre les méthodes destinées aux graphes statiques non orientés.

La figure 5.20 donne les résultats de compression : ratio et temps de compression des deux méthodes de compression des moteur P-GraCE et k^2-GraCE dans le cas des graphes non orientés statiques. Nous utilisons dans ce test les graphes : ChocWiki, Caida et netscience. Pour le moteur k^2-GraCE, nous fixerons la valeur de k à 2 en considérant uniquement la partie supérieure de la matrice d'adjacence. Tant dis que pour le moteur P-GraCE, nous utiliserons l'approche basée sur la méthode de clustering avec Spectral comme méthode de clustering et la méthode *step* comme méthode de sélection.

D'après l'illustration, nous observons que le ratio de compression du moteur k^2-GraCE est nettement meilleur que celui du moteur P-GraCE. Nous citons l'exemple du graphe Caida qui comptent 26475 nœuds et 53381 arêtes où nous obtenons un ratio de compression de 879 dans k^2-GraCE contre un ratio de 1,17 avec le moteur P-GraCE. Quant au temps d'exécution, nous

remarquons que le moteur P-GraCE est beaucoup plus rapide que à k^2-GraCE surtout lorsque la taille du graphe devient grande (exemple du graphe Caida).

5.6.3 Évaluation des méthodes destinées aux graphes dynamiques :

Dans cette partie, nous allons comparer entre les deux méthodes de compression destinées aux graphes dynamiques qui sont incluses, la première dans le moteur k^2-GraCE et la deuxième dans le moteur P-GraCE consistant en un nouveau schéma de compression que nous proposons. La figure 5.21 montre les résultats des deux méthodes de compression en choisissant les valeurs optimales de leurs paramètres.

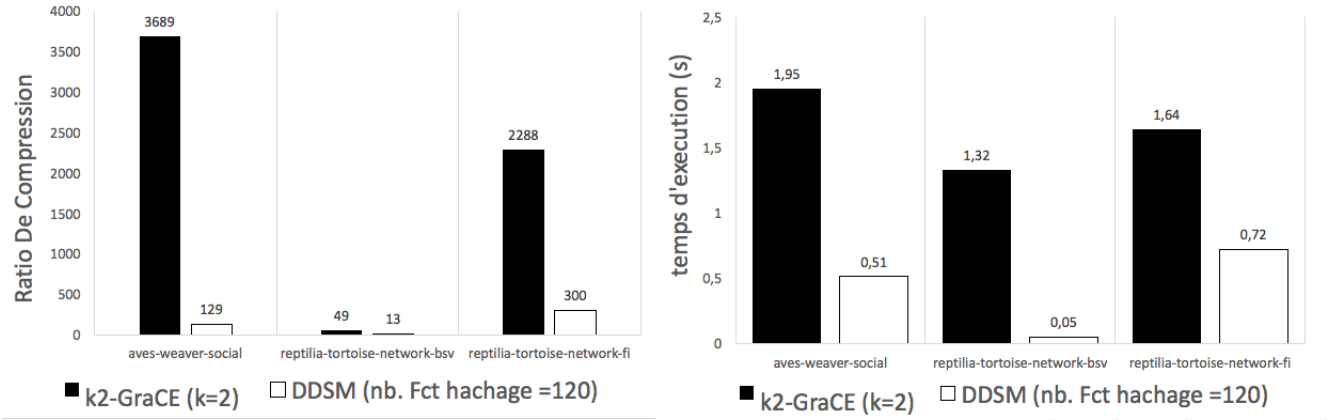


FIGURE 5.21 – Comparaison entre les méthodes destinées aux graphes dynamiques.

Nous remarquons que la ratio de compression est meilleur dans le moteur $k2$-GraCE. En effet, la méthode DDSM donne un ratio de compression très faible en le comparant avec le moteur k^2-GraCE (exemple : 3689 vs 129 pour le graphe aves-social-network). Cependant, nous observons bien que le temps d'exécution est nettement meilleur dans le cas de notre méthode. Nous rappelons aussi que notre méthode permet d'extraire les sous-graphes et les communautés les plus denses (les plus importantes) ainsi que leur comportement temporel avec un accès direct alors que le moteur k^2-GraCE nécessite un traitement supplémentaire qui peut être lourd dans le cas des grands graphes.

5.7 Synthèse des tests

A travers ce chapitre, nous avons pu évaluer nos deux moteurs tout en étudiant l'effet de différentes configurations sur leur performance. Des datasets de domaines hétérogènes ont été employés durant cette étape afin de déduire l'influence de ces derniers sur le choix de l'algorithme de compression. Nous avons opté pour trois métriques d'évaluation : ratio de compression, le nombre de bits par nœuds et le temps d'exécution de l'algorithme de compression.

Dans le cas du moteur k^2-GraCE, nous avons constaté que l'utilisation de la matrice d'adjacence pour la construction de l'arbre est trop couteuse en temps d'exécution. L'utilisation de

la structure proposée dans la bibliothèque SNAP donne des résultats nettement meilleurs pour cette métrique. Nous avons aussi observé que le choix du paramètre k influe fortement sur les performances de la compression. Selon les résultats obtenus, la valeur optimale dans le cas statique est bien k=2. De plus, nous avons constaté que l'utilisation d'un ré-ordonnancement des nœuds n'est intéressante que dans le cas des graphes autres que les graphes du web où nous avons obtenu une amélioration du ratio de compression d'un facteur de 1,4 dans le cas du graphe wiki-Vote. La généralisation proposée pour le cas des graphes statiques non orientés permet de doubler le ratio de compression améliorant ainsi la qualité de la compression. D'autre part, nous avons remarqué que l'utilisation du graphe original dans le cas dynamique donne de meilleurs résultats. En effet, nous avons constaté que l'utilisation de la matrice de différence n'est bénéfique que dans le cas où les captures possèdent un degré élevé de ressemblance des liens, ce qui n'est généralement pas le cas dans les graphes réels.

Le deuxième moteur, P-GraCE, permet de compresser le graphe en entrée avec ses sous-structures les plus importantes. Les tests ont montré qu'il n'est pas intéressant d'utiliser le troisième type de motifs dans les méthodes basées sur les propriétés de la matrice d'adjacence car il est peu fréquent dans les graphes réels. Pour le beam-search, nous avons bien remarqué que le niveau d'agrégation doit être choisi d'une manière qui assure un compromis entre le temps d'exécution et le ratio de compression. L'évaluation des différentes phases de l'approche basée sur les méthodes de clustering a montré que les paramètres donnant les meilleurs performances consiste en :(1) l'utilisation de la méthode *Step* comme méthode de sélection, (2) l'utilisation des trois méthodes de clustering ensembles pour l'extraction de motifs. Finalement, l'évaluation de la méthode que nous proposons a montré qu'elle donne des résultats compétitifs en terme de temps de compression. Certes, le moteur k2-GraCE offre un meilleur ratio dans le cas dynamique mais il n'offre pas les mêmes avantages que la méthode que nous proposons. En effet, le résultat obtenu par DDSM permet une analyse très rapide des communautés et leurs comportements temporels.

La comparaison entre les deux moteurs nous a amené à déduire que l'utilisation du moteur k^2-GraCE est meilleur dans le cas où le but est de diminuer la taille du graphe uniquement. Cependant, le moteur P-GraCE permet d'avoir des informations interprétables qui aident dans l'analyse du graphe car il permet de synthétiser l'information incluse dans le graphe initiale sous forme des sous-structures les plus pertinentes.

Conclusion générale

De nos jours, les graphes sont omniprésents. Cependant, leur taille présente un obstacle presque insurmontable à la compréhension du caractère essentiel des données. D'où la nécessité de la compression qui permet de réduire la taille des graphes tout en gardant le caractère utile de l'information incluse dans ces derniers.

Dans notre travail, nous nous sommes focalisées sur les méthodes de compression basées sur l'extraction de motifs et celles basées sur les arbres k^2-trees. Dans un premier temps, nous avons étudié les différentes méthodes et techniques s'inscrivant dans les deux classes ainsi que leurs différentes classifications existantes dans la littérature. Cette étude nous a initié au domaine de compression et nous a permis de mieux comprendre l'approche théorique de chaque classe. Suite à cela, nous avons proposé une nouvelle classification consistant en une rectification d'une ancienne classification proposée dans un Master précédent (W. GUERMAH, 2018). Nous nous sommes basées pour cela sur le principe de fonctionnement et les fondements théoriques de ces méthodes. Nous avons conclu cette partie en établissant une étude comparative au sein d'une même classe et entre les deux classes en considérant différents critères : type de graphe en entrée, type de compression, ...

La deuxième partie de notre travail consiste en l'implémentation d'une méthode de chaque classe que nous avons proposées dans la première étape dans le but de les réunir dans une même plateforme et d'établir une comparaison plus objective entre ces techniques. De ce fait, nous avons proposé deux moteurs de compression de graphes (un moteur par classe). Le premier moteur k^2-GraCE exploite les propriétés de la matrice d'adjacence pour compresser le graphe. Il supporte trois types de graphe : graphes statiques orientés, graphes statiques non orientés et les graphes dynamiques. Nous l'avons doté d'une phase de pré-traitement qui permet d'exploiter davantage les caractéristiques des graphes. Le deuxième moteur P-GraCE quant à lui supporte trois types de graphe : graphes orientés statiques, graphes non orientés statiques et finalement les graphes étiquetés. Il englobe quatre approches de compression : une approche basée sur le beam search, une approche basée sur les méthodes de clustering, une approche basée sur le MinHashing et une approche basée sur l'extraction de motifs à partir de la matrice d'adjacence. Nous avons aussi proposé une nouvelle méthode de compression s'intitulant DDSM destinée pour les graphes dynamiques et permettant de fournir une compression interprétable du graphe initial.

La dernière étape de notre projet représente une étape d'évaluation des deux moteurs de compression ainsi que la méthode que nous avons proposée. Nous avons utilisé pour cela des graphes réels issus de domaines différents ainsi que trois métriques d'évaluation : le ratio de compression, le nombre de bits par nœud et le temps d'exécution. Dans le cas du moteur k^2-GraCE, nous avons montré que l'utilisation de la matrice d'adjacence dégrade les performances

en terme de temps d'exécution et que l'utilisation de la liste d'adjacence ou la structure de la bibliothèque SNAP donne des résultats meilleurs. Pour le module de pré-traitement de ce moteur, nous avons constaté que l'utilisation d'algorithme de ré-ordonnancement ne donne une amélioration que dans le cas des graphes qui ne sont pas issus du domaine du web. Nous avons aussi obtenu une amélioration d'un facteur de 2 du ratio de compression lorsque la partie supérieure uniquement de la matrice est considérée. Le meilleur ratio obtenu pour ce moteur est de 5474 dans le cas du graphe Retilia–toroise-network-bsv qui est un graphe dynamique. Pour le deuxième moteur P-GracE, les tests ont montré que, dans le cas de la méthode basée sur les propriétés de la matrice d'adjacence, les motifs de la classe (01) ou de la classe (03) (i.e les motifs de ayant la forme suivante : 10...010...0) sont plus intéressants. Nous avons aussi constaté que l'approche basée sur les méthodes de clustering produit une meilleur compression lors de l'utilisation des trois méthodes de clustering ensemble comme méthode d'extraction de motifs et la méthode *Step* comme méthode de sélection. Concernant la méthode que nous avons proposée, nous avons obtenu un ratio de compression qui se stabilise à partir de 50 permutations avec un temps de compression compétitif. La comparaison entre les deux moteurs a montré que le moteur k^2-GraCE donne des résultats nettement meilleurs que le moteur P-GraCE en terme de taux de compression. Cependant, le moteur P-GraCE offre une compression dans un temps meilleur tout en produisant une compression qui facilite l'analyse et la compréhension du graphe initial à travers ses sous-structures les plus importantes ayant généralement une interprétation selon le domaine d'application.

Comme perspective à ce travail, nous aimerions bien tester nos deux moteurs avec des graphes encore plus larges pour pouvoir les évaluer davantage. Nous voudrons aussi tester notre méthode contre les méthodes existantes dans la littérature, notamment TimeCrunch (Shah et al., 2015), sur les mêmes graphes afin de mieux comparer et situer notre contribution dans la littérature.

Annexe A

Extraction de voisins dans un arbre k^2-tree

La compression par les arbres k^2-trees a connu un large succès dans la communauté scientifique. L'une des causes les plus importantes qui ont favorisé cela est qu'elle permet d'extraire le voisinage des nœuds sans reconstitution du graphe initiale. Cette partie se veut une explication des deux algorithmes d'extraction de voisins directes et inverses.

Comme nous l'avons déjà expliqué dans la section 3.2 de la partie conception du moteur k^2-GraCE, les arbres k^2-tree sont représentés en mémoire à l'aide de deux chaines binaires, T et L, qui regroupe le contenu de l'arbre de haut vers le bas et de la gauche vers la droite. Nous fournissons ci-après les algorithmes d'extraction de voisinage (Brisaboa et al., 2009).

Algorithme 11 : Direct

Entrée :

- n : la taille de la sous-matrice
- p : l'indice de la ligne
- q : l'indice de la colonne
- x : l'indice dans l'arbre

Sortie : affichage des voisins Directes

```
1: si x ⩾ |T| alors
2:    si L[x-|T|] =1 alors
3:       afficher q
4: sinon
5:    si x = -1 ou T[x]=1 alors
6:       y = rank(T,x)*k^2 + p/(n/k)
7:       pour j =0 ... k-1 faire
8:          Direct(n/k,p mod(n/k),q+
            j*(n/k),y+j)
```

Algorithme 12 : Inverse

Entrée :

- n : la taille de la sous-matrice
- q : l'indice de la colonne
- p : l'indice de la ligne
- x : l'indice dans l'arbre

Sortie : affichage des voisins Inverses

```
1: si x ⩾ |T| alors
2:    si L[x-|T|] =1 alors
3:       afficher p
4: sinon
5:    si x = -1 ou T[x]=1 alors
6:       y = rank(T,x)*k^2 + q/(n/k)
7:       pour j =0 ... k-1 faire
8:          Inverse(n/k,q mod(n/k),p+
            j*(n/k),y+j*k)
```

Annexe B

Description des graphes de test

Graphe de test	Caractéristiques	
	Nombre de nœuds	Nombre de liens
eu-2005	862 milles de nœuds	19M de liens
Cond-mat	36 milles de nœuds	171 milles de liens
CommNet	taille : 225.9MB	
SalesDay	1 milles de nœuds	
Movielens10M	10 milles de nœuds	10M de liens
ASOregon	13 milles nœuds	37 milles liens
Enron	80 milles nœuds	288 milles liens
uk-2002	18 milles nœuds	298 milles liens
graphe test de GCUPMT	8 192 milles nœuds	
Composante chimique	21 étiquettes	422 transactions
NotreDame	325 milles de nœuds	1M de liens

TABLE B.1 – Graphes de test

Annexe C

Algorithmes des méthodes de clustering

Algorithme 13 : SlashBurn

Entrée :

- G(V,E) : Graphe non orienté
- n : nombre de nœuds dans le graphe

Sortie : ensemble de structures

```
1:
2: D = [] //tableau de degré
3: GCC : un graph
4: pour  i=0 .. n-1 faire
5:     D[i] = Degré(i)
6: hub = { i | max(D[i])}
7: spokes = { i | min(D[i])} //nœuds isolés
8: GCC = G(V) - {e_hub} - {e_spokes}
9: SlashBurn(GCC, Taille du GCC)
```

Algorithme 14 : KCBC

Entrée :

- G : Graphe non orienté
- n : nombre de nœuds dans le graphe

Sortie : ensemble de structures

1: Degré = []
2: F = G
3: stop = VRAI
4: structure = { }
5: //étape 1 : Calculer le degré de chaque nœud
6: **pour** i =0 ... n-1 **faire**
7: Degré [i]= $\mathbf{Deg}_G$(i)
8: k_{max} = k
9: **tantque** k_{max} > 1 and and stop = VRAI **faire**
10: //étape 2 : Décomposition du graphe avec k_{max}
11: F = G
12: **pour** x =0 ... n-1 **faire**
13: **si** $\mathbf{DEG}_F$(x) < k_{max} **alors**
14: $\mathbf{Supprimer}_F$(x)
15: **si** F = **alors**
16: stop = FAUX
17: **sinon**
18: k_{max} = k_{max} - 1
19: //étape 3 : Identification des structures
20: structure = structure ∪ { composants connexes ∈ F }
21: //étape 4 : Suppression des arêtes entre structures
22: **pour** $struct_i$, $struct_j$ ∈ structure **faire**
23: $\mathbf{supprimerArête}_F(struct_i \cap struct_j)$

Algorithme 15 : Louvain

Entrée :

- G(V,E) : Graphe non orienté pondéré
- n : nombre de nœuds dans le graphe

Sortie : ensemble de structures

1: //Phase 1
2: $C = \{\{v_i\} \mid v_i \in G(V)\}$
3: **répéter**
4: **pour** $v_i \in C_x$ and $v_j \in C_y$ and $(v_i, v_j) \in G(E)$ **faire**
5: H = $\{ \Delta Q_i(C_x, C_y) \mid C_x, C_y \in C \}$
6: **si** $max\Delta Q_i(C_x, C_y) > 0$ **alors**
7: $C_y = C_y \cup \{v_i\}$
8: $C_x = C_x$ - $\{v_i\}$
9: //Phase 2
10: V' = $\{ C_x \mid C_x \in C \}$
11: E' = $\{ \sum \frac{(v_i,v_j)}{1} \mid v_i \in C_x \mid v_j \in C_y \mid e_{i,j} \in G(E) \}$
12: G = G'(V',E')
13: **jusqu'à** aucune amélioration possible
14: ensemble de structures = $\{ v_i \mid v_i \in G \}$

Bibliographie

(2011). Graph based knowledge discovery. In *Subdue*.

(2012). *Quelques rappels sur la théorie des graphes*. IUT Lyon Informatique.

(April 2017). Caida network dataset. In *KONECT*.

Alvarez-Garcia, S., de Bernardo, G., Brisaboa, N. R., and Navarro, G. (2017). A succinct data structure for self-indexing ternary relations. *Journal of Discrete Algorithms*, 43 :38–53.

Álvarez-García, S., Freire, B., Ladra, S., and Pedreira, Ó. (2018). Compact and efficient representation of general graph databases. *Knowledge and Information Systems*, pages 1–32.

Aragon, C. R. and Seidel, R. G. (1989). Randomized search trees. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 540–545. IEEE.

Asano, Y., Miyawaki, Y., and Nishizeki, T. (2008). Efficient compression of web graphs. In *International Computing and Combinatorics Conference*, pages 1–11. Springer.

Badr, M. (2013). *Traitement de requêtes top-k multicritères et application à la recherche par le contenu dans les bases de données multimédia*. PhD thesis, Cergy-Pontoise.

Barbay, J., Golynski, A., Munro, J. I., and Rao, S. S. (2006). Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Annual Symposium on Combinatorial Pattern Matching*, pages 24–35. Springer.

Boldi, P., Rosa, M., Santini, M., and Vigna, S. (2011). Layered label propagation : A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*, pages 587–596. ACM.

Boldi, P. and Vigna, S. (2004). The webgraph framework i : compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM.

Brisaboa, N. R., de Bernardo, G., Gutiérrez, G., Ladra, S., Penabad, M. R., and Troncoso, B. A. (2015). Efficient set operations over k2-trees. In *Data Compression Conference (DCC), 2015*, pages 373–382. IEEE.

Brisaboa, N. R., De Bernardo, G., Konow, R., and Navarro, G. (2014a). K 2-treaps : Range top-k queries in compact space. In *International Symposium on String Processing and Information Retrieval*, pages 215–226. Springer.

Brisaboa, N. R., De Bernardo, G., and Navarro, G. (2012). Compressed dynamic binary relations. In *Data Compression Conference (DCC), 2012*, pages 52–61. IEEE.

Brisaboa, N. R., Ladra, S., and Navarro, G. (2009). k 2-trees for compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 18–30. Springer.

Brisaboa, N. R., Ladra, S., and Navarro, G. (2013). Dacs : Bringing direct access to variable-length codes. *Information Processing & Management*, 49(1) :392–404.

Brisaboa, N. R., Ladra, S., and Navarro, G. (2014b). Compact representation of web graphs with extended functionality. *Information Systems*, 39 :152–174.

Buehrer, G. and Chellapilla, K. (2008). A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106. ACM.

Claude, F. and Ladra, S. (2011). Practical representations for web and social graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1185–1190. ACM.

Claude, F. and Navarro, G. (2010a). Extended compact web graph representations. In *Algorithms and Applications*, pages 77–91. Springer.

Claude, F. and Navarro, G. (2010b). Fast and compact web graph representations. *ACM Transactions on the Web (TWEB)*, 4(4) :16.

De Berg, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O. (1997). Computational geometry. In *Computational geometry*, pages 1–17. Springer.

De Bernardo, G., Álvarez-García, S., Brisaboa, N. R., Navarro, G., and Pedreira, O. (2013). Compact querieable representations of raster data. In *International Symposium on String Processing and Information Retrieval*, pages 96–108. Springer.

de Bernardo Roca, G. (2014). *New data structures and algorithms for the efficient management of large spatial datasets*. PhD thesis, Citeseer.

Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., and Shalita, A. (2016). Compressing graphs and indexes with recursive graph bisection. *arXiv preprint arXiv :1602.08820*.

Fages, J.-G. (2014). *Exploitation de structures de graphe en programmation par contraintes*. PhD thesis, Ecole des Mines de Nantes.

Garcia, S. A., Brisaboa, N. R., de Bernardo, G., and Navarro, G. (2014). Interleaved k2-tree : Indexing and navigating ternary relations. In *Data Compression Conference (DCC), 2014*, pages 342–351. IEEE.

Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics.

Guillaume, J.-L. and Latapy, M. (2002). The web graph : an overview. In *Actes d'ALGOTEL'02 (Quatrièmes Rencontres Francophones sur les aspects Algorithmiques des Télécommunications)*.

Hennecart, F., Bretto, A., and Faisant, A. (2012). Eléments de théorie des graphes.

Hernández, C. and Navarro, G. (2014). Compressed representations for web and social graphs. *Knowledge and information systems*, 40(2) :279–313.

Jean-Charles Régin, A. M. (2016). Théorie des graphes. Technical report.

Ketkar, N. S., Holder, L. B., and Cook, D. J. (2005). Subdue : Compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st international workshop on open source data mining : frequent pattern mining implementations*, pages 71–76. ACM.

Khan, K. U., Nawaz, W., and Lee, Y.-K. (2014). Set-based unified approach for attributed graph summarization. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, pages 378–385. IEEE.

Koutra, D., Kang, U., Vreeken, J., and Faloutsos, C. (2015). Summarizing and understanding large graphs. *Statistical Analysis and Data Mining : The ASA Data Science Journal*, 8(3) :183–202.

LeFevre, K. and Terzi, E. (2010). Grass : Graph structure summarization. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 454–465. SIAM.

Lehman, E., Leighton, F. T., and Meyer, A. R. (2010). Mathematics for computer science. Technical report, Technical report, 2006. Lecture notes.

Lelewer, D. A. and Hirschberg, D. S. (1987). Data compression. *ACM Computing Surveys (CSUR)*, 19(3) :261–296.

Lemmouchi, S. (2012). *Etude de la robustesse des graphes sociaux émergents*. PhD thesis, Université Claude Bernard-Lyon I.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets : Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Liu, Y., Safavi, T., Dighe, A., and Koutra, D. (2018a). Graph summarization methods and applications : A survey. *ACM Computing Surveys (CSUR)*, 51(3) :62.

Liu, Y., Safavi, T., Shah, N., and Koutra, D. (2018b). Reducing large graphs to small supergraphs : a unified approach. *Social Network Analysis and Mining*, 8(1) :17.

Liu, Y., Shah, N., and Koutra, D. (2015). An empirical comparison of the summarization power of graph clustering methods. *arXiv preprint arXiv :1511.06820*.

Lopez, P. (2003). Cours de graphes.

Maneth, S. and Peternek, F. (2015). A survey on methods and systems for graph compression. *arXiv preprint arXiv :1504.00616*.

Maneth, S. and Peternek, F. (2018). Grammar-based graph compression. *Information Systems*, 76 :19–45.

Martínez-Prieto, M. A., Fernández, J. D., and Cánovas, R. (2012). Compression of rdf dictionaries. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 340–347. ACM.

Maserrat, H. and Pei, J. (2012). Community preserving lossy compression of social networks. In *2012 IEEE 12th International Conference on Data Mining*, pages 509–518. IEEE.

Müller, D. (2012). *Introduction à la théorie des graphes*. Commission romande de mathématique (CRM).

Parlebas, P. (1972). Centralité et compacité d'un graphe. *Mathematiques et sciences humaines*, 39 :5–26.

Pellegrini, M., Haynor, D., and Johnson, J. M. (2004). Protein interaction networks. *Expert review of proteomics*, 1(2) :239–249.

Pieterse, V., Kourie, D. G., Cleophas, L., and Watson, B. W. (2010). Performance of c++ bit-vector implementations. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 242–250. ACM.

Rigo, M. (2010). *Théorie des graphesorie des graphes*. Université de liège, Faculté des sciences Département de mathématiques.

Rossi, R. A. and Ahmed, N. K. (2015a). The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Rossi, R. A. and Ahmed, N. K. (2015b). The network data repository with interactive graph analytics and visualization. In *AAAI*.

Rossi, R. A. and Zhou, R. (2018). Graphzip : a clique-based sparse graph compression method. *Journal of Big Data*, 5(1) :10.

Roux, P. (2014). *Théorie des graphes*.

SABLIK, M. (2018). Graphe et langage.

Sethi, G., Shaw, S., Vinutha, K., and Chakravorty, C. (2014). Data compression techniques. *International Journal of Computer Science and Information Technologies*, 5(4) :5584–6.

Shah, N., Koutra, D., Zou, T., Gallagher, B., and Faloutsos, C. (2015). Timecrunch : Interpretable dynamic graph summarization. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1055–1064. ACM.

Shah, R. J. (2018). Graph compression using pattern matching techniques. *arXiv preprint arXiv :1806.01504*.

Shen, Z., Ma, K.-L., and Eliassi-Rad, T. (2006). Visual analysis of large heterogeneous social networks by semantic and structural abstraction. *IEEE transactions on visualization and computer graphics*, 12(6) :1427–1439.

Shi, L., Tong, H., Tang, J., and Lin, C. (2015). Vegas : Visual influence graph summarization on citation networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(12) :3417–3431.

Shi, Q., Xiao, Y., Bessis, N., Lu, Y., Chen, Y., and Hill, R. (2012). Optimizing k2 trees : A case for validating the maturity of network of practices. *Computers & Mathematics with Applications*, 63(2) :427–436.

Uthayakumar, J., Vengattaraman, T., and Dhavachelvan, P. (2018). A survey on data compression techniques : From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences*.

W. GUERMAH, T. B. (2018). Compression de graphes : étude et classification. Master's thesis, Esi.

Yıldırım, H., Chaoji, V., and Zaki, M. J. (2012). Grail : a scalable index for reachability queries in very large graphs. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(4) :509–534.

Zhang, H., Duan, Y., Yuan, X., and Zhang, Y. (2014a). Assg : Adaptive structural summary for rdf graph data. In *International Semantic Web Conference (Posters & Demos)*, pages 233–236. Citeseer.

Zhang, Y., Xiong, G., Liu, Y., Liu, M., Liu, P., and Guo, L. (2014b). Delta-k 2-tree for compact representation of web graphs. In *Asia-Pacific Web Conference*, pages 270–281. Springer.