Subgraph Isomorphism Search In Massive Graph Data

Chems Eddine Nabti

To cite this version:

Chems Eddine Nabti. Subgraph Isomorphism Search In Massive Graph Data. Databases [cs.DB]. Université de Lyon, 2017. English. NNT : 2017LYSE1293 . tel-01781831

HAL Id: tel-01781831

https://theses.hal.science/tel-01781831

Submitted on 30 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre NNT : xxx

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1

École Doctorale ED512
Ecole Doctorale Informatique et Mathématiques

Spécialité de doctorat :
Discipline : Informatique

Soutenue publiquement le , par :
Chems eddine NABTI

Subgraph Isomorphism Search In Massive Graph Data

Devant le jury composé de :

LIETARD Ludovic, Maitre de Conférences, HDR, Université de Rennes1 Rapporteur
TAMINE-LECHANI Lynda, Professeure des Universités, UPS Toulouse Rapporteur(e)
TERMIER Alexandre, Professeur des Universités, Université de Rennes1 Examinateur

SEBA Hamida, Maitre de conférences, HDR, Université Claude Bernard Lyon1 Directrice de thèse

UNIVERSITE CLAUDE BERNARD - LYON 1

Président de l'Université	**M. le Professeur Frédéric FLEURY**
Président du Conseil Académique	M. le Professeur Hamda BEN HADID
Vice-président du Conseil d'Administration	M. le Professeur Didier REVEL
Vice-président du Conseil Formation et Vie Universitaire	M. le Professeur Philippe CHEVALIER
Vice-président de la Commission Recherche	M. Fabrice VALLÉE
Directrice Générale des Services	Mme Dominique MARCHAND

COMPOSANTES SANTE

Faculté de Médecine Lyon Est – Claude Bernard	Directeur : M. le Professeur G.RODE
Faculté de Médecine et de Maïeutique Lyon Sud – Charles Mérieux	Directeur : Mme la Professeure C. BURILLON
Faculté d'Odontologie	Directeur : M. le Professeur D. BOURGEOIS
Institut des Sciences Pharmaceutiques et Biologiques	Directeur : Mme la Professeure C. VINCIGUERRA
Institut des Sciences et Techniques de la Réadaptation	Directeur : M. X. PERROT
Département de formation et Centre de Recherche en Biologie Humaine	Directeur : Mme la Professeure A-M. SCHOTT

COMPOSANTES ET DEPARTEMENTS DE SCIENCES ET TECHNOLOGIE

Faculté des Sciences et Technologies	Directeur : M. F. DE MARCHI
Département Biologie	Directeur : M. le Professeur F. THEVENARD
Département Chimie Biochimie	Directeur : Mme C. FELIX
Département GEP	Directeur : M. Hassan HAMMOURI
Département Informatique	Directeur : M. le Professeur S. AKKOUCHE
Département Mathématiques	Directeur : M. le Professeur G. TOMANOV
Département Mécanique	Directeur : M. le Professeur H. BEN HADID
Département Physique	Directeur : M. le Professeur J-C PLENET
UFR Sciences et Techniques des Activités Physiques et Sportives	Directeur : M. Y.VANPOULLE
Observatoire des Sciences de l'Univers de Lyon	Directeur : M. B. GUIDERDONI
Polytech Lyon	Directeur : M. le Professeur E.PERRIN
Ecole Supérieure de Chimie Physique Electronique	Directeur : M. G. PIGNAULT
Institut Universitaire de Technologie de Lyon 1	Directeur : M. le Professeur C. VITON
Ecole Supérieure du Professorat et de l'Education	Directeur : M. le Professeur A. MOUGNIOTTE
Institut de Science Financière et d'Assurances	Directeur : M. N. LEBOISNE

Abstract

Querying graph data is a fundamental problem that witnesses an increasing interest especially for massive structured data where graphs come as a promising alternative to relational databases for big data modeling. However, querying graph data is different and more complex than querying relational table-based data. The main task involved in querying graph data is subgraph isomorphism search which is an NP-complete problem. Subgraph isomorphism search is an important problem which is involved in various domains such as pattern recognition, social network analysis, biology, etc. It consists to enumerate the subgraphs of a data graph that match a query graph. The most known solutions of this problem are backtracking-based. They explore a large search space which results in a high computational cost when we deal with massive graph data.

To reduce time and memory space complexity of subgraph isomorphism search. We propose to use compressed graphs. In our approach, subgraph isomorphism search is achieved on compressed representations of graphs without decompressing them. Graph compression is performed by grouping vertices into super vertices. This concept is known, in graph theory, as modular decomposition. It is used to generate a tree representation of a graph that highlights groups of vertices that have the same neighbors. With this compression we obtain a substantial reduction of the search space and consequently a significant saving in the processing time.

We also propose a novel encoding of vertices that simplifies the filtering of the search space. This new mechanism is called compact neighborhood Index (CNI). A CNI distills all the information around a vertex in a single integer. This simple neighborhood encoding reduces the time complexity of vertex filtering from cubic to quadratic which is considerable for big graphs. We propose also an iterative global filtering algorithm that relies on the characteristics of CNIs to ensure a global pruning of the search space.

We evaluated our approaches on several real-word datasets and compared them with the state of the art algorithms.

Résumé

L'interrogation de graphes de données est un problème fondamental qui connait un grand intérêt, en particulier pour les données structurées massives où les graphes constituent une alternative prometteuse aux bases de données relationnelles pour la modélisation des grandes masses de données. Cependant, l'interrogation des graphes de données est différente et plus complexe que l'interrogation des données relationnelles à base de tables. La tâche principale impliquée dans l'interrogation de graphes de données est la recherche d'isomorphisme de sous-graphes qui est un problème NP-complet.

La recherche d'isomorphisme de sous-graphes est un problème très important impliqué dans divers domaines comme la reconnaissance de formes, l'analyse des réseaux sociaux, la biologie, etc. Il consiste à énumérer les sous-graphes d'un graphe de données qui correspondent à un graphe requête. Les solutions les plus connues de ce problème sont basées sur le retour arrière (backtracking). Elles explorent un grand espace de recherche, ce qui entraîne un coût de traitement élevé, notamment dans le cas de données massives.

Pour réduire le temps et la complexité en espace mémoire dans la recherche d'isomorphisme de sous-graphes, nous proposons d'utiliser des graphes compressés. Dans notre approche, la recherche d'isomorphisme de sous-graphes est réalisée sur une représentation compressée des graphes sans les décompresser. La compression des graphes s'effectue en regroupant les sommets en super-

sommets. Ce concept est connu dans la théorie des graphes par la décomposition modulaire. Il sert à générer une représentation en arbre d'un graphe qui met en évidence des groupes de sommets qui ont les mêmes voisins. Avec cette compression, nous obtenons une réduction substantielle de l'espace de recherche et par conséquent, une économie significative dans le temps de traitement.

Nous proposons également une nouvelle représentation des sommets du graphe, qui simplifie le filtrage de l'espace de recherche. Ce nouveau mécanisme appelé "compact neighborhood Index (CNI)" encode l'information de voisinage autour d'un sommet en un seul entier. Cet encodage du voisinage réduit la complexité du temps de filtrage de cubique à quadratique. Ce qui est considérable pour les données massives.

Nous proposons également un algorithme de filtrage itératif qui repose sur les caractéristiques des CNIs pour assurer un élagage global de l'espace de recherche.

Nous avons évalué nos approches sur plusieurs datasets et nous les avons comparées avec les algorithmes de l'état de l'art.

List of Figures

1.1 Hierarchical Model 2
1.2 Network Model [39] 2
1.3 Querying a data graph 3
1.4 graph isomorphism problem [54] 5

2.1 Example of Graphs. 12
2.2 Example of Induced and Partial Subgraphs. 13
2.3 Subgraph isomorphism search. 17
2.4 A partial construction of the search tree. 19
2.5 State of the art Methods. 21
2.6 the NDS distance [55] 26
2.7 Core-Forest-Leaf Decomposition [6] 31
2.8 Example CPI [6] 32
2.9 Running Example [6] 32
2.10 Subgraph isomorphism search on Trinity[47] 34

3.1 Graph compression with [12]. 46
3.2 Compressing steps with modular decomposition. S: series module. P: parallel module. N: neighborhood module [31]. 48
3.3 Example of a graph and its compression [44]. 49
3.4 The architecture for the proposed framework. 50
3.5 Compression Step of the Running Example. 52
3.6 Flowchart of step 1 (Supervertex Selection). 55
3.7 Flowchart of step 2 (Subgraph Search). 55
3.8 Supervertex Selection Phase on our Running Example. 57

3.9 Tree representation of Modules [31]. 58
3.10 Subgraph Search Phase of the Running Example. 59
3.11 AIDS dataset. 66
3.12 NASA dataset. 67
3.13 Human dataset. 67
3.14 Path and Clique Queries. 68
3.15 WebGoogle dataset. 69
3.16 Wiki-talk dataset. 69
3.17 Patent Citation dataset. 70
3.18 LiveJournal dataset. 70
3.19 Pokec dataset. 71
3.20 Orkut dataset. 71

4.1 Running Example. 77
4.2 MND Filter on on the Running Example (pruned of the vertices that do not match query labels). 80
4.3 Needless NLF filtering 81
4.4 NLF filtering with two different vertex parsing orders 83
4.5 CNIs of the Query graph and the Data graph. 90
4.6 Filtering iterations of our running example. 93
4.7 Time performance on small datasets (varying $|V(Q)|$). Results are in logscale. 104
4.8 Time performance on the small dataset DANIO-RERIO (varying $|\Sigma|$ and the label distribution). 105
4.9 Scalability testing (varying $|V(Q)|$). 106
4.10 Scalability testing on large graphs (varying $|V(Q)|$). 107
4.11 Scalability testing (varying $|V(G)|$). 107

List of Tables

3.1 Graph Dataset Characteristics. $avg|V|$: average number of vertices. $avg|E|$: average number of edges. 63
3.2 width=17cm 64

4.1 Notation 85
4.2 Graph Dataset Characteristics. 101

Contents

Abstract **iii**

Résumé **v**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
- 1.1 Thesis Scope 6
- 1.2 Thesis organization 7

2 Subgraph Isomorphism search **9**
- 2.1 Basic Definitions 11
- 2.2 Querying graph data 14
- 2.3 Subgraph isomorphism search over a single large data graph . . 17
- 2.4 Existing Algorithms 20
  - 2.4.1 Ullmann's algorithm 22
  - 2.4.2 VF2 23
  - 2.4.3 SPath and GraphQL 24
  - 2.4.4 GADDI 25
  - 2.4.5 QuickSI 27
  - 2.4.6 Turbo-iso 29
  - 2.4.7 CFL-match 30
  - 2.4.8 Other Methods and techniques 33
- 2.5 Analysis 35

2.6 Conclusion 39

3 SUBGRAPH ISOMORPHISM SEARCH ON COMPRESSED GRAPHS 41

3.1 Introduction 43

3.2 Graph Compression 43

3.3 Compress and Search 51

3.3.1 Candidate Supervertex Selection 53

3.3.2 Subgraph Search 57

3.4 Performance Evaluation 60

3.4.1 Datasets 61

3.4.2 Results 65

3.4.3 Discussion 66

3.5 Conclusion 73

4 Compact Neighborhood Index for Subgraph Queries in Massive Graphs 75

4.1 Motivation 77

4.2 Our approach 84

4.2.1 Compact Neighborhood Index (CNI) 85

4.2.2 Proof of Theorem 1 87

4.3 Proof Sketch of Lemma 3 89

4.3.1 Iterative Local Global Filtering Algorithm (ILGF) 89

4.3.2 Subgraph Search 94

4.3.3 Extension to Larger Graphs 95

4.4 Experiments 97

4.4.1 Datasets 97

4.4.2 Results 101

4.5 $cni(v)$ at $(k > 1)$-hops Neighborhood 103

4.6 Conclusion 108

5 Conclusion and Perspectives 111

5.1 Conclusion 111

5.2 Perspectives 114

List of Publications 117

References **119**

Chapter 1: Introduction

Contents

1.1 Thesis Scope 6
1.2 Thesis organization 7

Graphs are data structures composed of a set of vertices and a set of edges where an edge connects two vertices. A graph is an effective way of formalizing problems and representing objects. They are used to represent complex and heterogeneous linked data in various domains. With graphs, vertices represent objects and edges represent relations between these objects.

Graphs are very flexible, they allow adding new kinds of relationships, new vertices to an existing structure without disturbing existing queries and application functionalities. This is why graphs are widely used in data modeling, especially for massive data. However this is not new, the first data modeling tool is graph-based. The hierarchical data model was the first data modeling tool to be created. First appearing in 1966 as an improvement of general file-processing systems. The main improvement was the possibility to create relationships between information in a data model, which is insured by graphs. The main characteristic of a hierarchical data model is the treelike structure. It consists of a collection of records that are connected to each other through links with a parent-child relationship. Figure 1.1 shows an example of a hierarchical data model.

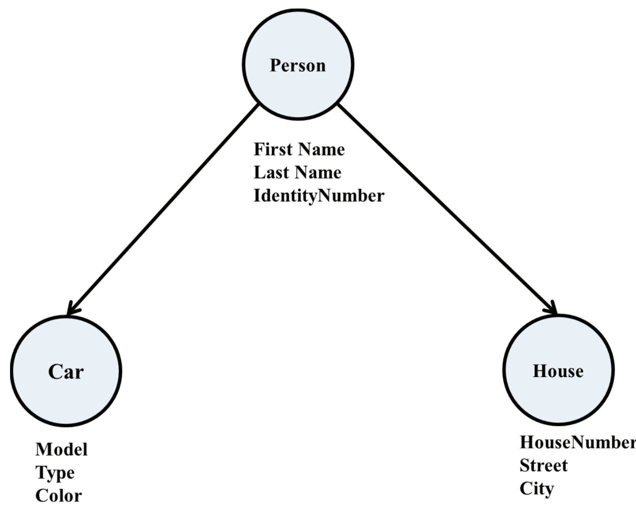


Figure 1.1: Hierarchical Model

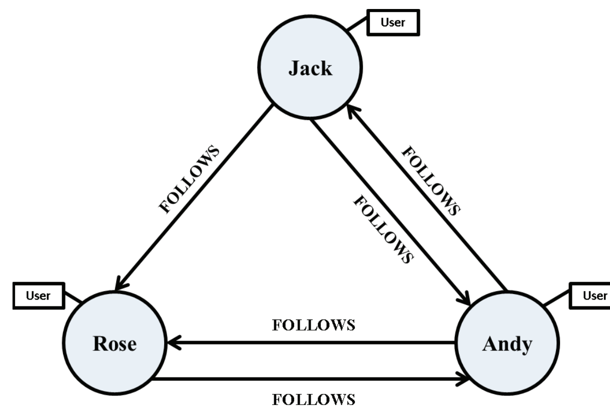


Figure 1.2: Network Model [39]

As an extension of the hierarchical model, the network model was introduced. Originally invented by Charles Bachman in the 70s, it apears as a new data model. It was conceived as a flexible way of representing objects and their relationships. It allows transversal connections, and uses a graph structure. Basically, while the hierarchical database model structures data as a tree of records, with each record having one parent record and many children, the network model allows each record to have multiple parent and child records, forming a generalized

graph structure. Figure 1.2 shows a small example of a network model, that represents a small network of twitter users.

Graph data models knew a peak of interest in the wake of the NoSQL mouvement and were developped more deeply in the era of massive data. Graph characteristics made graph data models more porwerfull than the well appreciated relationnel data model. In a graph data model, vertices represent data, edges represent relations between two data, and both vertices and relations may have special properties: the labels.

Graphs are also used naturally to represent networks, especially social networks: facebook, twitter, etc. They are also used in biology, and chemistry, because molecules and chemical data are graphs by definition.
Using graphs as means of representing data cannot be successful without developing effective ways to search and query this kind of data, especially in nowdays were graphs are large and growing rapidly.
Figure 1.3 shows an example of a graph query problem, in which the idea is to find the occurences of a query graph on a larger data graph.

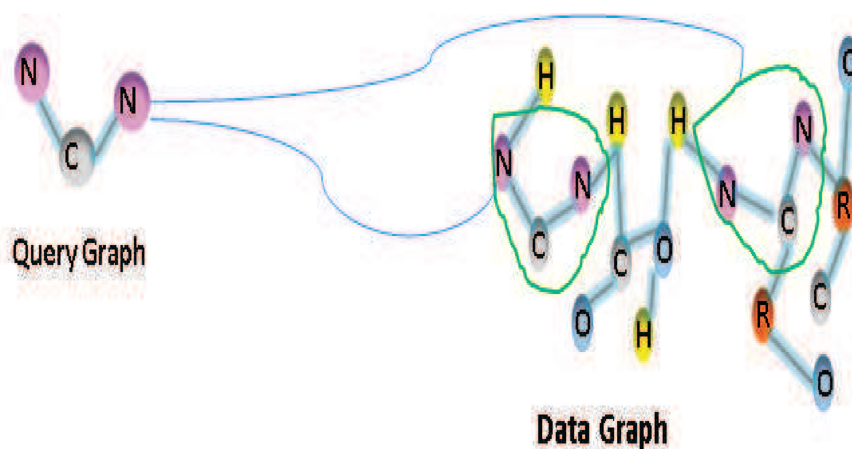


Figure 1.3: Querying a data graph

Querying graph data is a challenging issue. In fact, querying a graph can be processed by searching for subgraphs using the relation of inclusion, or by graph similarity (checking for structural similarity). In the first type of querying (i.e., searching for graphs using the relation of inclusion), we can distinguish two problems: subgraph search and supergraph search.
The supergraph search consists in finding all subgraphs in the data graph for which the query is a supergraph. However, subgraph search consists in finding all graphs in the data graph for which the query is a subgraph. This problem is called subgraph isomorphism search, and is considered as a main task involved in querying data graphs.
The problem of determining isomorphism between two graphs is shown in Figure 1.4. An isomorphism between two graphs G and H is a bijection f, between the vertex sets of G and H. Such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H. First became known during 1960's as a method of comparing two chemical structures, this problem was listed in 1979, by Garey and Johnson, as one of 12 problems belonging to NP, but not known to be either NP-complete or solvable in polynomial time. However, the problem of subgraph isomorphism in which we focus in our thesis is known to be NP-complete [19]. This problem arises in many real word applications related to query processing. The main problem in nowdays data is the amount of information to be processed, and the very large search space in which we query data. To tackle this problem, researchers try to reduce the processing time, the search space, and also to reduce the amount of memory space used to store massive graph data.

Subgraph isomorphism is the problem of finding all embeddings of a query graph into a target graph called data graph. Most existing subgraph isomorphism algorithms are based on a backtracking framework which computes solutions by

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

Figure 1.4: graph isomorphism problem [54]

incrementally matching all query vertices to candidate data vertices. The most known algorithms to subgraph isomorphism search are: Ulmann's algorithm [49], VF2 [41], GADDI [55], Spath [57], QuickSI [45], and GraphQL [25]. These algorithms use different join orders, pruning rules, and auxiliary information to prune-out falsepositive candidates as early as possible. But none of these algorithms is designed to handle all types of graphs with all sizes. For example, QuickSI [45] is designed for handling small graphs, GraphQL [25] and SPath [57] are designed for handling large graphs. Some of these proposed methods, mainly because of the complexity of the subgraph isomorphism problem, show exponential time behavior.

The porposed algorithms are mainly built arround two basic tasks: filter and search (or filtering and verification). The more powerfull the filtering is, the more powerfull is the algorithm that searches for subgraph isomorphism. Filtering is a manner to reduce the search space by eleminating non relevant vertices. However, this filtering can lead to high cost, so it must be efficient without being time consuming. Since reducing the search space is very important, a simplified representation of graphs will be very usefull. This is the main focus of our work.

1.1 Thesis Scope

In this thesis, we focus on simplifying graph representations to ensure better performance with subgraph isomorphism search, especially in the case of massive data. We worked on two main approaches:

- A compression of the whole graph: in this case our data graph is summarized and we achieve subgraph isomorphism search on the compressed version of the graph. The main idea in graph summarizing approaches it to find a short representation of the input graph, in the form of a compressed graph. Summarizing a graph can be very usefull:

  a. It reduces the storage space. Depending on the size of the graph, a graph that does not fit in memory before summarizing will be loaded after compression.

  b. Because graphs became smaller, they can be queried and analyzed faster and more easily.

  c. Noise filter. Summarizing graphs help eliminating non important information. It also highlights only important ones.

  Graph summarizing have also many challenges. The main challenge is to process all the amount of data contained in large graphs, without losing any information, or important graph structures. In our first approach, we use modular decomposition as a mean of compression. Modular decomposition of graphs consists to highlight set of vertices that have the same neighbors and so are not distinguishable from outside. These sets of vertices are called modules. We perform subgraph isomorphism search on compressed graphs without decompressing them.

- A compression of the neighborhood of a vertex: In this case, we propose a new manner to encode vertex's neighborhood in order to simplify filtering. In our approach, all the information arround a vertex is compacted in a single integer leading to a simple but effective filtering scheme for processing subgraph isomorphism search. Filtering is the main task that reduces the search space during subgraph isomorphism search.

In the two cases, the aim is to be able to deal with massive data.

1.2 Thesis organization

The remaining of this thesis contains three chapters:

Chapter 2 'Subgraph Isomorphism search' describes the state of the art, discusses the problem of subgraph isormorphism search, and shows how existing algorithms handle this problem.

Chapter 3 'Subgraph isomorphism search on compressed graphs' is dedicated to our first contribution in which we propose a new method to subgraph search in compressed graphs, without decompressing them. The proposed approach is evaluated on nine datasets, and is compared with other algorithms from the literature.

Chapter 4 'Compact Neighborhood Index for Subgraph Queries in Massive Graphs' represents our second contribution, in which a vertex's neighborhood compression is processed. This compression allows a constant time pruning mechanism, by using topological information. The proposed algorithm is also compared with other methods from the literature.

Finally, we conclude the manuscript by summarizing the major contributions of this thesis and proposing research directions for future work.

Chapter 2: Subgraph Isomorphism search

In this chapter, we present the subgraph isomorphism search problem which is one of the most interresting problems in graph theory. We also survey and discuss the algorithms proposed to solve this problem.

Contents

2.1 Basic Definitions **11**

2.2 Querying graph data **14**

2.3 Subgraph isomorphism search over a single large data graph **17**

2.4 Existing Algorithms **20**

2.4.1 Ullmann's algorithm 22

2.4.2 VF2 23

2.4.3 SPath and GraphQL 24

2.4.4 GADDI 25

2.4.5 QuickSI 27

2.4.6 Turbo-iso 29

2.4.7 CFL-match 30

2.4.8 Other Methods and techniques 33

2.5 Analysis **35**

2.6 Conclusion **39**

2.1 Basic Definitions

A graph is a mathemathical concept defined as follows:

Definition 1. *A graph G is a 3-tuple $G = (V(G), E(G), \ell, \Sigma)$, where $V(G)$ is a set of vertices (also called nodes), $E(G) \subseteq V(G) \times V(G)$ is a set of edges connecting the vertices, $\ell : V(G) \cup E(G) \to \Sigma$ is a labeling function on the vertices and the edges where Σ is the set of labels that can appear on the vertices and/or the edges.*

when a complex object is modeled by a graph, vertices and edges represent respectively its entities and the relations between these entities. These entities can be labeled by assigning one or more values (symbolic or numeric) to each vertex and/or edge. In this case, the graph is called a labeled graph. If no labels are used, the graph is then called a non-labeled graph. This type of graph is generally used when only its structure or shape is important and not the labels of vertices and edges.
However, any non-labeled graph can be represented by a labeled graph by associating the same vertex label and the same edge label to all vertices and all edges, respectively. If all graph edges are symbolized by arrows, the graph is then called oriented graph (labeled or not). For example web services are represented by labeled oriented graphs. Figure 2.1 shows some graph examples.

An undirected edge between vertices u and v is denoted indifferently by (u, v) or (v, u). For each $v \in V(G)$, $d(v)$ denotes the degree of v, i.e., the number of neighbors of v, where a neighbor is a vertex adjacent to v. The label or set of labels of a vertex v is given by $\ell(v)$.
The notation $G = (V, E)$, with ℓ omitted means that we actually do not need the labels of the vertices but just their identifiers.

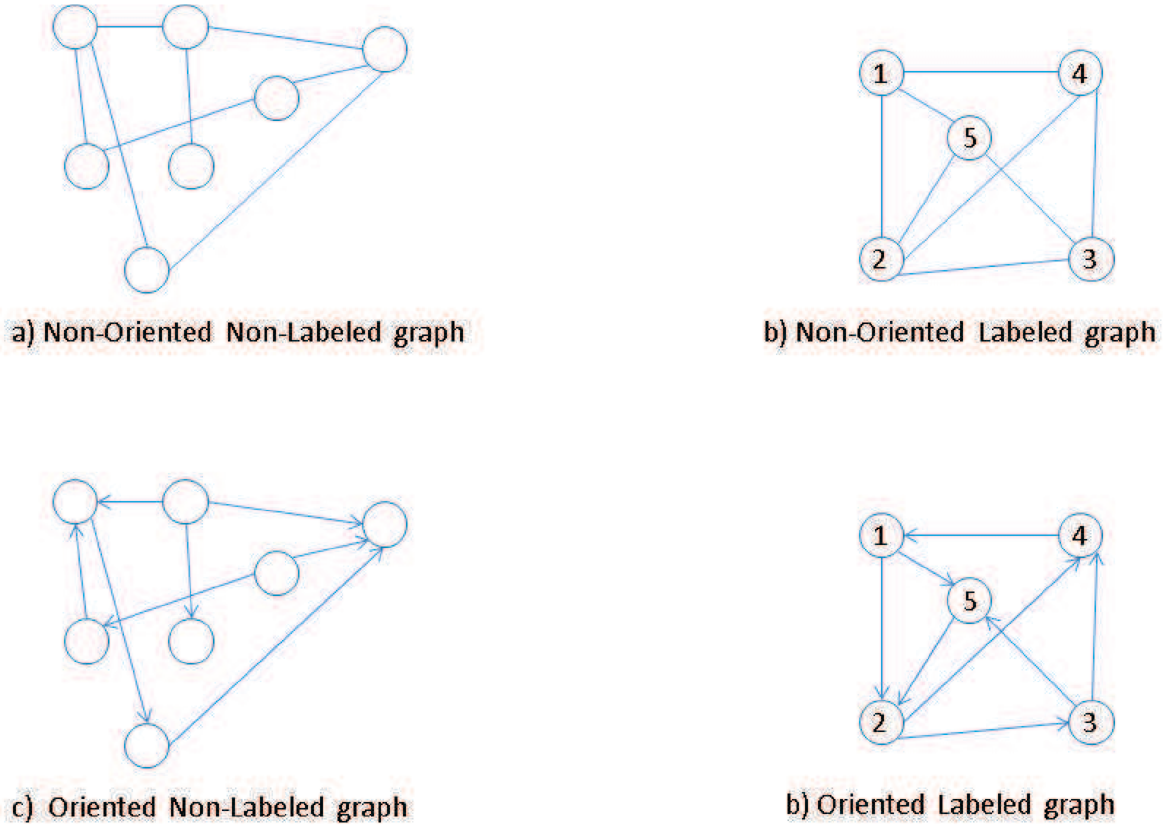


Figure 2.1: Example of Graphs.

In our thesis we consider data graphs defined as simple labeled graphs. Simple graphs are graphs with no edges involving a single vertex.

A graph that is contained in another graph is called a subgraph. Here some subgraph definitions.

Definition 2. *A graph* $G_1 = (V(G_1), E(G_1), \ell_1, \Sigma)$ *is a subgraph of a graph* $G_2 = (V(G_2), E(G_2), \ell_2, \Sigma)$ *if* $V(G_1) \subseteq V(G_2)$*,* $E(G_1) \subseteq E(G_2)$*,* $\ell_1(x) = \ell_2(x)\ \forall x \in V(G_1)$*, and* $\ell_1(e) = \ell_2(e)\ \forall e \in E(G_1)$*.*

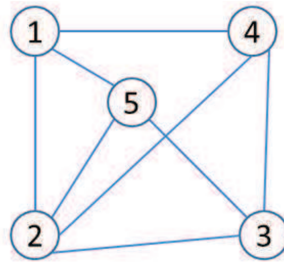
Definition 3. *Induced Subgraph. For two graphs* G_1 *and* G_2*, where:*
$G_1 = (V(G_1), E(G_1), \ell_1, \Sigma)$, $G_2 = (V(G_2), E(G_2), \ell_2, \Sigma)$. G_1 *is an induced subgraph of* G_2*, denoted* $G_1 \subseteq_i G_2$*, if:*

- $V(G_1) \subseteq V(G_2)$

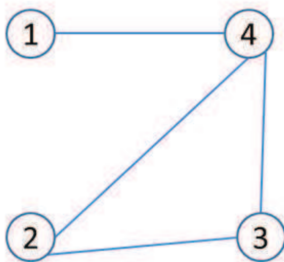
- $E(G_1) = E(G_2) \cap (V_{G_1} \times V_{G_1})$

Definition 4. *Partial Subgraph. For two graphs G_1 and G_2, where $G_1 = (V(G_1), E(G_1), \ell_1, \Sigma)$, $G_2 = (V(G_2), E(G_2), \ell_2, \Sigma)$. G_1 is a partial subgraph of G_2, denoted as $G_1 \subseteq_p G_2$, if G_1 is a graph that does not contain all the edges of G_1 having their ends in $V(G_1)$.*

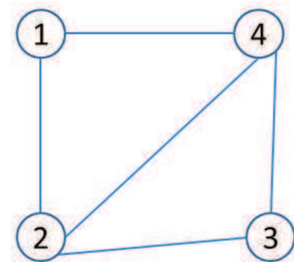
Figure 2.2, illustrates the concepts of induced and partial subgraphs.



a) A Graph G



b) A Partial subraph of G



c) An Induced subraph of G

Figure 2.2: Example of Induced and Partial Subgraphs.

2.2 Querying graph data

As a data structure, graphs are increasingly used to model data and complex objects. They allow to convey as much information as possible, to ensure an efficient representation of complex objects and also a relevant comparison between two objects. Thus various real applications such as social networks and protein interactions use graphs as a model of representation. Graphs can also represent complex relationships such as the organization of entities in images which can be used to identify objects and scenes.

In many cases, the success of an application based on a graph representation of data is directly dependent on the efficiency of an underlying graph query processing. Talking about graph query processing lead directly to one of the most popular problem in graph theory, which is graph and subgraph matching. Graph matching consists to find the correspondence between the vertices of two graphs which provides the best alignment of their structures. Generally, graph matching methods can be divided into two broad categories: Exact and inexact matching according to their results. In other words, exact graph matching returns graphs or subgraphs that match exactly a given graph, however, inexact matching returns a ranked list of the most similar matches.

Exact graph matching approaches aim to find out if an exact mapping between the vertices, and the edges of the compared graphs is possible. This requires a strict correspondence between the two objects being matched, or at least between subparts of them.
Graph Isomorphism is a variant of exact graph matching defined as follows:

Definition 5. *A graph* $G_1 = (V(G_1), E(G_1), \ell_1, \Sigma)$ *and a graph* $G_2 = (V(G_2), E(G_2), \ell_2, \Sigma)$ *are said to be isomorphic if there exists a bijective function* $h : V(G_1) \rightarrow V(G_2)$ *such that the following conditions hold:*

1. $\forall u \in V(G_1) : \ell_1(u) = \ell_2(h(u))$

2. $\forall (u,v) \in E(G_1) : (h(u),h(v)) \in E(G_2)$ *and* $\ell_1((u,v)) = \ell_2((h(u),h(v)))$

3. $\forall (h(u),h(v)) \in E(G_2) : (u,v) \in E(G_1)$ *and* $\ell_2((h(u),h(v))) = \ell_1((u,v))$

In exact matching we can find also other forms like maximum common subgraph, monomorphism, and homomorphism.

- The maximum common subgraph is the problem of finding the largest part of two graphs that is identical in terms of structure, which is refered to the maximum common subgraph.

- Monomorphism is a variety of exact graph matching where each vertex of the first graph is mapped to a distinct vertex of the second one, and each edge of the first graph has a corresponding edge in the second one. The second graph, however, may have both extra vertices and extra edges.

- A Graph Homomorphism f from a graph $G_1 = (V(G_1), E(G_1), \ell, \Sigma)$ to a graph $G_2 = (V(G_2), E(G_2), \ell, \Sigma)$, written as $f : G_1 \rightarrow G_2$, is a mapping $f : V(G_1) \rightarrow V(G_2)$ from the vertex set of G_1 to the vertex set of G_2 such that $(u,v) \in E(G_1)$ implies $(f(u), f(v)) \in E(G_2))$ but not vice versa.

Inexact graph matching means that it is not possible to find an isomorphism between the two graphs to be matched. This is the case when the query graph and the data graph have not the same number of vertices. The interest of inexact graph matching has been recently increased due to the application of graphs to areas such as cartography, character recognition, and medicine. In these areas, automatic segmentation of images results in situations where the data graph contains more vertices than the query graph. That is why applications on these areas do usually require inexact graph matching techniques [5]. Usually, inexact

matching algorithms do not impose the edge-preservation constraint used in exact matching.

In the two cases, exact and inexact matching, querying a data graph can be categorized into two classes: subgraph matching over a single large data graph, and subgraph containement search over a graph database. In both categories, algorithms first filter the list of candidates then verify isomorphism. However, the verification phase in subgraph containement search aim to check if there exisits one subgraph isomorphism for each graph candidate. But in subgraph matching over a single data graph, it aims to find all embeddings for a given query graph in a data graph.

Subgraph containement search over a graph database needs to index data graphs (from the graph database), that contain a query. Then, both filtering and verification mechanisms are performed after indexing.

There are different ways to index data graphs. The most known approaches use graph-features that include (tree-feature, frequent substructures, path-based indexing). The idea is to find the best way to represent candidates, in order to facilitate the isomorphism checking.

The filtering phase must be very effecient in order to minimize the list of candidates, which will minimize the verification cost. In the verification phase the final list of candidates given by the filtering algorithm will be parsed, and each candidate will be compared with the query. A candidate that matches the query will be added to the final list of results.

A lot of approaches were proposed to tackle the subgraph containement search problem, such as: gIndex [51], Tree+delta [59] FG-index [28], gCode [62], and others.

In our thesis, we focus on the second category: subgraph isomorphism search over a single large data graph which is more difficult than the subgraph contain-

ment search, because subgraph matching requires enumerating all embeddings. For a query graph Q, a data graph G, all embeddings of Q in G are to be ennumerated. Generaly, to tackle this problem authors aim to find the best way to visit vertices. The more non relevant vertices are filtered, the more effective the verification will be.

2.3 Subgraph isomorphism search over a single large data graph

In Subgraph isomorphism search, the main goal is to enumerate all occurrences of a query graph Q within a data graph G.

Definition 6. *Given two graphs $G = (V(G), E(G), \ell, \Sigma)$ and $Q = (V(Q), E(Q), \ell, \Sigma)$, Q is subgraph isomorphic to G if there is an injective function $f : Q \rightarrow G$ such that:*

1. $\forall v \in V(Q), f(v) \in V(G)$ *and* $\ell(v) = \ell(f(v))$.

2. $\forall (u,v) \in E(Q), (f(u), f(v)) \in E(G)$ *and* $\ell(u,v) = \ell(f(u), f(v))$.

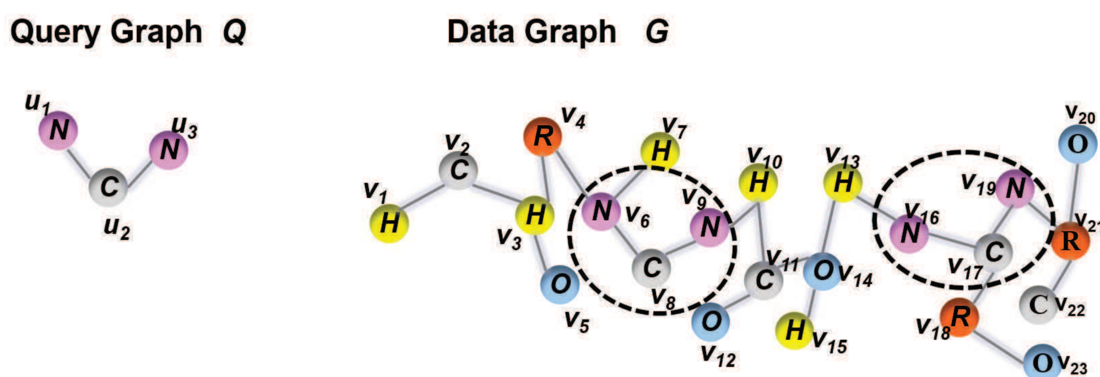


Figure 2.3: Subgraph isomorphism search.

Figure 2.3 depicts an example where the data graph contains two occurrences of the query graph.

The basic solution to enumerating the occurrences of Q into G is to directly compare the vertices of the query with the vertices of the data graph. This comparison constructs a search tree. In this search tree, each internal vertex maps a vertex of the query to a vertex of the data graph. Each path from the root to a leaf in the search tree represents either a unsuccessful mapping, between the query and a subgraph, that have been dropped by the algorithm, or a successful one that corresponds to a subgraph that is isomorphic to the query. Figure 2.4 presents a part of the obtained recursion tree for the query graph and the data graph of Figure 2.3.
Exploring this recursing tree is the main task of subgraph isomorphism search algorithms. Several existing algorithms use backtracking to explore the search tree, but they do not explore the whole tree. To avoid exploring the whole tree, they use filtering methods that prune unpromising branches of the tree. Nevertheless, even with pruning functions, this method rises two main challenges:

- It is memory consuming: besides storing the data graph which can have a significant size, exploring the search tree has a high memory consumption and involves complex data structures to support backtracking.
- It is time consuming: backtracking and testing the possible mappings between vertices has a high computational cost. It is exponential in function of the number of vertices in the involved graphs.

To deal with this, existing solutions perform filtering/prunning mechanisms to reduce the size of the search tree.

Existing algorithms are all about one basic point: the more effective and quick is the filtering, the earlier and easier is finding solutions. But this basic point

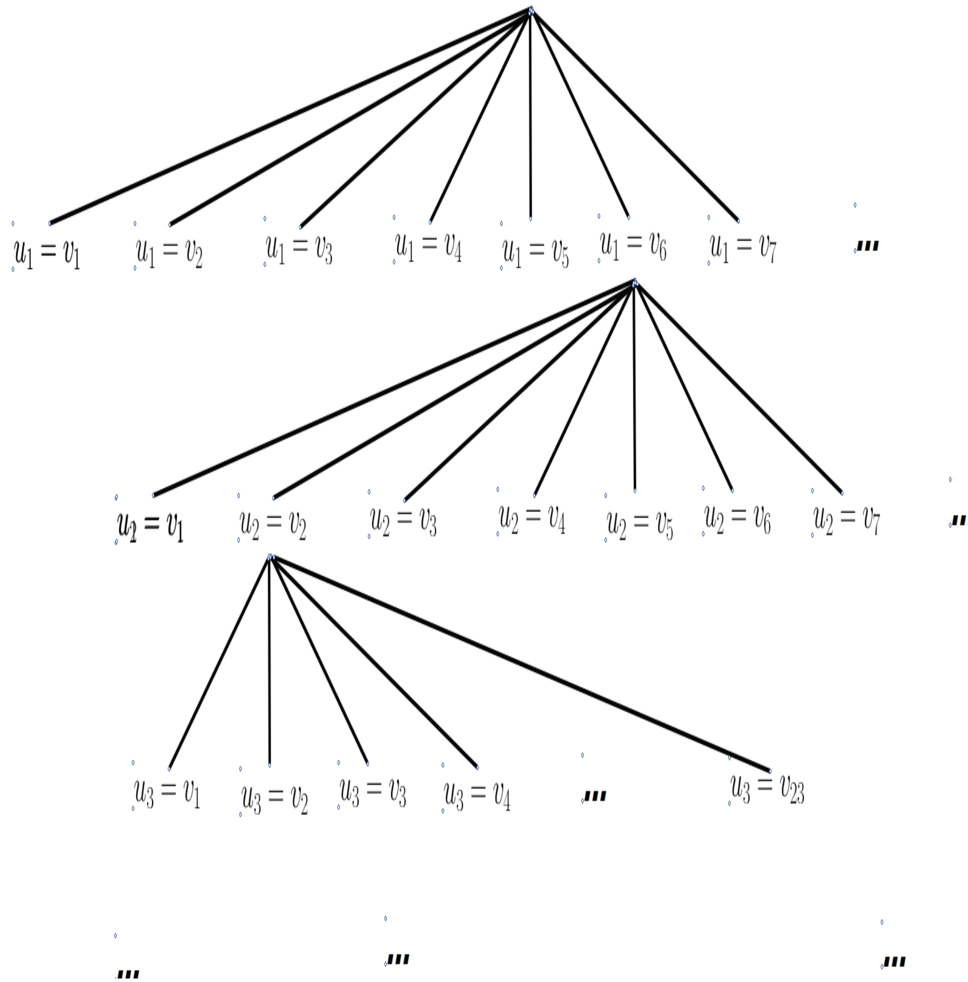


Figure 2.4: A partial construction of the search tree.

is addressed in diferent ways with different algorithms and methods. Globaly, algorithms go with a filtering and verification process to find all occurrences of the query in the data graph. The first task, which is the filtering, is the prunning mechanism that aims to delete irrelevant candidates in the search space. Filtering is an important step to determine the efficiency of the algorithm.

A good filtering algorithm is an algorithm that does not filter relevant candidates, filters the maximum of non relevant candidates, and achieves these two last tasks as fast as possible. With such a filtering, the final step of verification

will be more efficient and with less cost.

The second phase is the verification step which is generally based on the Ullmann's backtracking subroutine, that searches in a depth-first manner for matchings between the query graph and the data graph obtained by the filtering step (final list of candidates).

We first present the main algorithms of the state of the art. Then, we analyse them according to their filtering mechanism.

2.4 Existing Algorithms

In this section, we describe subgraph isomorphism algorithms, in a chronological order to show their evolution. First Ulmann's algorithm [49] addresses all forms of exact graph matching, but it is less suited for the maximum common subgraph problem. It is the first result in subgraph matching. Here, the algorithm matches vertices one by one according to the input order of query vertices. After Ulmann's algorithm, VF2, QuickSI, GraphQL, GADDI, and SPath algorithms were proposed to enhance Ulmann's algorithm performance. These algorithms use vertex neighboring informtation, and some other filtering techniques to remove fals-positive candidates, as soon as possible. SPath focuses on reducing the candidates of query vertices by exploiting neighborhood-based information. According to [57] SPath is more performing than GraphQL. On the other hand, VF2 is more performing than Ulmann's algotithm. In [33] authors use the word superior to compare two algorithms (SPath is superior to GraphQL, and VF2 is superior to ULmann).

Figure 2.5, illustrates existing algorithms for subgraph isomorphism search over a single large data graph, with a short explanation of each one. A relation of superiority between algorithms according to [33] is also given. The most efficient

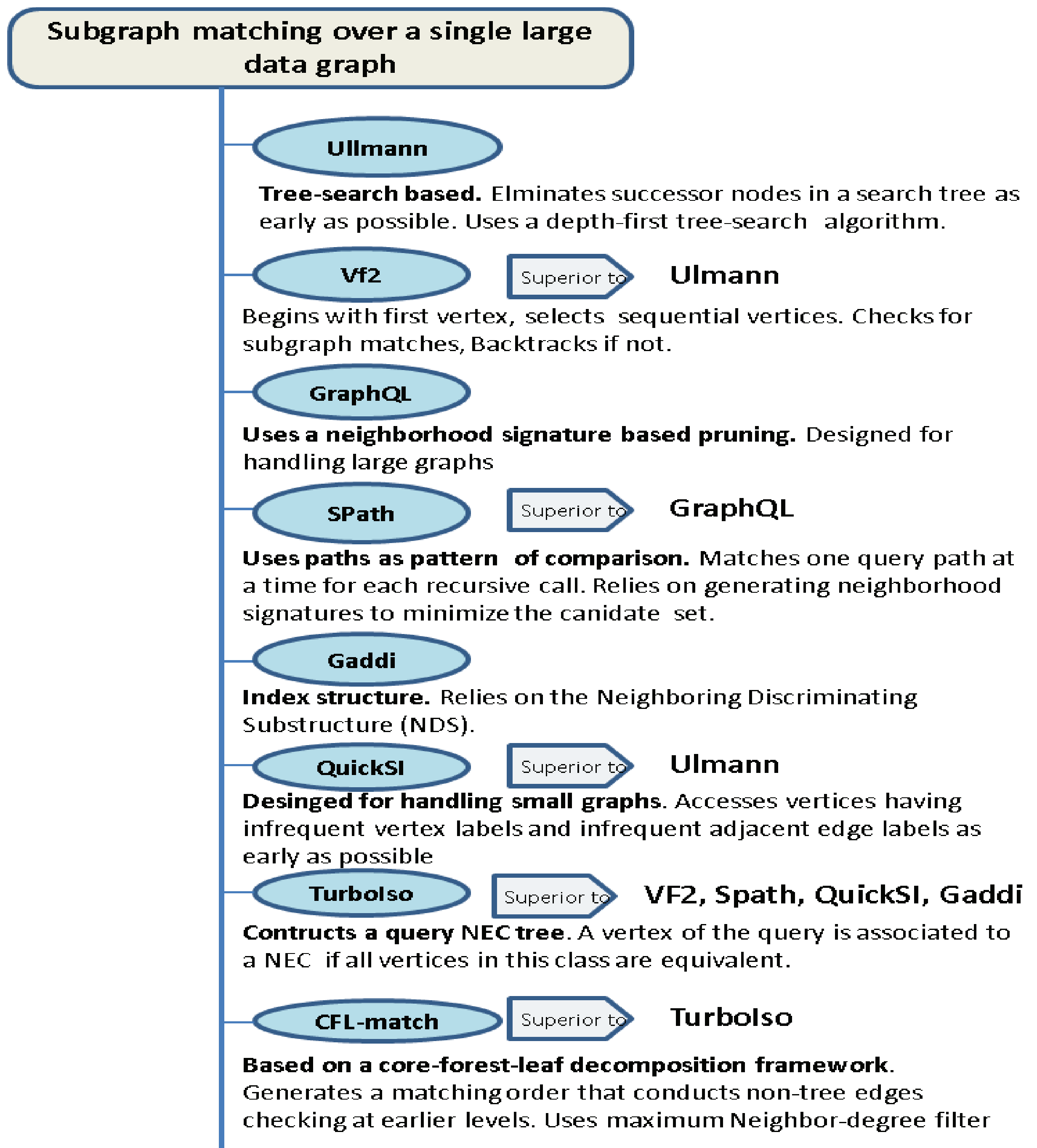


Figure 2.5: State of the art Methods.

algorithms that we found in the literature are Turboiso [23] and CFL-match [6]. In our contributions, we compare with these two algorithms to show the performance of our methods.

There are also several works that surveys existing algorithms. For instance, in [33], authors implements five algorithms VF2 [41], QuickSI[45], GraphQL [25], GADDI [55], and SPath [57] in a common framework. They use a generic subgraph isomorphism algorithm as a backtracking algorithm which finds solutions by incrementing partial solutions, or abandoning them. In this generic algorithm, authors selects first a group of candidate vertices. After that, the algorithm performs a recursive subroutine to find mapping pairs of vertices.

2.4.1 Ullmann's algorithm

Ullmann's algorithm [49] is a tree-search based algorithm. It is a backtracking algorithm. It is composed of two tasks: tree-search and refinement procedure. The algorithm begins with a data graph G, and a query graph Q. In the first task, i.e., tree-search, the algorithm finds a set of candidate vertices for each query vertex. Basically, for each vertex u in Q, the algroithm finds a set of candidate vertices $C(u) \subseteq V(G)$, such that $\ell(u) \subseteq \ell(v)$. Where ℓ, is the labeling function, and $v \subseteq V(G)$ is a candidate vertex. After that it invokes a recursive subgraph search subroutine. This subroutine finds a mapping between a query vertex and a data vertex. It takes one vertex at time. All reported embeddings, $Em(G)$, are stored as output of the subgraph search subroutine. The list of embeddings $Em(G)$ will be used in the refinement procedure.

After that, the second task, i.e., refinement procedure, is processed to minimize the computation time required for the subgraph isomorphism testing, by reducing the search space. To do this, the algorithm filters out candidate

vertices, $v \in C(u)$, that have smaller degree than the query vertex through all adjacent query vertices of u. If the adjacent vertex u' is already in the list of embeddings $Em(G)$, which means that: $(u', v') \in Em(G)$, then it checks if there is a corresponding edge (v, v') in the data graph G.

The complexity of Ullmann's algorithm depends on the size of the graph. Supppose that the size of the data graph G is n, and that of the query graph is m. The complexity of the algorithm in the best case will be: $O(nm)$. But it can go up to $O(m^n n^2)$ in the most case. As a result, the processing time explodes exponentially . So it is very expensive.

2.4.2 VF2

Generally, solutions to the subgraph isomorphism search prolem can be obtained by exhaustive search of all possible partial matching (Brute-force search) as we have seen in Ullmann's method. In order to further reduce the search space, the algorithm VF2 [41] presents a new concept of state space representation. A state s represents a partial solution of the correspondence between two graphs. M is the final set of partial solutions, $M(s)$ is a subset of M representing the current partial solution of state s. The transition between a state s and its successor s' corresponds to a new pair of matching vertices.
The algorithm starts with the first vertex, selects a vertex connected from the already matched query vertices, search for a subgraph match, and backtracks if not. The first search is basically the same as in Ullman's algorithm. The difference between them is in the refinement phase.
VF2 algorithm refinement phase uses a set of rules. Authors present these rules in two groups: syntactic feasibility rules, and semantic feasibility rules. These rules are as follows:

- Prune out any non connected vertex $v \in C(u)$ from already matched data vertices.

- Prune out any vertex $v \in C(u)$ such that, $|C_q \cap adj(u)| > |C_g \cap adj(v)|$. Where, C_q is a set of adjacent and not-yet-matched query vertices connected from the set of matched query vertices, M_q. C_g is a set of adjacent and not-yet-matched data vertices from the set of matched data vertices, M_g.

- Prune out any vertex v in $C(u)$ such that, $|adj(u)\ C_q\ M_q| > |adj(v)\ C_g\ M_g|$.

VF2 is an improved version of VF algorithm which was proposed by the same author in [14]. It has reduced the memory requirement from $O(n^2)$ in VF to $O(n)$ (in VF2) for n vertices. Unlike VF2, VF algorithm does not define any order in which query vertices are selected.

2.4.3 SPath and GraphQL

In Spath [57], paths are used as patterns of comparison. Basically, paths are used in the matching phase instead of matching each single vertex. SPath uses neighborhood signatures to minimize candidate sets. In the data graph processing, the algorithm computes a neighborhood signature for each vertex. A neighborhood signature ($NS(u)$) of a vertex u is computed as follows: $NS(u) = \{S_k(u)|k \leq k_0\}$ where, $S_k(u)$ is the k-distance set of u. Each element in $S_k(u)$ is relative to a label l denoted as $S_k^l(u)$. An element contains a set of vertices v_i where v_i satisfies: $d(u, v_i)$, and $l(v) = l(u)$.

In the other hand, neighborhood signatures are also computed for each vertex v in the query graph. After that, a filtering mechanism is used in order to minimize the matching candidate set, $C(v)$. To test if a given vertex u in $C(v)$ must be pruned or not, authors compares $NS(v)$ and $NS(u)$ to see if there

is a containement or not. This last is defined in [57] as: $NS(v) \sqsubseteq NS(u)$ if $\forall k \leq k_0, \forall l \in \Sigma$, then, $|\bigcup_{k \leq k_0} S_k^l(v)| \leq |\bigcup_{k \leq k_0} S_k^l(u)|$. For each u in $C(v)$ if $NS(v)$ is not contained in $NS(u)$, then u is safely pruned. After that, the shortest path of each vertex u in the dataset, is generated from its matched vertex v in the query graph up to k' by-product of $NS(v)$. The shortest path of u is denoted as p_u. The set of shortest paths must cover all query graph's edges (this set is constructed jointly). And finally the isomorphism testing is done by an edge-to-edge matching.

GraphQL was introduced earlier to SPath. SPath has better performance as compared to GraphQL. Both perform neighbourhood-signature-based pruning before starting a subgraph matching procedure. Spath performance is directly related to the neighbors scope. Which means that, the bigger the neighborhood scope is, the more the filtering time of Spath increases. Also, Spath uses costly join operations [57].

2.4.4 GADDI

GADDI [55] computes a neighbourhood discriminating structure distance between pairs of neighbouring vertices of the data graph. Then, the algotithm launches a subgraph matching algorithm. This one applies a two-way pruning and incorporates a dynamic matching schema.

The neighborhood discriminating distance is based on frequent substructure count. First the algorithm generates intersecting subgraphs between each pair of vertices from their neighborhood's sets. After that to prune out unpromising subtructures, discriminative substructures are selected from the frequent ones. A discriminative substructure is denoted as $DS(G)$. A substructure is to be called frequent, if it is subgraph isomorphic to at least half of the intersecting subgraph

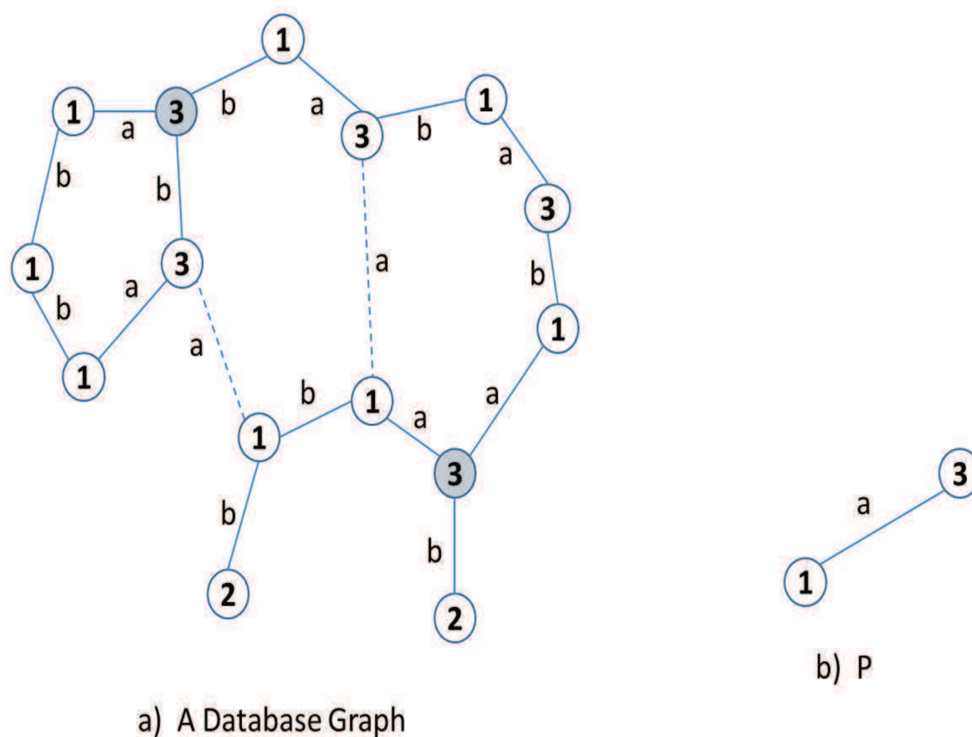


Figure 2.6: the NDS distance [55]

of two vertices v_1 and v_2 in the data graph. After finding a $DS(G)$ for a given substructure P. A new NDS (Neighboring Discriminating Substructure) distance denoted as $d_{NDS}(G, v_1, v_2, P)$, is calculated for each pair of neighboring vertices as an index structure.

This distance is the number of matches of P in the intersecting subgraph $Int(G, v_1, v_2)$. In the example of Figure 2.6, Length =4 (Length is the upper bound of the shortest distance between a pair of vertices to be indexed), k=3 (the k-neighboring), P is a substructure. The distance between the two filled vertices is three because there are three matches in their intersecting subgraphs. Matches of the discriminative substructure in the interscting subgraphs are marked by dashed lines.

The matching phase is processed as follows :

1. Vertex matching and distance-based pruning: a vertex in the data graph is matched to one vertex in the query graph using a depth-first matching algorithm. For the new matched vertices, a two-way-pruning strategy is processed. The first way is by fixing the query graph's vertex and searching for data graph's vertices that can match it. To do this, $\forall v \in G, \forall u \in Q, d(Q,u,v) \leq Length$, find: $v' \in G, d(G,v,v') \leq length$, where: 1) $\ell(v) = \ell(v)$, 2) $d(Q,u,v) \geq d(G,v,v')$, and 3) $d_{NDS}(Q,u,v) \leq d_{NDS}(G,v,v')$. The second way-pruning consists to fix the data graph's vertex and search for query graph's vertices that can match it. With the same conditions as the first way. After that the algorithm prunes vertices in the data graph that does not participate in the current matching.

2. Dynamic matching algorithm: this algorithm aims to find all possible matches for the query graph. To avoid useless calculations, the algorithm stores edges after matching a query graph vertex with a data graph vertex for the first time. If they verify the three conditions mentioned above, two lists are created for each vertex in the data graph. One represent the list of query vertices that the data graph's vertex can be matched to. The second one is the list of query vertices with which the data graph's vertex cannot be matched to.

2.4.5 QuickSI

QuickSI [45] is a subgraph isomorphism search alrorithm, developed to support large graphs. QuickSI tries to access vertices having infrequent vertex labels and infrequent adjacent edge labels as early as possible. The algorithm computes vertices' frequences before starting the search procedure. QuickSI relies on the concept of QI-sequence to reduce the search space. A QI-sequence of a graph

G is a rooted spanning tree for a query Q. A spanning tree of Q is a subgraph that is a tree which includes all of the vertices of Q, with minimum possible number of edges. The QI-sequence is represented in [45] as a regular expression : $SEQ_Q = [[T_i R^*_{ij}]^\beta]$. Where β is the number of vertices in Q, T_i is a spanning entry (a vertex in the spanning tree). For each $i \in [1, \beta]$, T_i contains: $T_i.v$ that records a vertex, and $[T_i.p, T_i.l]$. $T_i.p$ is the parent of $T_i.v$, and $T_i.l$ stores the label of $T_i.v$. Also R_{ij} represents extra entries that are stored. These extra entries may be vertices' degree, or edges that do not appear in the spanning tree.

To choose an effective QI-sequence, the query graph Q is converted to a weighted graph according to the average number of possible mappings from Q to a data graph G. A minimum spanning tree is based on edge weights. This one is to be used to generate a QI-sequence of Q. Vertices' weights are used to determine the order of the first two entries. Using this, the QuickSI algorithm checks if there is a sequence of a subgraph g in G, which is identical to SEQ_q. If two sequences are identical, then the corresponding graphs must be identical. Quick-SI adopts a depth-first-search order following the order of sequences in SEQ_q.

After that, a new filtering approach is proposed to reduce the subgraph isomorphism testing cost. The filtering uses an index called swift-index, based on two techniques : a prefix-pruning technique and a prefix-sharing technique. Swift-index uses tree features instead of subgraph features. A tree feature is organized by a prefix tree, where each vertex is an entry T_i of a tree feature sequence. A prerfix SEQ^i_f is an induced subgraph of the tree feature f against its vertices. By processing a depth-first search on the prefix-tree, if a prefix SEQ^i_f cannot be mapped to q, then the feature f that corresponds to this sequence is prunned. No need to see all the sequence. A prefix sharing is used when features share some prefixes in the prefix-tree. The idea is to keep one mapping and

replace it with another one after using it against all features with the shared prefix.

To control the index size, only frequent and discriminative trees are chosen. The frequency of a feature f, is computed by $frq(f) = \frac{|\{g|f \subset g \wedge g \in D\}|}{|D|}$, f is frequent iff $frq(f) \geq \delta$, where $\delta \in [0,1]$. Also a discriminative measure $dis(f)$ is defined as: $dis(f) = \frac{|f.list|}{|\bigcap\{f'.list|f' \subset f \wedge f' \in I\}|}$. f is discriminative iff $dis(f) < 1 - \gamma$, where $\gamma \in [0,1]$.

2.4.6 Turbo-iso

Turbo$_{iso}$ is a subgraph isomophism search solution [23] that focuses on solving the matching order selection problem, the blind exploitation of neighborhood, and the permutation problem of the existing algorithms.

First of all, Turbo$_{iso}$ constructs the query NEC tree. A NEC is a class that contains a group of vertices. A vertex v of a query graph Q is associated to a NEC, if all vertices in this class are equivalent to v. Two vertices are said to be equivalent if they have the same label, and for each embedding m, there is another embedding m' which contains m without the matching of this two vertices with their permutaion. m contains the matching of two vertices.
Rewriting the query graph into a NEC tree is performed by a BFS search, starting with a start query vertex. The start vertex is chosen by a ranking function: $Rank(u) = \frac{freq(G.l(u))}{deg(u)}$, $freq(G,\ell)$ is the number of data vertices in G that have label l.

After this phase, a candidate region exploration is processed. It identifies candidate regions which are subgraphs of the data graph where there is more chance to find embeddings for the query graph. When performing a subgraph isomorphism search on all candidate regions, all embeddings can be found. Next, a matching order is obtained by sorting each path in the NEC tree in ascending

order according to its number of candidate vertices. Finally, according to the matching order obtained and using candidate data vertices of the NEC vertices, a recursive routine is processed for subgraph search.

Turbo-iso was tested and also compared with the most performing algorithms such as VF2 and QuickSI and showed better performance.

2.4.7 CFL-match

CFL-match [6] is the most recent method to search for subgraph matching. First the query graph is decomposed into three substructures. Then subgraph matching is performed on each of these substructures. This algorithm is based on a core-forest-leaf decomposition of the query graph. It generates a matching order that conducts non-tree edges checkings at earlier levels. It aims to postpone cartesian products. CFL-match [6] uses first a costly filtering method. It uses the neighborhood label frequency filter to ensure that a data vertex is a candidate. CFL-match proposes after that another way to filter in order to reduce the time processing of the first filter. The new filtering mechanism is the maximum Neighbor-Degree filter. It can be verified in constant time for each candidate data vertex. In the Core-Forest Decomposition, authors use a spanning tree Q_T of the query Q in which, the group of edges of Q that are not in the set of edges of Q_T are called non-tree edges. The rest of edges are called tree edges. For each set of non-tree edges of a spanning tree in Q, the core-forest decomposition computes a small dense subgraph that contains the set of non-tree edges. This subgraph is the minimal connected subgraph. The subgraph composed of the non-tree edges of Q is called the core-structure of Q. For the rest of edges (tree edges), the subgraph is called forest-structure of Q and denoted T. After the core-forest decomposition, there is the forest-leaf decomposition. Here, the

forest-structure T is decomposed. into a forest-set V_T and a leaf-set V_I. All leaf-set vertices will be at the end of the matching order.

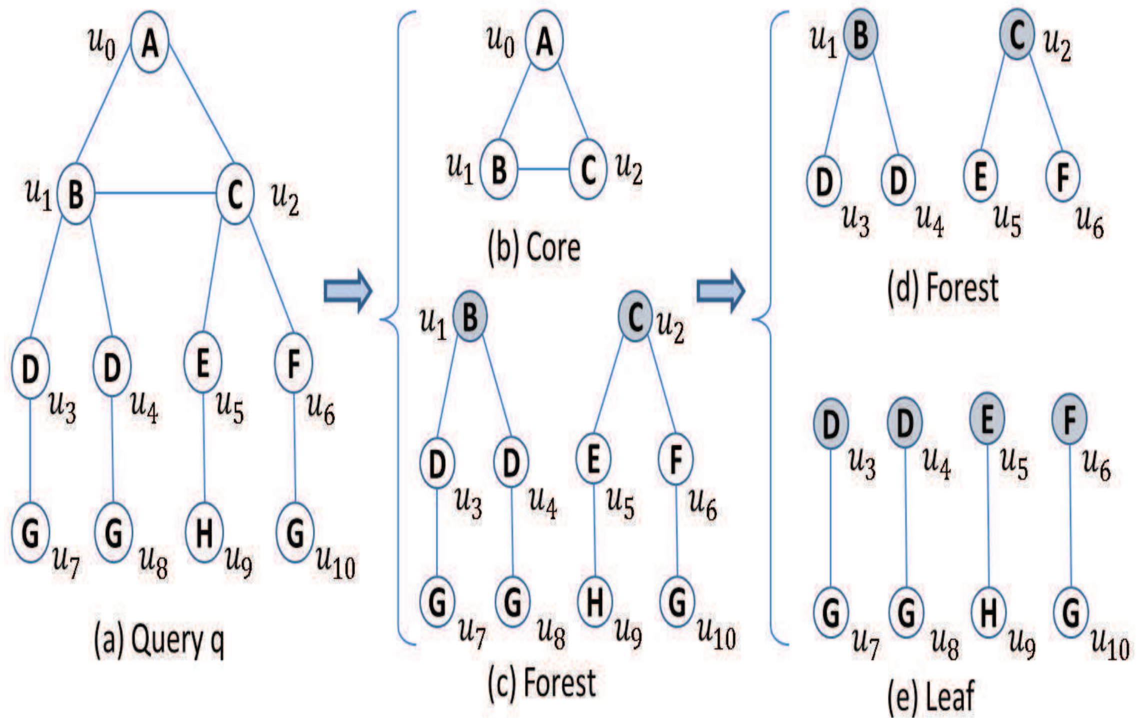


Figure 2.7: Core-Forest-Leaf Decomposition [6]

To encode all possible embeddings of a query in a given data graph, a data structure called CPI (compact path-index) is built. Figure 2.8 shows an example of CPI. In this Figure, we can see that the candidate set of u_0 and u_1 are $u_0.C = v_0, ..., v_4$ and $u_1.C = v_5, ..., v_9$. After that, authors use a specific CPI-based Matching Order Selection. All embeddings of Q, in the data graph G, are sorted by parsing the CPI. G is used only for non-tree edges checking. Figure 2.9 shows a running example, here we can see the constructed CPI for the query graph Q over the data graph G. Finally after constructing CPI and with a related matching order, a CPI-based Forest-Match, and a CPI-based Leaf-Match are

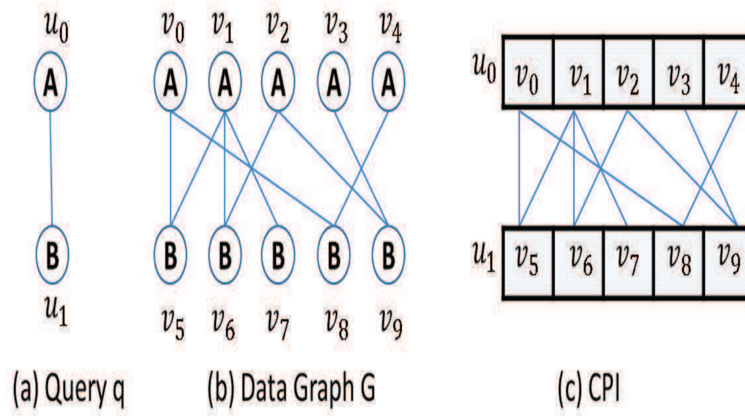


Figure 2.8: Example CPI [6]

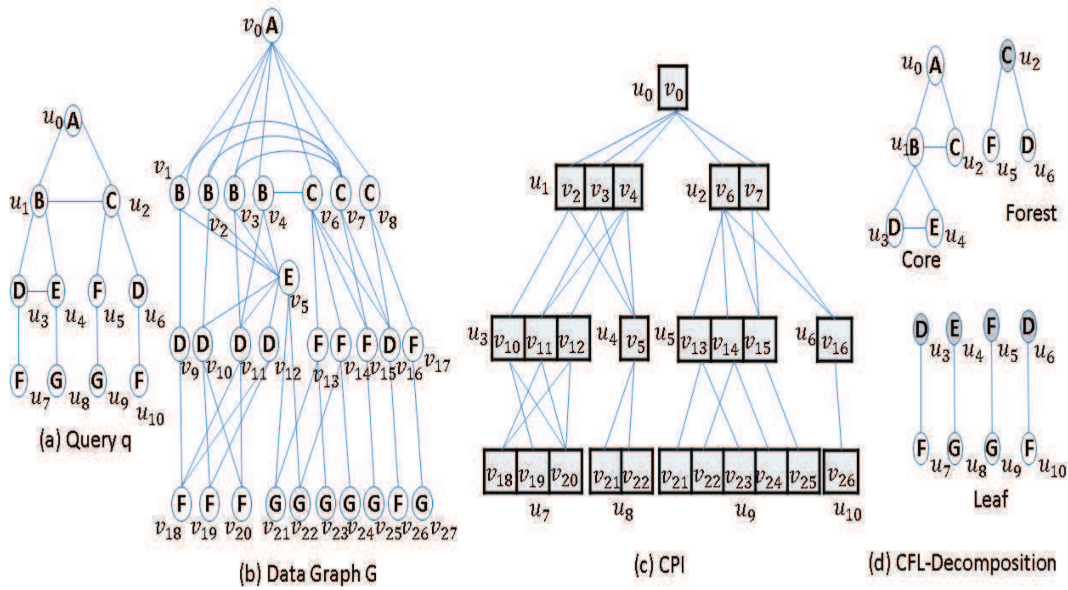


Figure 2.9: Running Example [6]

performed to find CPI-based Embeddings.

2.4.8 Other Methods and techniques

Other methods that are more general or that use specific computing plateforms exist for subgraph isomorphism search. We survey, in the following, some of them. The Nauty algorithm, of Brendan McKay [38], detects isomorphism between untyped graphs that may be directed or undirected. Nauty uses transformations to reduce graphs to a canonical form that may be checked relatively quickly for isomorphism. Specifically, the algorithm computes invariants for each vertex in a graph (e.g., degree and counts of adjacent vertices) that are used for candidate selection. Nauty partitions a graph into non-overlapping subsets such that the vertices in a particular subset share identical invariant values. Subsets having the same invariant values can then be compared across graphs. If all subsets are isomorphic between two graphs, then the two graphs must be isomorphic. Alternatively, if two graphs contain subsets with differing invariants, there is no need to test isomorphism between the sets directly.

Authors in [47] propose a subgraph matching algorithm for very large graphs deployed on a distributed memory store. They use the Trinity memory cloud, which provides a unified adress space for a set of machines, as if a large graph is stored in one machine.
The subgraph matching of this method is showed on Figure 2.10. It needs three steps:

1. Query decomposition and STwigs ordering. A Query graph is decomposed into a set of STwigs. An STwig is a two level tree structure. It contains a root and the list of its child vertices. An STwig is the basic unit of graph access. To find a matching between STwigs a MatchSTwig() function is

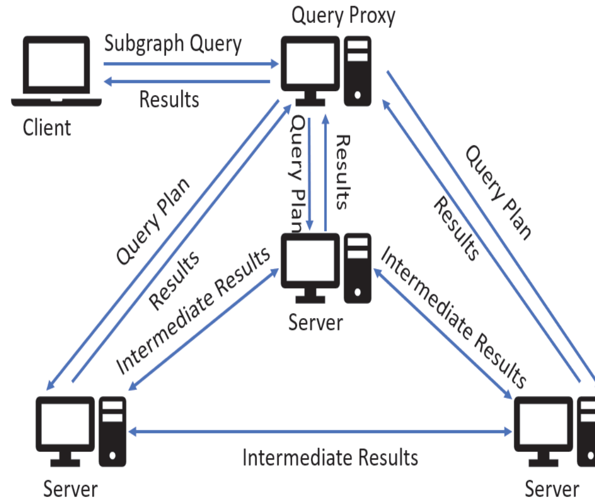


Figure 2.10: Subgraph isomorphism search on Trinity[47]

used, this function returns a set of STwigs that match the query.

2. Exploration. Using the order selection of STwigs, the list of STwigs is processed one by one. For each label a of the STwig q_1, a set of vertices that match a is sorted. After that the algorithm compares the union of STwigs labels' sets to see if they can produce an answer on their own. Each STwig provides its group matching, for example for q_1, a group $G(q_1)$ is sorted. All STwigs are processed until all vertices of the query are bound.

3. Joins. A sequence of results $(G(q_1), G(q_2), ..., G(q_k))$ is generated in the exploration step. This sequence is joined to produce a final answer. This join is based on a sample-based join cost estimation method and cost-based join order selection method [47].

TMODS [20] is another algorithm proposed for subgraph isomorphism search. It uses a set of genetic algorithms to find exact and inexact pattern matches in directed, attributed graphs. Patterns may specify both structural and attribute characteristics.

TMODS searches for patterns from bottom to up, finding sub-patterns first and then composing them into more complex higher-level patterns.

2.5 Analysis

We studied the above algorithms specially according to their pruning mechanisms. We distinguish three main prunning mechanisms that can be used separately or combined for more efficiency:

1. Vertex properties: This technique is a local pruning mechanism. A local pruning prunes the set of mappings that are candidates for a single vertex. Vertex properties such as the degree and the label, are powerful pruning mechanism used by most algorithms. In fact, the final search space is the result of joining of the sets of available mappings of each query vertex. Thus, given a query graph $Q = (V(Q), E(Q), \ell, \Sigma)$ and a data graph $G = (V(G), E(G), \ell, \Sigma)$, the aim is to reduce as much as possible the sets $C(u_i)$, $i = 1, |V_Q|$, where $C(u_i)$ is the set of vertices of the data graph that match the query vertex u_i. The final search space is obtained by joining these sets, i.e., $C(u_1) \times C(u_2) \times \cdots \times C(u_{|V_Q|})$ [26]. The reduction of $C(u)$ is generally achieved using the neighborhood information of u. The amount of the obtained pruning depends on the scope of the considered neighborhood. The simplest solution considers the one-hop neighborhood such as the degree of the vertex and/or the labels of the neighbors. Neighborhood at $k-$hops is also used in some methods. Ullmann's Algorithm [49] refines C_u by removing the vertices that have a smaller degree than u. GraphQL [25] also uses the direct neighborhood by encoding within a sequence, the labels of the neighbors of each vertex. Furthermore, GraphQL uses an approximation algorithm proposed to further reduce the search space by discarding

the data vertices that are not compatible with the query vertex using the $k-$neighborhood around u. VF2 [41] looks to 2-hops neighborhood. SPath Spath [57] uses the k-neighborhood by maintaining for each vertex u a structure that contains the labels of all vertices that are at a distance less or equal than k from u. SPath uses its encoding of the k-neighborhood to remove the data vertices that have a k-neighborhood that does not englobe any k-neighborhood of query vertices. By rewriting, the query within a tree, QuickSI [45] and Turbo_{ISO} [23] use also the $k-$neighborhood with the particularity that the neighborhood is rooted at a more pruning vertex. The tree representation of Turbo_{ISO} is also more compact as it aggregates similar vertices.

2. Matching order: the matching order is a global prunning mechanism. a global pruning operates on the whole search space. The matching order determines in which order query vertices are handled and in which order data vertices are targeted. This order has also an important pruning capacity. For the data graph, the idea is to avoid a blind traversal of the search space and target specified regions on the data graph for subgraph isomorphism search. These regions are selected according to the properties of the query vertices. A candidate region for a query graph Q is a subgraph of the data graph G which may contain embedding of the query graph. So, performing subgraph isomorphism search on all candidate regions will ensure that all embedding can be obtained. However, minimizing the number of candidate regions and the size of each region is obviously important for faster matching. For this, regions are selected around the less popular query vertex in the data graph. For example, the authors of

[60] rank every query vertex u by :

$$rank(u) = \frac{freq(G, \ell(u))}{deg(u)} \tag{2.1}$$

Where $freq(G,l)$ is the number of data vertices in G that have label l, and $deg(u)$ means the degree of u. This ranking function favors lower frequencies and higher degrees which will minimize the number of regions. This ranking is also used in [23] in a more recent solution called Turbo$_{ISO}$.

However, when the labels and the degrees are not discriminative, this ranking becomes obsolete leading to visiting the whole search space.

For the query graph, the matching order means that the query vertices are handled in an order that simplifies their matching. For this, the most used order is equivalent to tree traversal of the query vertices. A spanning tree of the query is constructed according to a ranking equivalent to the one given by Equation (1). In [26, 46, 56, 58] the root of the tree is the less popular vertex. This solution is also adopted by Turbo$_{ISO}$ [23]. However, Turbo$_{ISO}$ goes further by grouping the vertices that have the same labels and the same neighborhood. The obtained smaller tree is called a NEC tree. Turbo$_{ISO}$ constructs candidate regions for the query Q in the data graph G by constructing for each region a BFS search tree T_G from the root vertex u'_s of the NEC tree Q' so that each leaf is on the shortest path from u'_s. Then, for the start vertex v_s of each target candidate region, identify candidate data vertices for each query vertex by performing depth-first search using T_G and starting from v_s. Turbo$_{ISO}$ reduces the number of regions using the ranking function given by Equation 2.1. When exploring candidate regions, Turbo$_{ISO}$ also minimizes the number of enumerated partial solutions by

ordering the NEC tree vertices by increasing sizes. Thus, paths involving fewer vertices are explored first, the space is pruned if no isomorphism is possible. In [23], Turbo$_{ISO}$ is compared to the other approaches and its superiority in processing queries is attested via extensive experimentations.

3. Query rewriting: is also a glabal prunning mechanism. Consists on representing the query in a form that simplifies its matching. Ullmann's algorithm and Spath do not define a global pruning mechanism and picks the query vertices in a random manner. VF2 and GADDI handle a query vertex only if it is conneceted to an already matched vertex. However GADDI uses an additional mechanism: a distance based on the number of frequent sub-structures between the K-neighborhoods of two vertices as a mean to prune globally the search space after each established mappings between a query vertex and a data vertex. QuickSI rewrites the query in the form of a tree: a spanning tree of the query. Edges and vertices of the query are weighted by the frequency of their occurrence in the data graph. Based on these weights, a minimum spanning tree is constructed and used to search the data graph. GraphQL selects the vertex that minimises the cost of the ongoing join operation. The cost of a join is estimated by the size of the product of the involved sets of vertices. Turbo$_{ISO}$ uses the ordering introduced in [60]. This ordering uses the popularity of query vertices in the data graph. Every query vertex u is ranked by $rank(u) = \frac{freq(G,\ell(u))}{deg(u)}$ introduced above. Furthermore, Turbo$_{ISO}$ rewrites the query within a tree using this ranking as in QuickSI but Turbo$_{ISO}$ aggregates the vertices that have the same labels and the same neighbors into a single vertex. This aggregation has been extended to data graphs in [43]

2.6 Conclusion

In this chapter, we presented and discussed the state of the art related to subgraph isomorphism search algorithms. We focused on the more recent and interesting methods but we also reviewed general approaches as well as those dedicated to specific plateforms. We then analysed these algorithms according on how they filter the search space.

In the next chapter, we introduce our first contribution that solves subgraph isomoprhism search problem on compressed graphs in order to deal with large graphs on a simple commodity machine.

Chapter 3: SUBGRAPH ISOMORPHISM SEARCH ON COMPRESSED GRAPHS

In this chapter, we present our first contribution: a subgraph isomorphism search that operates on compressed graphs. Our goal is to reduce the amount of memory used to store graphs, but also reduce the search space within a backtracking-based subgraph isomorphism algorithm. This explains why we use compressed graphs. With compressed graphs the processing time is also reduced. Our challenge is to query these compressed graphs without decompressing them. We analyze the processing time of the proposed algorithm and conduct extensive experiments on several datasets, with different sizes to attest the effectiveness of the proposed approach.

Contents

3.1 Introduction **43**

3.2 Graph Compression **43**

3.3 Compress and Search **51**

3.3.1 Candidate Supervertex Selection 53

3.3.2 Subgraph Search 57

3.4 Performance Evaluation **60**

3.4.1 Datasets 61
3.4.2 Results 65
3.4.3 Discussion 66
3.5 Conclusion **73**

3.1 Introduction

We recall that the problem of subgraph isomorphism search is the problem of finding the embedding of a query graph into a target graph called a data graph. The data graph is generally larger than the query and may contain several occurrences of it. The objective is then to enumerate all the occurrences of the query graph within the data graph in a reasonable time.
Given a query graph Q and a data graph G, a straightforward solution to enumerating the occurrences of Q into G is to directly compare the vertices of the query with the vertices of the data graph. This comparison constructs a search tree where each vertex of the tree corresponds to a mapping between a vertex from the query with a vertex from the data graph.

In this chapter, we target the memory and time consuming challenges related to parsing this search space by a new framework that handles efficiently the problem of querying graph data with subgraph isomorphism search. Our solution aims to deal with subgraph isomorphism challenges in massive graph databases through graph compression.
We first present graph compression and its algorithms, then we present an approach that ensites to search for subgraph isomorphism on compressed graphs.

3.2 Graph Compression

Graph compression, also known as graph summarizing, aims to reduce the amount of storage space required for storing graphs. It offers simple representations of huge graphs. Generally, wa are interested in compressing methods that do not require decompression to process the graphs. So, this summaries must retain an amount of the graph properties that are sufficient to the application.

Graph compression has the following advantages [53, 2].

1. Reduction of data volume and storage.

2. Speedup of graph algorithms and queries: in general, an algorithm that is executed on the compressed version of the graph is more efficient in term of time processing than when executed on the original graph.

3. Interactive analysis support: summarization is used to handle information extraction and to speed up user analysis.

4. Noise elimination: real graph data interfere with many hidden, or erroneous links and labels. Summarization is used to filter out noise and reveal patterns that exist in the data.

It has also several challenges [53]:

1. Volume of data: summarization techniques reduce the size of graphs but the challenge is with large graphs. Summarization methods need to process large amounts of data, and also to be able to handle scaling.

2. Complexity of data: the heterogenity of nodes and edges continues to increase within graphs, and other information such as labels or other data, from different sources, need specific design and quantification.

3. Definition of interestingness: it is difficult to determine, in an obective way, if an information is interesting or not.

4. Evaluation: evaluation of summaries becomes more difficult when more elements, such as visualization and multi-resolution summaries, are involved.

5. Change over time: the question is how to perform efficient analysis on dynamic data.

Several works published in the domain of graph compression concern the compression of the web graph [2, 42, 8]. These works aim to reduce the size of the web graph. The proposed compression focuses on compressing the edges between vertices. The main approach uses references between vertices that share a subset of edges. Compression of data graphs takes a different approach that we can qualify as label-oriented or query-oriented. Label-oriented compression compacts vertices that have the same label into a kind of supervertices as follows:

Definition 7. *[12] Given a labeled graph G with $V(G)$ partitioned into groups, i.e., $V(G) = V_1(G), V_2(G), \cdots, V_k(G)$ such that:*

1. $V_i(G) \cap V_j(G) = \phi, 1 \leq i \neq j \leq k$
2. *all vertices in $V_i(G), 1 \leq i \leq k$, have the same labels.*

We can summarize G into a compressed version $comp(G)$ where:

- *$comp(G)$ has exactly k vertices $v_1, v_2, \cdots, v_k$ that correspond to each of the groups of $V(G)$ (i.e., $V_i(G) \mapsto v_i$). The label of v_i is set to be the same as those vertices in $V_i(G)$, and*
- *an edge (v_i, v_j) with label l exists in $comp(G)$ if and only if there is an edge (u, u') with label l between some vertex $u \in V_i(G)$ and some other vertex $u' \in V_j(G)$.*

Figure 3.1 illustrates the compression (graph (b)) of the graph given in (a). Each vertex of the compressed graph is a group of vertices having the same label. However, this compression does not retain all the structural information available in the original graph. For example, the edge between the vertex labeled b and the vertex labeled d in Figure 3.1(b) cannot inform us if this edge links

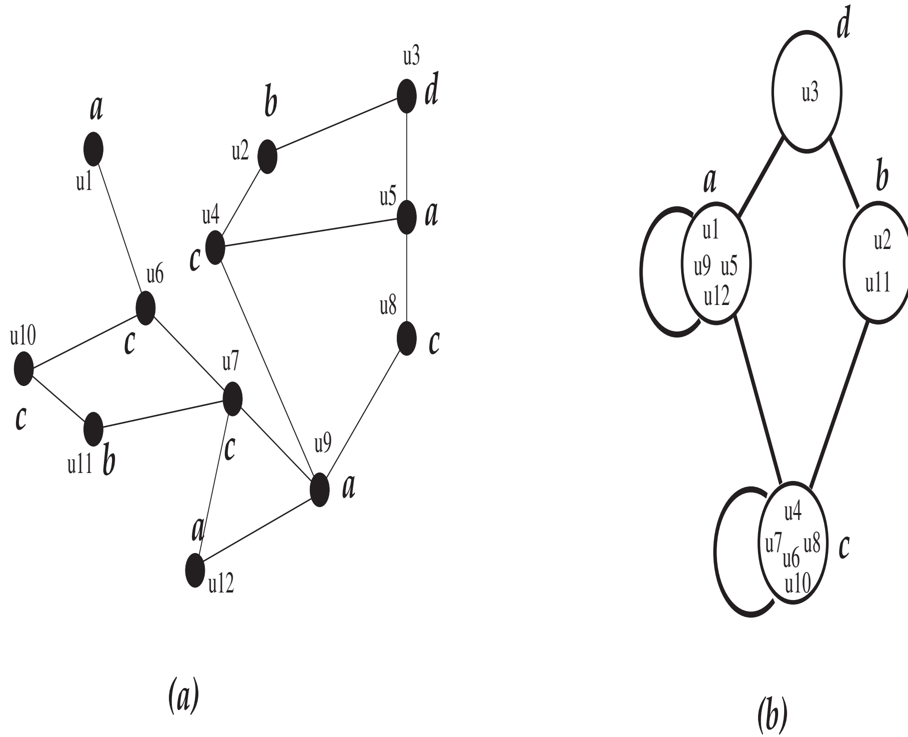


Figure 3.1: Graph compression with [12].

u_3 to u_{11} or u_3 to u_2. This means that the algorithms that use the compressed graphs do not aim to have exact solutions but approximative ones.

In [12], authors propose an algorithm that finds all frequent subgraphs in a database of large graphs where the graphs are compressed according to Definition 7.

We also have query-oriented compression which is a compression that preserves a kind or a class of queries [3, 27, 10, 37, 50, 16]. Given a graph G and a kind of queries Q, i.e., path queries, neighborhood queries, reachability queries, etc., the compression constructs a smaller graph G' such that the results on G for all queries in the class Q are equivalent to their results on G'. The compression function depends of the kind of the queries. For example, the compression function is equivalent to the one given by Definition 7 for pattern queries and it

groups the vertices that have the same neighbors for reachability queries. Grouping the vertices that have the same neighbors in a graph is a well known concept in graph theory called modular decomposition. Modular decomposition has been introduced by Gallai [17] to solve optimization problems. It is used to generate a tree representation of a graph that highlights groups of vertices that have the same neighbors outside the group. These subsets of vertices are called *modules*.

Definition 8. *A module of a graph $G = (V, E)$ is a set $M \subseteq V$ of vertices where all vertices in M have the same neighbors in $V \setminus M$.*

Modules are classified into three categories according to how the vertices are connected inside the module:

- **Series:** if $G[M]$ is a clique (A clique is a set of vertices connected to each other).
- **Parallel:** if $\overline{G}[M]$ is a clique.
- **Neighborhood:** Both, $G[M]$ and $\overline{G}[M]$ are connected graphs.

Figure 3.2*(a)* presents a graph and its modules. This example will be used as a running example throughout the paper. It is borrowed from [31] with a slight modification.

In [31], authors use modular decomposition to compress large graphs. They define a similarity measure between graphs using the obtained compressed graphs. They compress graphs by recursively compacting modules as illustrated in Figure 3.2. To obtain a unique representation of the graph only the modules that do not overlap with other modules are compacted.

To retain all the properties of the original graph with the obtained compact representation of the graph, adjacency information for neighborhood modules

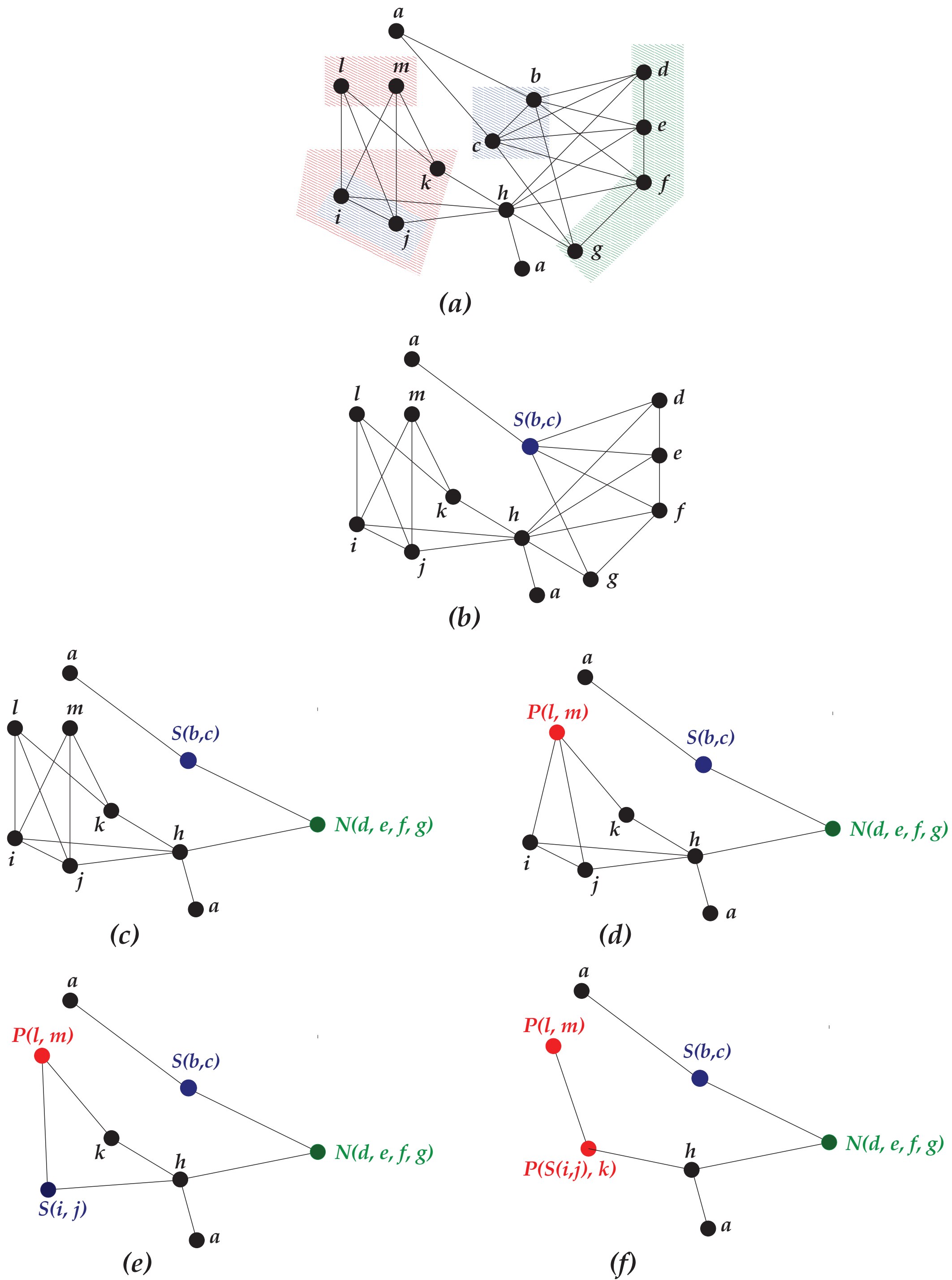


Figure 3.2: Compressing steps with modular decomposition. *S*: series module. *P*: parallel module. *N*: neighborhood module [31].

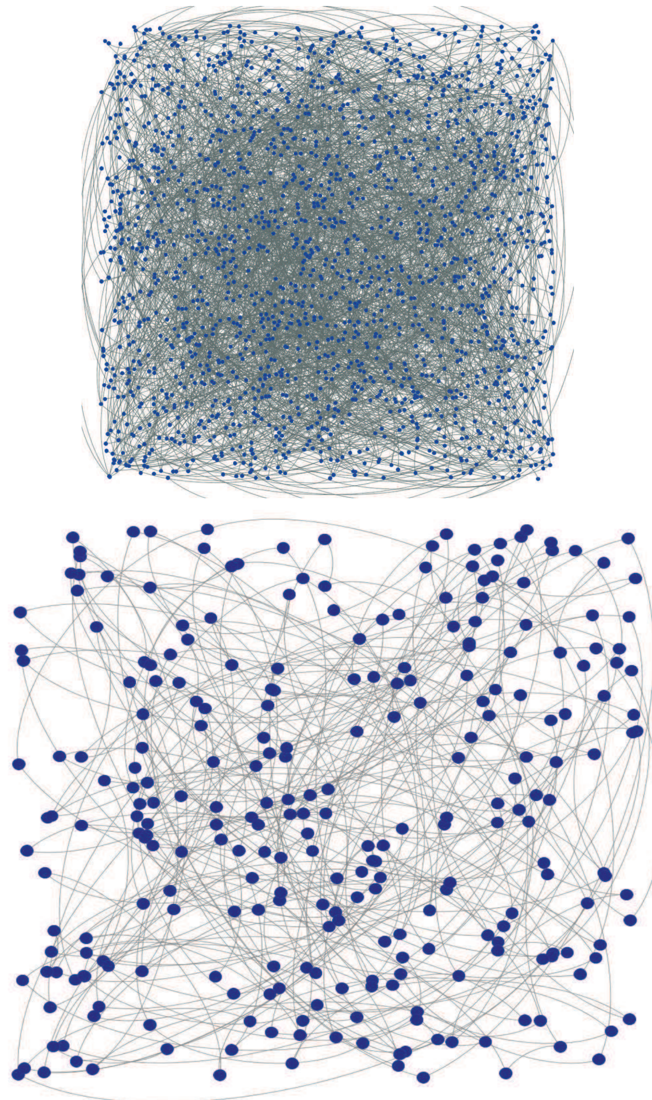


Figure 3.3: Example of a graph and its compression [44].

A protein interaction graph with 1818 vertices and 1833 edges and its compressed graph with 271 vertices and 321 edges.

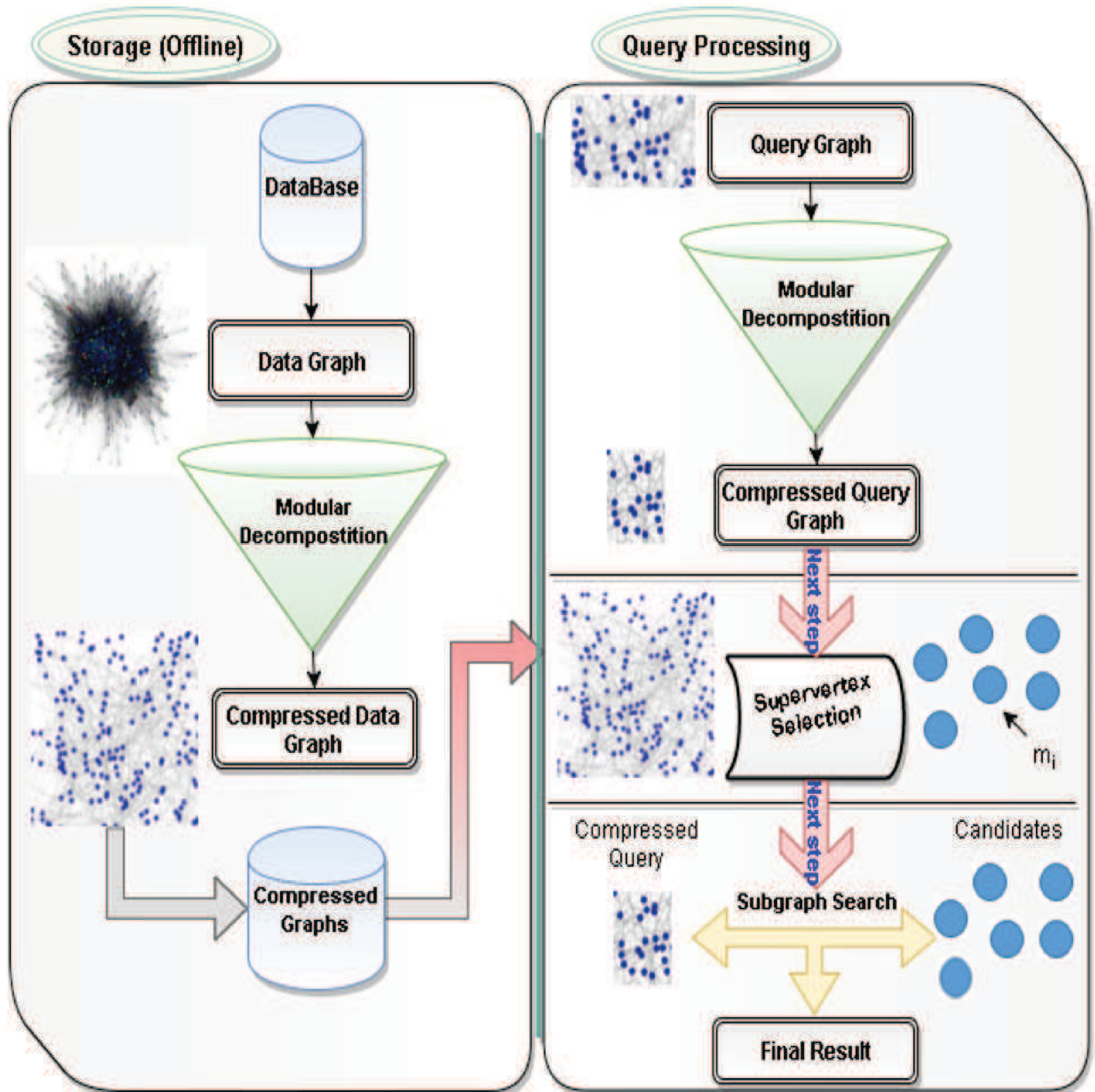


Figure 3.4: The architecture for the proposed framework.

must be stored. Series and parallel modules need no information about adjacency. For example, The obtained compressed graph illustrated in Figure 3.2*(f)* is a neighborhood module that can be denoted :

$$N(a, S(b, c), N(d, e, f, g), h, a, P(S(i, j), k), P(l, m)).$$

For this module, we retain the edges between the supervertices to keep adjacency information. This gives the final compressed graph. We also retain the edges that bind the vertices of the neighborhood module $N(d, e, f, g)$.

This compression method can allow high compression rates as illustrated in Figure 3.3 that presents a protein interaction graph and its compression obtained by modular decomposition.

A triangle listing algorithm is also proposed on graphs compressed by modular decomposition in [30]. In [43], the authors use a compression unifying Definition 7 and the concept of modules. They compact the vertices that have the same labels by distinguishing between two kinds of groups of vertices: those that are completely connected and form a clique, i.e., a series module, and those that are not connected at all,i.e., a parallel module.

In our framework, we also rely on modular decomposition to compress graphs mainly because it is a more general compression that encompasses the compressions used till now.

3.3 Compress and Search

In this section, we present our subgraph isomorphism search framework, named, Sum_{ISO}, in detail. Our framework aims to address subgraph isomorphism search in massive graph databases with a novel approach to reduce search space:

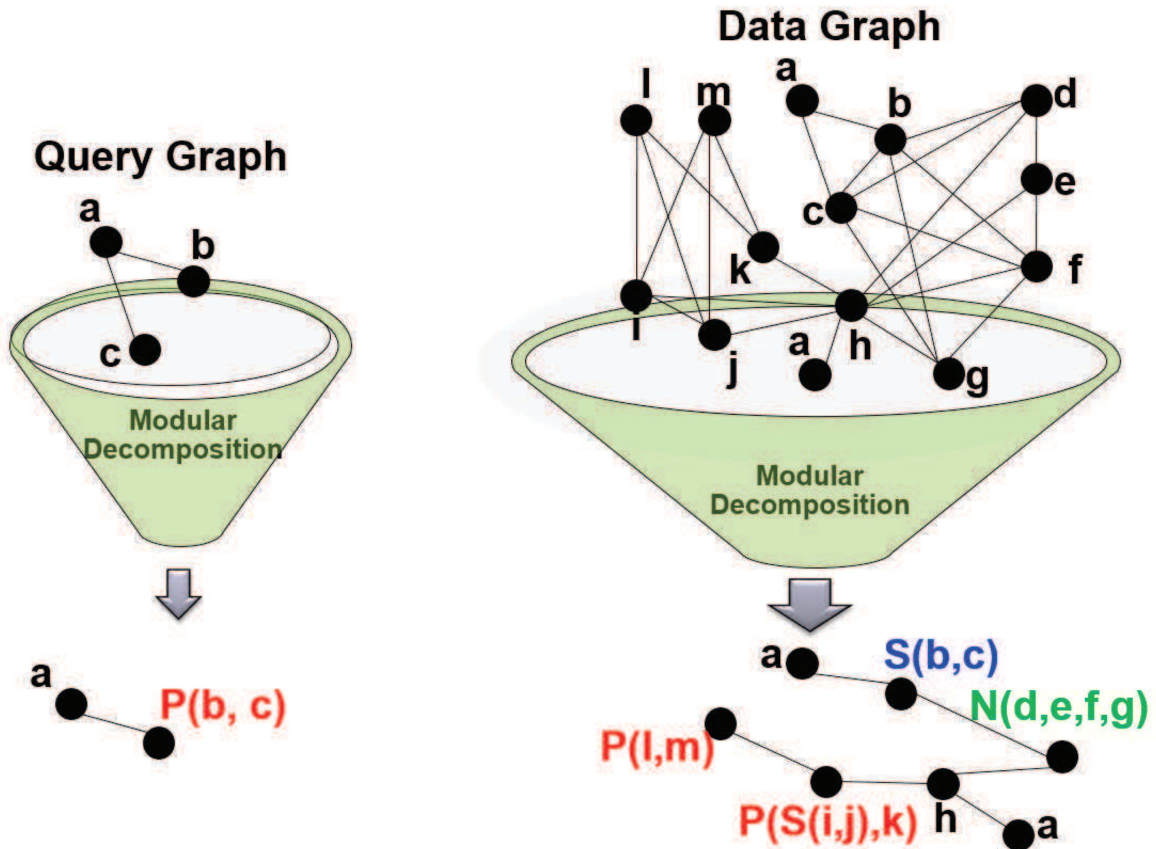


Figure 3.5: Compression Step of the Running Example.

compression.

To compress graphs, we use modular decomposition which is a well studied concept in graph theory [21]. To our knowledge, we are the first to use modular decomposition to reduce the search space in subgraph isomorphism search algorithms. As we will demonstrate, the benefits are threefold:

1. reduce the storage space needed to store graphs,
2. process the graphs without decompression, and
3. reduce the time necessary to achieve subgraph isomorphism search mainly because the search space is reduced.

Figure 3.4 illustrates the architecture of the proposed framework. Data graphs are compressed offline using modular decomposition. They are stored and processed in their compressed format. To process a query graph Q, Q is first compressed similarly to the data graph by modular decomposition. Figure 3.5 illustrates the compression step for our running example and the obtained compressed graphs (both query and data).

Query processing takes in entry the compressed versions $\mathcal{C}(Q)$ and $\mathcal{C}(G)$ of Q and G respectively and reports all the embeddings of Q in G. To avoid ambiguity, we will use the terms supervetex or module to denote a node in the compressed graph and we will denote it by m. The term vertex and leaf will denote a node from the original graph. The Algorithm operates in two phases: a candidate supervertex selection phase and a subgraph search phase. During the first phase, the compressed data graph is parsed to retain only regions of the graph that are likely to contain the query. This selection uses only the labels of the modules. During the second phase, a backtracking-like algorithm is used in each region to verify the embedding. In the following we detail both phases and show how we can find all the embedding by parsing the compressed data graph.

3.3.1 Candidate Supervertex Selection

The aim of this step is to determine the modules (supervertices) that are likely to match the query. With this step, we minimize the number of vertices of the data graph to be processed. For this, we explore the modules of $\mathcal{C}(G)$ to get all those that contain at least one of the labels of the query. Let $Cand$ denotes the obtained result with:

$$Cand = \{m \in \mathcal{C}(G) \text{ such that } \ell(m) \cap \ell(\mathcal{C}(Q)) \neq \varnothing\}.$$

After that the set of candidate modules is partitioned in several subsets where each of them is candidate for a single embedding. Each subset contains the minimum number of modules that satisfy all the labels of the query. Subgraph search is then invoked on each of these subsets. This step is illustrated in Figure 3.6 and its detailed actions are given by Algorithm 1.

Algorithm 1: Supervertex Selection.

```
Data: A summarized data graph C(G) and a summarized query C(Q).
Result: A set of candidate supervertices of C(G) that match C(Q).
begin
    Cand ← ∅;
    foreach m ∈ C(G) do
        if ℓ(Q) ∩ ℓ(m) ≠ ∅ then
            Cand ← Cand ∪ {m};
        end
    end
    C ← {s = {m_1, m_2, ⋯, m_j}|ℓ(Q) ⊆ ℓ(s)};
    foreach s ∈ C do
        P ← ∅;
        SubgraphSearch(C(Q), s, P);
    end
end
```

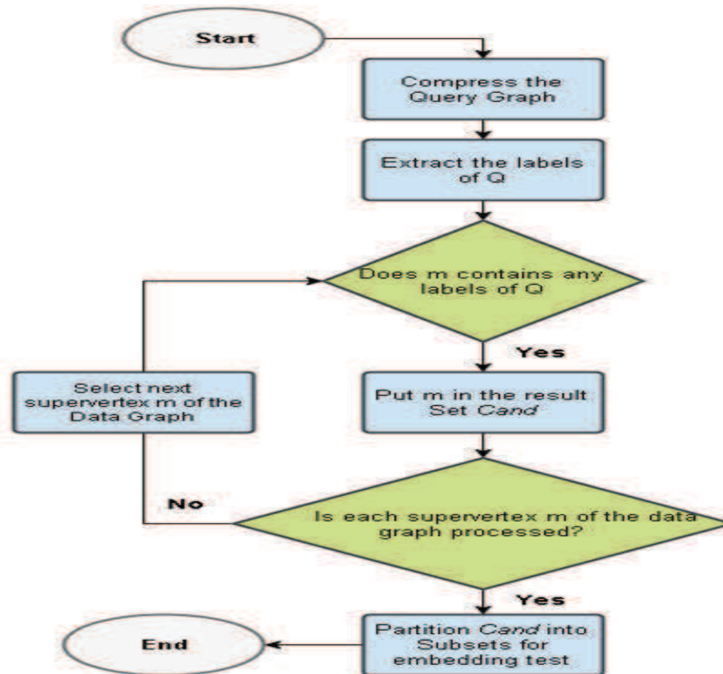


Figure 3.6: Flowchart of step 1 (Supervertex Selection).

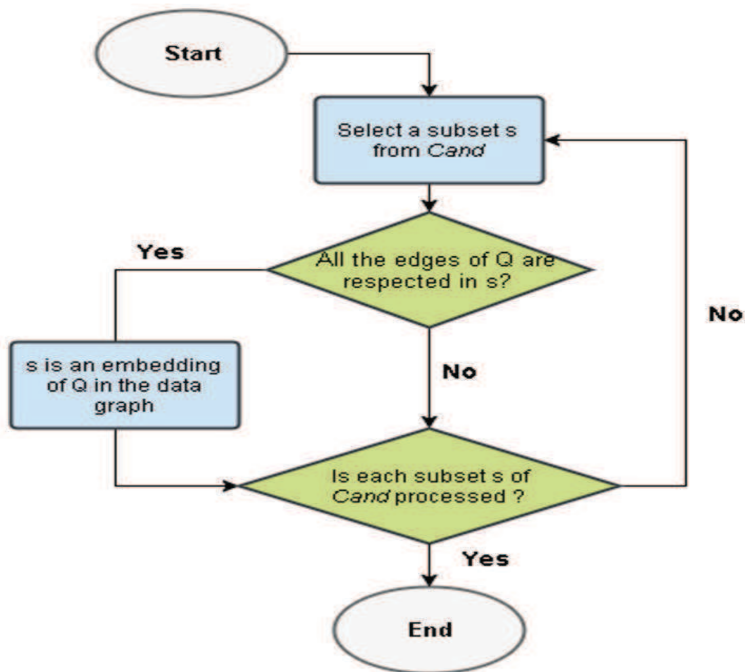


Figure 3.7: Flowchart of step 2 (Subgraph Search).

Algorithm 2: Subgraph Search.

Data: A set of modules from the data graph $s = \{m_1, m_2, \cdots, m_j\}$, the compressed query $\mathcal{C}(Q)$ and a partial embedding P.
Result: All embeddings of Q in s.

```
begin
    if |P| = |V(C(Q))| then
        Report P;
    else
        Choose a non matched vertex u from Leaves(m), m ∈ C(Q);
        C_u ← { non matched v ∈ Leaves(m_i) such that m_i ∈ s and ℓ(v) = ℓ(u) and
          IsJoinable(u, v, P)};
        foreach v ∈ C_u do
            P ← P ∪ {(u, v)};
            SubgraphSearch(C(Q), s, P);
            Remove (u, v) from P ;
        end
    end
end
```

Figure 3.8 illustrates the result of candidate supervertex selection on our running example. In this Figure, we can see that the query is compressed in a single supervertex labeled $S(P(b,c),a)$. $\ell(\mathcal{C}(Q)) = \{a,c,b\}$. The set of candidate supervertices is $Cand = \{1,2,5\}$, where 1, 2 and 5 are the identifiers of the supervertices that are candidates to match the query. The partitioning of $Cand$ yields to the subsets $\{1,2\}$ and $\{1,5\}$. This means that there are two regions in the data graph to explore for subgraph isomorphism.

Note that at this step, we have a set of candidates with no order. These candidates are selected solely on labels. No structural verification are done with the query. So, at the end of this step, we do not know if there is a subgraph in G that matches the query. The aim of the next step is to aggregate the candidate supervertices in order to verify if the structure of the query is preserved within them.

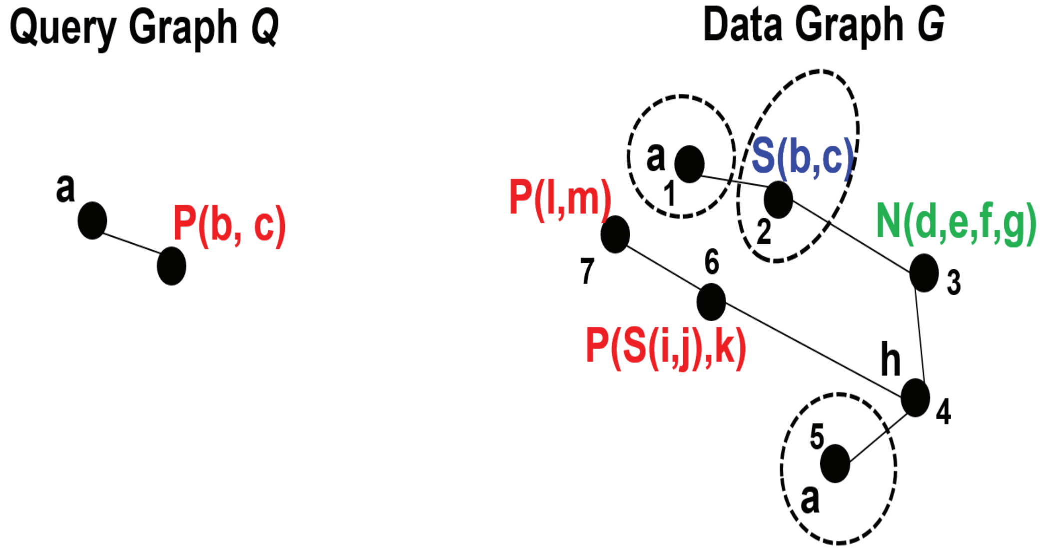


Figure 3.8: Supervertex Selection Phase on our Running Example.

3.3.2 Subgraph Search

The subgraph search phase is illustrated in Figure 3.7. Its detailed actions are given by Algorithm 2.

This step takes as inputs a query $\mathcal{C}(Q)$ and a set $s = \{m_1, m_2, \cdots, m_j\}$ of modules that are likely to contain an embedding of the query. It returns all the embeddings of the query in these modules. An embedding is represented by a set P of pairs (u, v), where u is a query vertex and v is the data vertex that matches u. For each vertex u in $\mathcal{C}(Q)$, SubgraphSearch first finds the set of candidate vertices C_u from the vertices of the modules of the set s. A vertex v of the data graph matches u if it has the same label as u and all the neighbors of u are matched to neighbors of v. This is verified by a call to function $IsJoinable$ (detailed in Algorithm 3). Given two vertices u (from the query) and v (from the data graph) to be matched, function $IsJoinable$ returns TRUE if the neighbors of vertex u are matched to neighbors of vertex v in the match P. To have the list

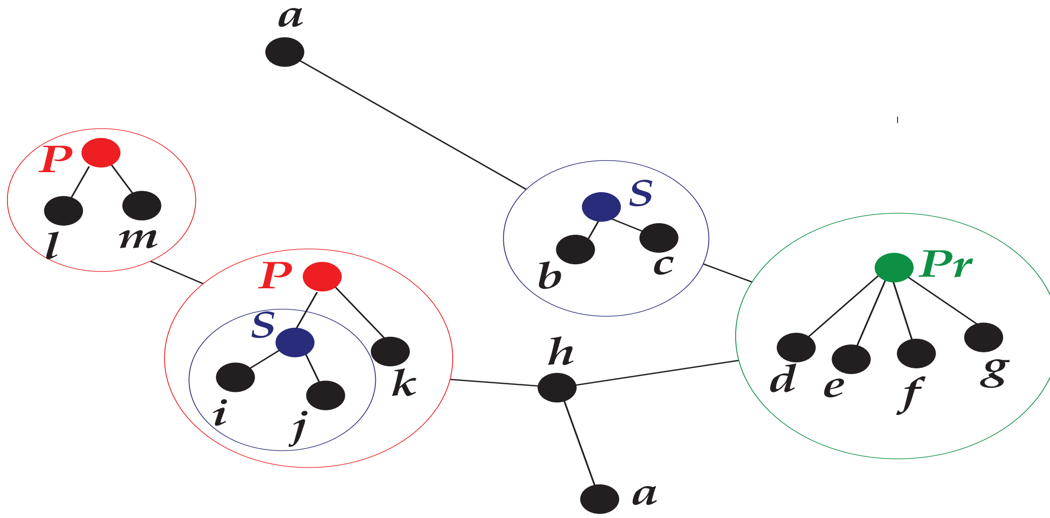


Figure 3.9: Tree representation of Modules [31].

of neighbors of a vertex in a compressed graph, we use function $Neighbors$ that takes advantage from the tree structure of the compressed graph to easily list the neighbors of a vertex as detailed in Algorithm 4. In fact, each module is a tree whose leaves are the vertices of the original graph as illustrated in Figure 3.9 for the modules of our example.

Given a vertex v, we denote by $Father(v)$ the module that contains it and by $root(\mathcal{C}(G))$ the module corresponding to the compressed graph. Given a module m, $Leaves(m)$ gives the leaves, i.e., vertices of the module m. Also, we will use $\ell(x)$ to denote the set of labels of a module or vertex x. According to the type (series, parallel or neighborhood) of the module that contains the vertex we can easily determine its neighbors. Algorithm 4 parses the subtree of $\mathcal{C}(G)$ that contains u from the father of u upward to the root of $\mathcal{C}(G)$. If a visited vertex x is a series module, then all the leaves of its descendants that are not in the branch that contains u are neighbors of u. If the visited vertex is a neighborhood module, neighbors of u are determined according to the edges of the module.

When a match (u, v) is verified, in procedure SubgraphSearch, it is reported

in P. As in any backtracking-based algorithm, SubgraphSearch uses recursion to complete the partial match until it meets the query. When a match fails, the procedure backtracks to the preceding state by removing the match. For our running example, only the region $\{1,2\}$ contains an embedding of the query. Region $\{1,5\}$ is dropped by the matching algorithm as illustrated in Figure 3.10.

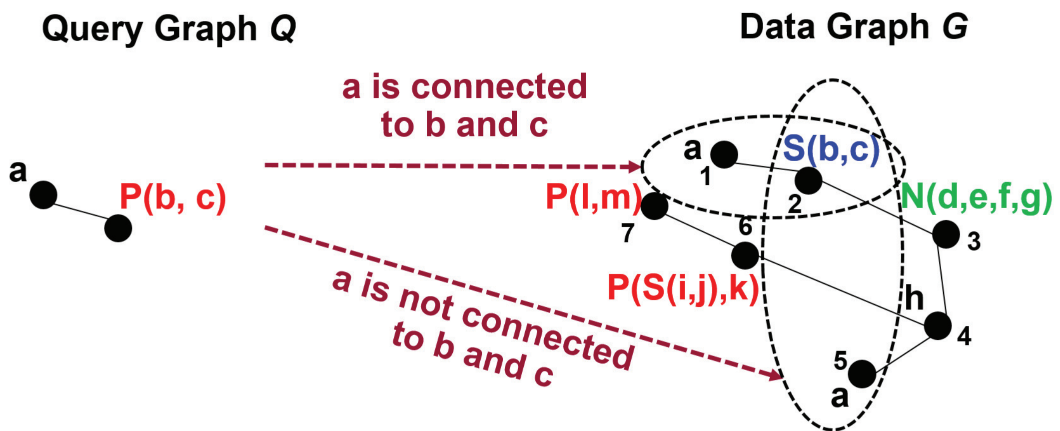


Figure 3.10: Subgraph Search Phase of the Running Example.

Algorithm 3: Verify that two vertices to be matched have the same adjacency ($IsJoinable$).

Data: Two vertices u and v to be matched.
Result: True is the the vertices have the same adjacency.
begin
  return ($\forall u' \in Neighbors(u)$, if u' is matched to v' then $v' \in Neignbors(v)$);
end

Algorithm 4: Computing the set of neighbors of a vertex in a compressed graph ($Neighbors$).

```
Data: A vertex u and a compressed graph C(G).
Result: The set of neighbors of u in G.
begin
    N ← ∅;
    z ← u;
    x ← Father(u);
    while x ≠ root(C(G)) do
        switch type of x do
            case a series module do
                foreach child y ≠ z of x do
                    N ← N ∪ Leaves(y);
                end
            case a Neighborhood module do
                foreach edge (z, y) ∈ x do
                    N ← N ∪ Leaves(y);
                end
        end
        z ← x;
        x ← Father(x);
    end
    return N
end
```

3.4 Performance Evaluation

In this section, we present a detailed evaluation of Sum_{ISO}. We evaluate mainly the execution time performance of Sum_{ISO} over different type of graphs and size

of queries. We also compared it with the most efficient state of the art algorithm, called Turbo_{ISO} and presented in [23]. We recall that Turbo_{ISO} is itself compared to the other existing solutions in [23] and showed to be superior to them.

We first describe the datasets used in the experiments, then we present our results.

3.4.1 Datasets

We use nine different real-world graphs to evaluate our approach. Three of the considered datasets were used in [23] to prove the superiority of Turbo_{ISO} against the other algorithms of the literature described in Chapter 2. These datasets are referred to as AIDS, NASA, and HUMAN. The AIDS and HUMAN datasets are also available in the RI database of biochemical data[1] [9]. The six other datasets come from the Stanford Large Network Dataset Collection (http://snap.stanford.edu/) and are referred to as Patent Citation [36], Pokec [48], LiveJournal [34], Orkut [52], WebGoogle [34] and Wiki-talk [35]. These are mainly large networks corresponding to real social networks, web graphs or citation networks. For these graphs, we introduced randomly labels on the vertices and the edges. The totality of the datasets are described below:

- *AIDS database*: This dataset consists of graphs representing molecular compounds. It contains 10,000 graphs of 27 edges. The number of unique labels in AIDS is 51.
- *NASA database*: This dataset contains 36,790 trees with an average size of 32, and a number of unique labels of 117,302.
- *Human*: This dataset consists of one large graph representing a protein

[1] http://ferrolab.dmi.unict.it/ri/ri.html#description

interaction network. This graph has $4,675$ vertices and $86,282$ edges. The number of unique labels in the dataset is 90.

- *Pokec* : Pokec is a highly popular on-line social network in Slovakia that contains friendship relationships. It has been on production for more than 10 years and connects more than 1.6 million people. The Pokec dataset contains anonymized data of the whole network.

- *Patent Citation* : This is the citation network among US Patents. It is maintained by the *US National Bureau of Economic Research*. This dataset contains all the utility patents granted from January 1963 to December 1999. It includes almost 4 million patents and almost 17 millions citations.

- *LiveJournal*: is an on-line social network with almost 5 million highly active members that regularly update their contents. With LiveJournal, members have journals, individual blogs, shared blogs and also declares their friendship relations.

- *Orkut*: is a free on-line social network with more than 3 million members and more than 117 friendship connections. This network is provided by *The Online Social Networks Research Project* [1].

- *WebGoogle*: this is Google web graph. Vertices represent web pages and edges represent hyperlinks between them. It was released in 2002 by Google as a part of Google Programming Contest.

- *Wiki-Talk*: this graph is Wikipedia talk network. A vertex represents a registered user in Wikipedia. An edge connects user i to user j and means that user i has, at least once, edited a talk page of user j in order to communicate and discuss updates to various articles on Wikipedia.

Table 4.2 summarises the characteristics of the nine datasets. Besides the average number of vertices and edges of the graphs in the dataset, we also give the average compression rate of each dataset. Given a graph G and its compressed graph $\mathcal{C}(G)$, the compression rate of G is given by: $CR(G) = \frac{|E(\mathcal{C}(G)|)|}{|E(G)|} \cdot 100\%$. It compares the number of edges in $\mathcal{C}(G)$ in respect to G. We also provide the time necessary to compress each dataset.

Table 3.1: Graph Dataset Characteristics. $avg|V|$: average number of vertices. $avg|E|$: average number of edges.

Dataset	Number of graphs	$avg|V|$	$avg|E|$	$Compression$ rate	Compression time(s)
AIDS	10,000	26	27	56.8%	0.230
NASA	36,790	94	32	44.2%	0.180
HUMAN	1	4,675	86,282	61%	0.195
WEBGOOGLE	1	5,105,039	2,378,948	53.4%	6.23
WIKI-TALK	1	2,394,385	5,021,410	50.2%	4.23
PATENT CITATION	1	3,774,762	16,518,948	42%	2.1
POKEC	1	1,632,803	30,622,564	44%	3.62
LIVEJOURNAL	1	4,847,571	68,993,773	30%	9.0
ORKUT	1	3,072,441	117,185,083	57%	14.2

Graphs within the three datasets were preliminarily compressed using an extension of the algorithm proposed in [11, 22] that computes the modular decomposition of a graph in linear time. So, we compress an input graph in $O(n+m)$ time, where n is the number of vertices and m the number of edges of the graph.

To show the storage saving obtained by compressing the datasets, Table 3.2 reports the size on disk of each dataset before and after compression.

The experiments are performed on a 2.40 GHz $Intel(R)\ Core(TM)\ i5-4210U$ 64 bits laptop with 8 GB of RAM running windows 7. The algorithm is implemented in C++.

For the AIDS, NASA, and HUMAN datasets, we use the same query sets as in

Table 3.2: Size on disk.

Dataset	Size on disk	Size on disk after compression
AIDS	4.59 Mb	2.18 Mb
NASA	24 Mb	14.42 Mb
HUMAN	1.15 Mb	0.24 Mb
WEBGOOGLE	71.8 Mb	34.1 Mb
WIKI-TALK	63.3 Mb	32.3 Mb
PATENT CITATION	267 Mb	189 Mb
POKEC	404 Mb	302 Mb
LIVEJOURNAL	1 Gb	0.8 Gb
ORKUT	1.64 Gb	0.93 Gb

[23]. These queries are constructed as follows [23]:

- AIDS and NASA query sets: For each of these datasets, the authors of [23] constructed 6 query sets (Q4, Q8, Q12, Q16, Q20, Q24), each of which contains 1,000 query graphs of the same size. Additionally, each query Q_i is contained in a query Q_{i+1}. Each query is a subgraph of a graph in the dataset.

- Human Query sets: For this dataset, the authors of [23] generated three kind of queries:

  1. Subgraph queries as for the Aids and Nasa datasets. In this case, we have 10 query sets obtained by varying the number of query sizes from 1 to 10.

  2. Clique queries where the query subgraph is a complete graph. For biological datasets, such as Human, a clique Query corresponds to a protein complex [26].

  3. Path queries where the query subgraph is a path. A path query corresponds to transcriptional or signaling pathways [26].

For the large datasets, i.e., WebGoogle, Wiki-Talk, Kopec, Patent Citation, Live-Journal and Orkut, we constructed for each graph, 5 query sets (Q100, Q200, Q300, Q400, Q500). Each set Qi contains 100 query graphs of the same size i.

The time performance reported in the results is the average time computed over the sets of queries of the same size.

3.4.2 Results

Figure 3.11 shows the experimental results for AIDS. We can clearly see that the time performed by Turbo_{ISO} decreases when the query size increases. This is explained in [23] by the containment relationship among the query sets in AIDS. We can also observe the same behavior with Sum_{ISO} which achieves better than Turbo_{ISO}. In our case, this can be explained by important compression rate of AIDS that yields a small number of candidates to be considered.

Figure 3.12 shows the experimental results for NASA. For this dataset, Sum_{ISO} achieves significantly better than Turbo_{ISO} for all the queries.

Figure 3.13 shows the results of subgraph queries over the human dataset. The superiority of Sum_{ISO} over Turbo_{ISO} is clearly observable as soon as the query size is greater than 8.

Figure 3.14 shows the results of subgraph isomorphism search for path and clique queries over the Human dataset. For the clique queries, Sum_{ISO} significantly outperforms Turbo_{ISO}. This is mainly due to the fact that a clique is compressed to a single vertex in our approach. For path queries, we have also a better results than Turbo_{ISO} even if not significantly. We explain this by the

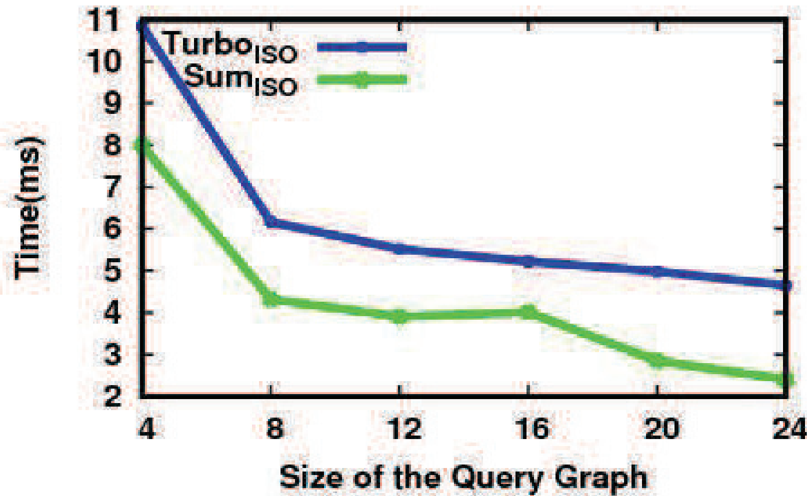


Figure 3.11: AIDS dataset.

fact that paths are not summarized by modular decomposition.

Our results on the large datasets WebGoogle, Wiki-talk, Patent Citation, LiveJournal, Pokec and Orkut, illustrated respectively on Figures 3.15, 3.16, 3.17, 3.18, 3.19 and 3.20 definitely settle the effectiveness of the proposed approach.

3.4.3 Discussion

Subgraph isomorphism search is an NP-complete problem. This implies an exponential processing time, i.e. a processing time that grows with the size of the graphs. Compression allows to reduce the size of data to save storage space. Consequently, it deals with the Volume of the data by reducing the required storage space. Volume is one of the most significant aspect of big data and

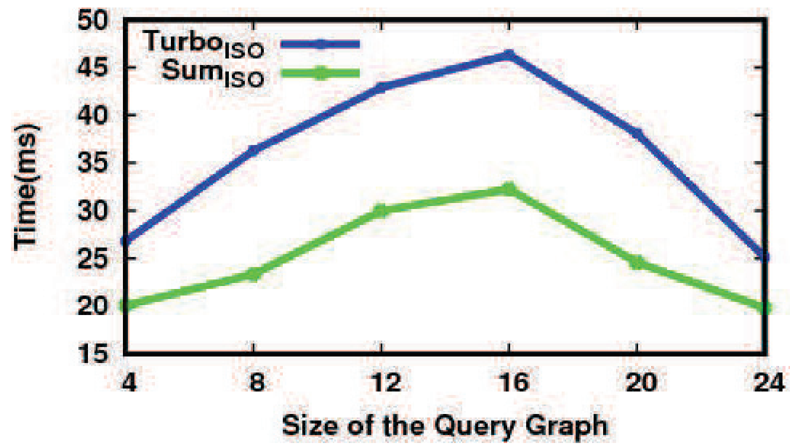


Figure 3.12: NASA dataset.

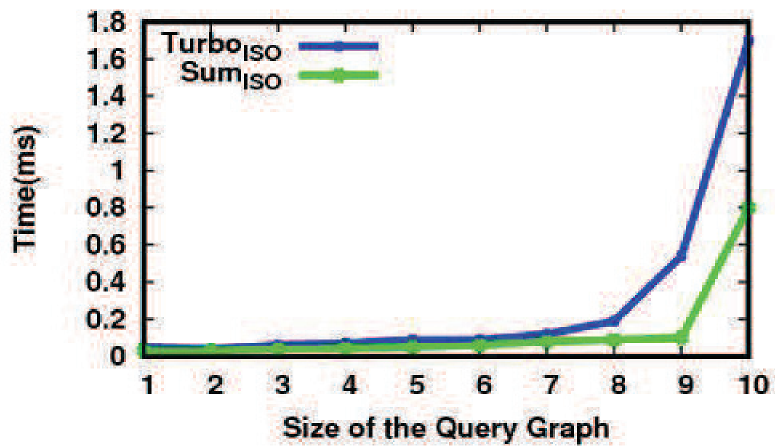
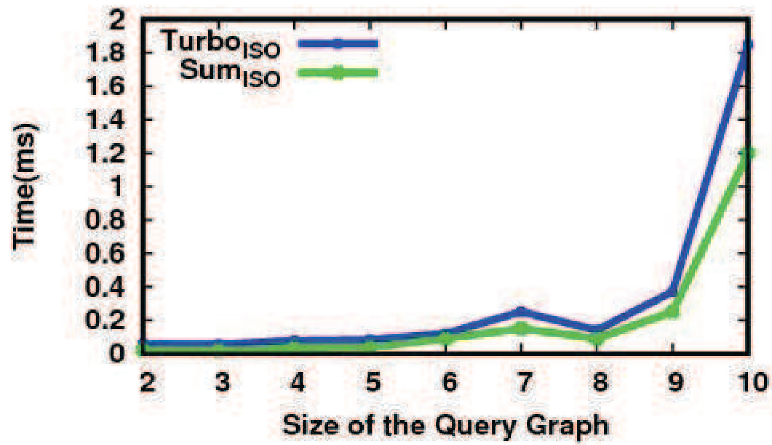
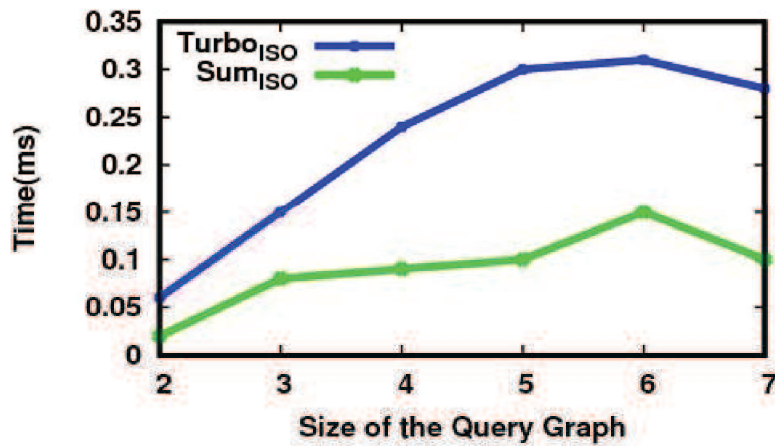


Figure 3.13: Human dataset.



(a) Path Queries.



(b) Clique Queries.

Figure 3.14: Path and Clique Queries.

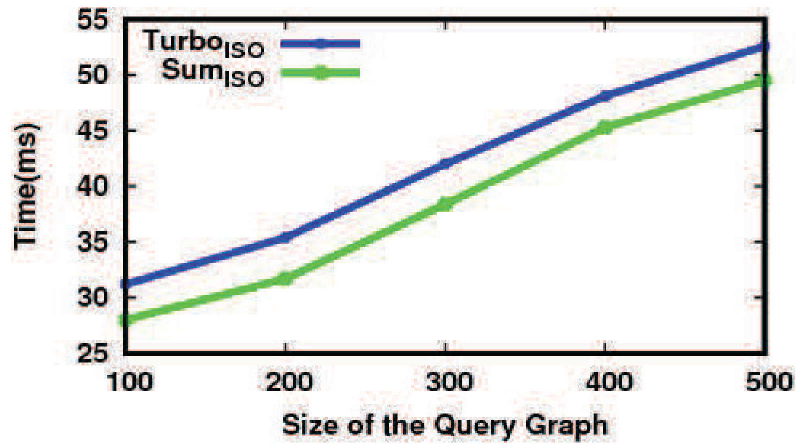


Figure 3.15: WebGoogle dataset.

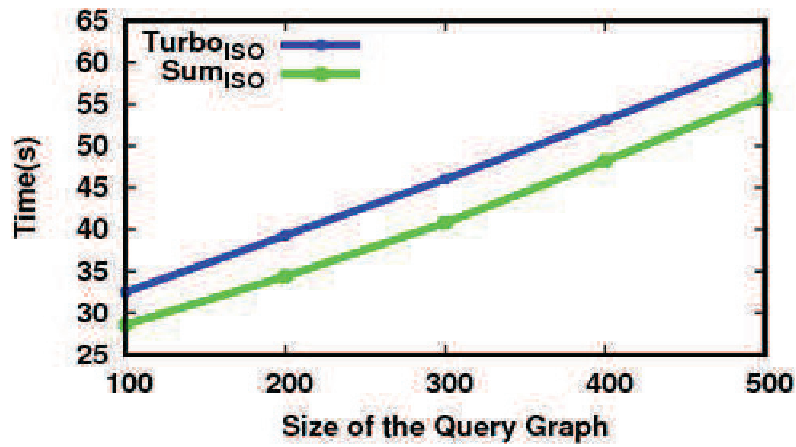


Figure 3.16: Wiki-talk dataset.

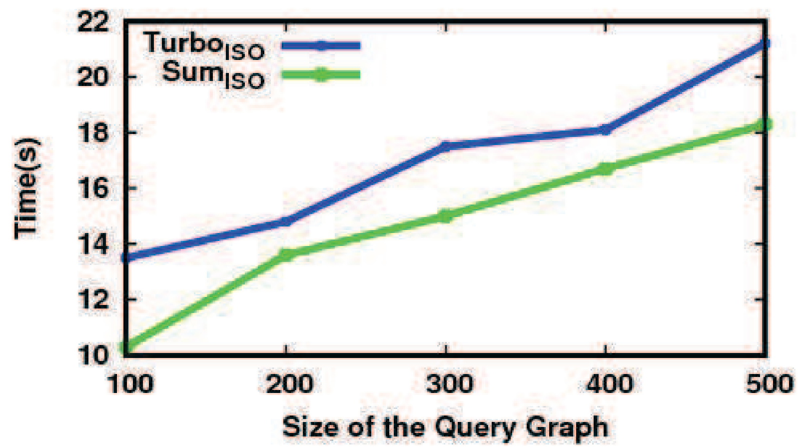


Figure 3.17: Patent Citation dataset.

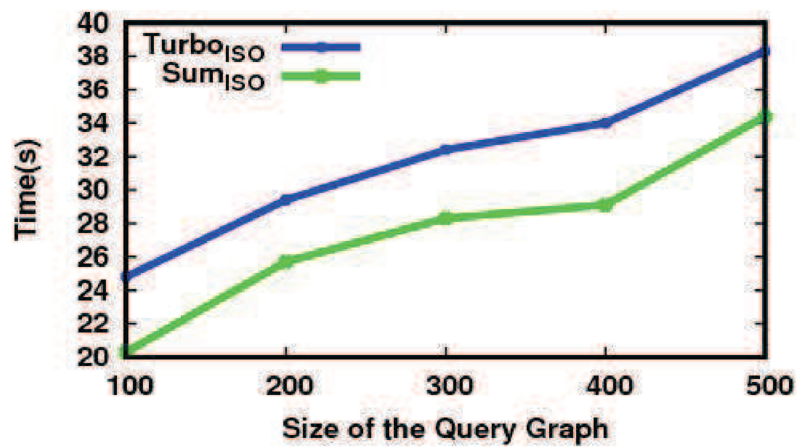


Figure 3.18: LiveJournal dataset.

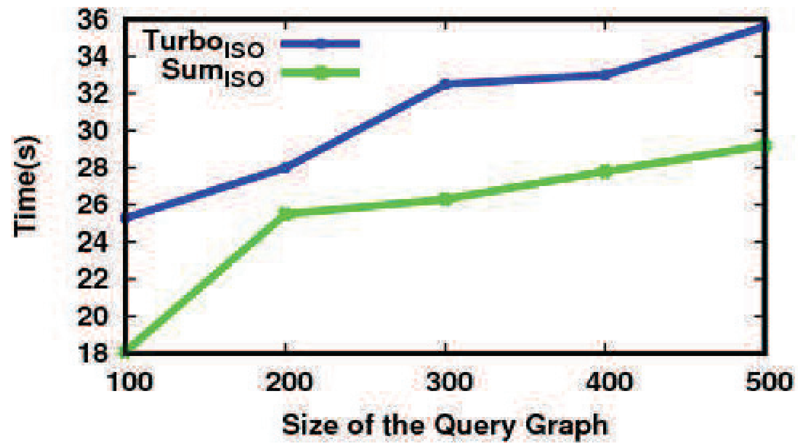


Figure 3.19: Pokec dataset.

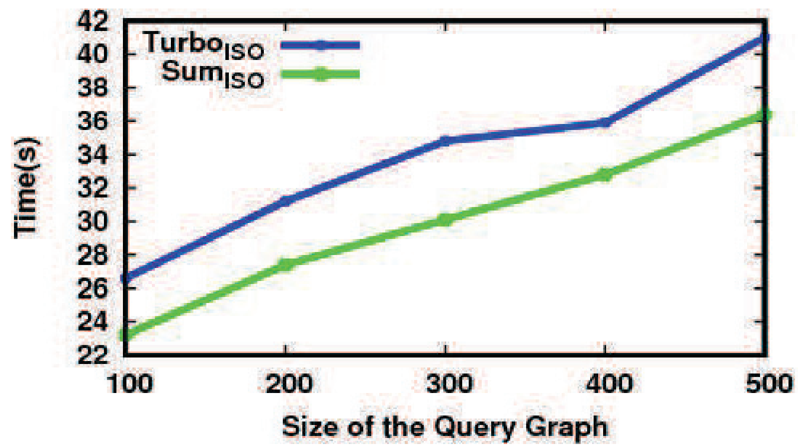


Figure 3.20: Orkut dataset.

storage is a requirement which cannot grows indefinitely: it is limited.

By compressing data, reducing time processing is not taken for granted by no means. Contrarily, compression may increase time processing as it generally requires decompressing the data before processing it. This is the main reason that motivates the choice of modular decomposition to compress the graphs. Not only we reduce the size of the graphs as we can see in Table 3.2, storage space is divided by at least a factor of two for almost all the datasets. Furthermore and most importantly, we do not decompress the graphs for processing, they are handled in their compressed form which saves time. Finally, the compressed graphs are simpler and their processing for subgraph search is lighter than processing the original graphs. This is clearly illustrated by Figures 3.11 to 3.20. However, this does not change the complexity of the subgraph isomorphism problem. It is clear that the exponential aspect of the processing time remains and is visible as soon as the size of the graphs increases. in fact, we can see on almost all the figures that plots the execution time performance a clear rise of the time at the end of the queries. This steep rise shows the exponential aspect of the time curve that gradually takes shape with the increase of the size of the query graph. The figures where no rise is observed, i.e., Figures 3.12 and 3.14, are exceptions that can be explained by a rapid punning for the last queries that decreased the search time. In fact, the queries where chosen randomly and it is difficult to foresee the behaviour of the search algorithm. Some queries even large may be simple to process because the search algorithm does not find a lot of candidate supervertices in the first step and consequently terminates rapidly. It is difficult to study this aspect of graph search algorithms because it depends on the choice of the queries. We can do that manually for very small queries but not at this scale. To meet both space and time performance with compression, the best solution is to use an inexact subgraph search algorithm [15, 31, 13].

With such methods that do not enforce an exact mapping between the query graph and the the data graph we can achieve better time performance.

3.5 Conclusion

In this chapter, we presented our first contribution, we addressed the problem of querying massive graph data in a manner that allows us to handle also the volume issue with compression. The advantage of compressing data can be huge as the volume of data and consequently the quantity of space used to store it can be massively reduced. And how about avoiding decompressing the data for query processing? This is the main contribution of our work. We presented a new approach to deal with scalability of subgraph isomorphism search in massive graph databases. In our approach, data graphs are summarized to reduce the number of vertices and edges. This reduces the search space of subgraph isomorphism search and minimizes storage requirement of the graphs. Our subgraph isomorphism search algorithm, Sum_{ISO}, finds all the embedding of a query graph in a summarized data graph without decompressing the graph. Compression is achieved by modular decomposition that generalises existing compression methods used in the literature. Our experimentations show that the proposed approach achieves good performance on both time processing of queries and space storage of data graphs.

As part of future work related to this first contribution, it is interesting to investigate how this approach can be implemented with MapReduce or an external memory framework to handle larger query graphs and also to reduce its time cost. Another extension concerns the approach itself. In fact, we have not combined existing pruning methods with compression in the Subgraph

Search phase and it may be possible to define some rules to prune the sets of candidate supervertices selected for the matching step by relying on vertex invariants, matching order and/or the properties of the compression. It will be also interesting to see if it is feasible to run such an approach on a graph database like Neo4j by designing and developing all the necessary database operations such as create, delete and insert on the compressed dataset.

Chapter 4: Compact Neighborhood Index for Subgraph Queries in Massive Graphs

In this chapter, we propose a novel approach to subgraph isomorphism search. The main idea is to distill the semantic and topological information that surround a vertex in a graph into a simple integer. This simple neighborhood encoding reduces the time complexity of vertex filtering from cubic to quadratic which is considerable for big graphs. With this encoding, we propose a very effective global filtering algorithm that is used to reduce the search space before subgraph search. The second advantage of our algorithm is that it is suitable for all graph access models: main memory, external memory, and streams by performing one sequential pass of the disk file (or the stream of edges) of the input graph. This is very usefull for graphs that do not fit into main memory. We conduct extensive experiments using synthetic and real datasets in different application domains, to compare our approach with the state of the art algorithms, and attest its effectiveness and efficiency.

Contents

4.1 Motivation 77

4.2 Our approach 84

4.2.1 Compact Neighborhood Index (CNI) 85
4.2.2 Proof of Theorem 1 87
4.3 Proof Sketch of Lemma 3 89
4.3.1 Iterative Local Global Filtering Algorithm (ILGF) 89
4.3.2 Subgraph Search 94
4.3.3 Extension to Larger Graphs 95
4.4 Experiments 97
4.4.1 Datasets 97
4.4.2 Results 101
4.5 $cni(v)$ at $(k > 1)$-hops Neighborhood 103
4.6 Conclusion 108

4.1 Motivation

We recall that subgraph isomorphism search, also known as exact Subgraph matching or Subgraph queries, is the problem of enumerating all the occurrences of a query graph within a larger graph called the data graph. Figure 4.1 shows an example of a query graph and a data graph. This example will be used throught the chapter to illustrate the algorithms and concepts.

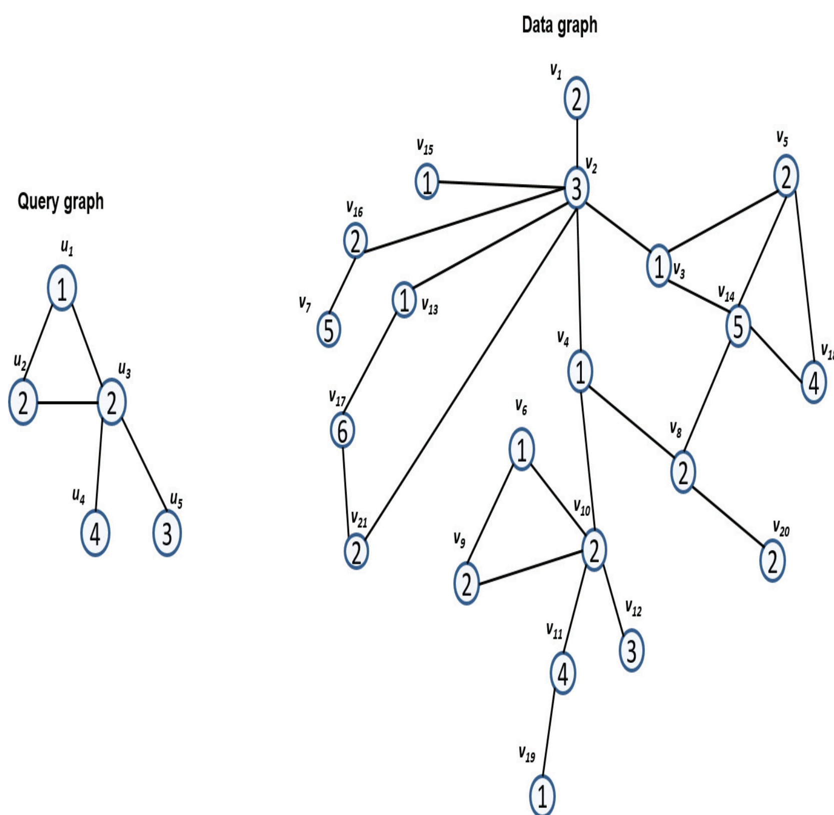


Figure 4.1: Running Example.

Most solutions to tackle this problem are based on exploiting a search space in the form of a recursion tree that maps the query vertices to the data graph vertices. However, existing algorithms never construct entirely the recursion tree and use prunning methods to have smaller search space. Filtering is fundamental

as it reduces the search space explored by the searching task. Existing algorithms differ by the pruning power of the filtering mechanisms they implement but also by when these filters take place with respect to searching. Our analysis of these two points of difference, highlighted four weaknesses in the state of the art algorithms that we address within the proposed framework. These weaknesses are as follows:

Weakness 1: *High filtering cost*. The main pruning mechanism used by existing methods during filtering are the features of the $k-$neighborhood of query vertices. This is the amount of information used when matching a query vertex with data vertices. The more information is used, i.e., k is big, the more the pruning of the search space can be important. However, representing compactly the $k-$neighborhood for practical comparisons is a challenging issue. In fact, the representation of this information has a direct impact on its cost which increases with the value of k. Besides filtering with the vertex label and the vertex degree, the lightest k-neighborhood filter is to consider the features of the one-hop neighborhood, i.e., $k = 1$. For this, recent approaches such as Turbo_{Iso} [23] and CFL-Mactch [6] use the Neighborhood Label Frequency (NLF) filter [61]. NLF ensures that a data vertex v is a candidate for a query vertex u only if the neighborhood of v includes the neighborhood of u (see lines 5-9 of Algorithm 5).

However, NLF is expensive: it is $\mathcal{O}(|V(Q)||V(G)||\mathcal{L}(Q)|)$ where $|V(Q)|$ is the number of vertices of the query , $|V(G)|$ is the number of vertices in the data graph and $\mathcal{L}(Q)$ is the set of unique labels of the query graph which is $\mathcal{O}(|V(Q)|)$ in the worst case. So, to avoid applying NLF systematically on each vertex, CFL-match [6] proposes the Maximum Neighbor-Degree (MND) filter, which can be verified in constant time for each candidate data vertex. The maximum neighbor-degree of a vertex u in a graph G, denoted $mnd_G(u)$, is the maximum degree of all its neighbors [6]. A data vertex v is not a candidate for a query

Algorithm 5: NLF and MND filters.

```
Data: A potential candidate vertex v for a query vertex u
Result: TRUE if v is candidate for u and FALSE otherwise
begin
    if mnd_G(v) < mnd_Q(u) then
        return (FALSE);
    end
    foreach label l ∈ ℓ(N(u)) do
        if |{w ∈ N(v)|ℓ(w) = l}| < |{w ∈ N(u)|ℓ(w) = l}| then
            return (FALSE);
        end
    end
    return (TRUE);
end
```

vertex u if $mnd_G(v) < mnd_Q(u)$. As MND is not as powerful as NLF, the idea is to apply it before applying NLF as detailed in Algorithm 5 (see lines 2-3). However, MND is not always effective as we can see in the example depicted in Figure 4.2 where only 3 vertices are pruned with the MND filter and consequently NLF must be applied for each of the remaining vertices.

It is also worth noting that for some neighborhood configurations filtering is useless and only the searching step is decisive. Let consider the query and data graphs depicted in Figure 4.3 where all the vertices have the same label and the same degree and let consider that $k = 1000$. Clearly, in this case, the 1000 comparisons required by NLF for each query vertex and each data vertex are needless. This doesn't mean that filtering is not necessary but that its cost must be reduced. Interestingly, using a less costly filtering with Ullmann's native subgraph searching subroutine outperforms the state of the art algorithms as showed by our experiments.

Weakness 2: Global filtering Vs local filtering. Depending on its scope, filtering can be characterised as global or local. Local filtering designates the

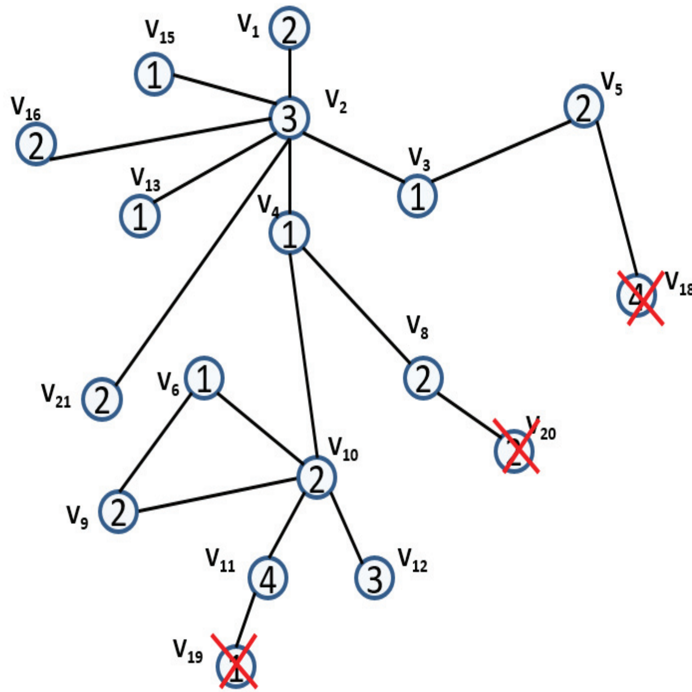


Figure 4.2: MND Filter on on the Running Example (pruned of the vertices that do not match query labels).

filtering methods that reduce the number of data vertices candidates for a given query vertex, i.e., reduce the size of $C(u_i)$, $i = 1,|V_Q|$, where $C(u_i)$ is the set of vertices of the data graph that are candidates for the query vertex u_i. Global filtering designates the filtering methods that can be applied on the entire search space, obtained by joining the above sets, i.e., $C(u_1) \times C(u_2) \times \cdots \times C(u_{|V_Q|})$. Our study of existing algorithms shows that local pruning is predominant. Some mechanisms allow global pruning but they require extra passes of the data graph to be effective. The matching order is such a mechanism. However, it is a very difficult problem to choose a robust matching order mainly because the number of all possible matching orders is exponential in the number of vertices. So, it is expensive to enumerate all of them. For example, Tuorbo$_{Iso}$ relies on vertex

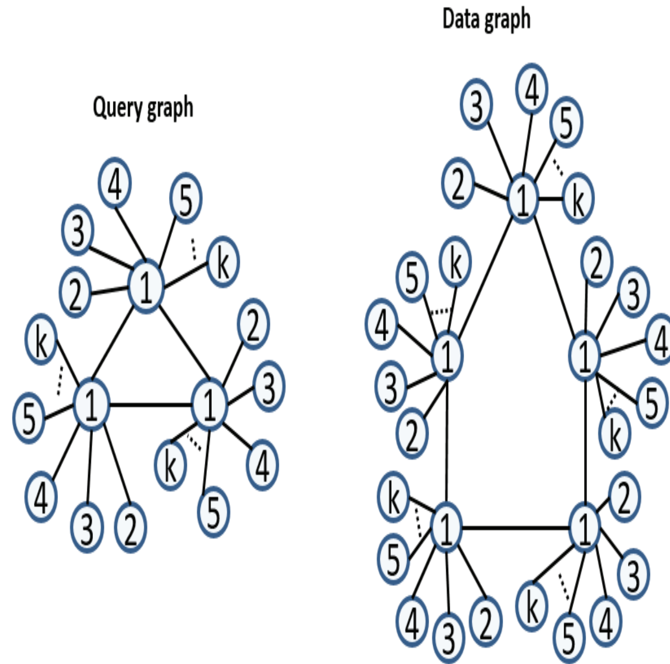


Figure 4.3: Needless NLF filtering

ordering for pruning. However, to compute this order, it needs to compute for each query vertex a selectivity criteria based on the frequency of its label in the data graph.

To deal with this problem, we introduce the Iterative Local Global Filtering mechanism (ILGF), a simple way to achieve global punning relying on local pruning filters.

Weakness 3: Late filtering. Our analysis of how filtering and searching are undertaken with respect to each other in the state of the art algorithms revealed that most algorithms apply their filtering mechanisms during subgraph search. In fact, little filtering, reduced mainly to label or degree filtering, is undertaken prior to subgraph search. This means that, the first cartesian products involved by subgraph search are costly. To tackle this, CFL-match [6] applies the MND-NLF filter prior to subgraph search. However, as we can see in Figure 4.4, the

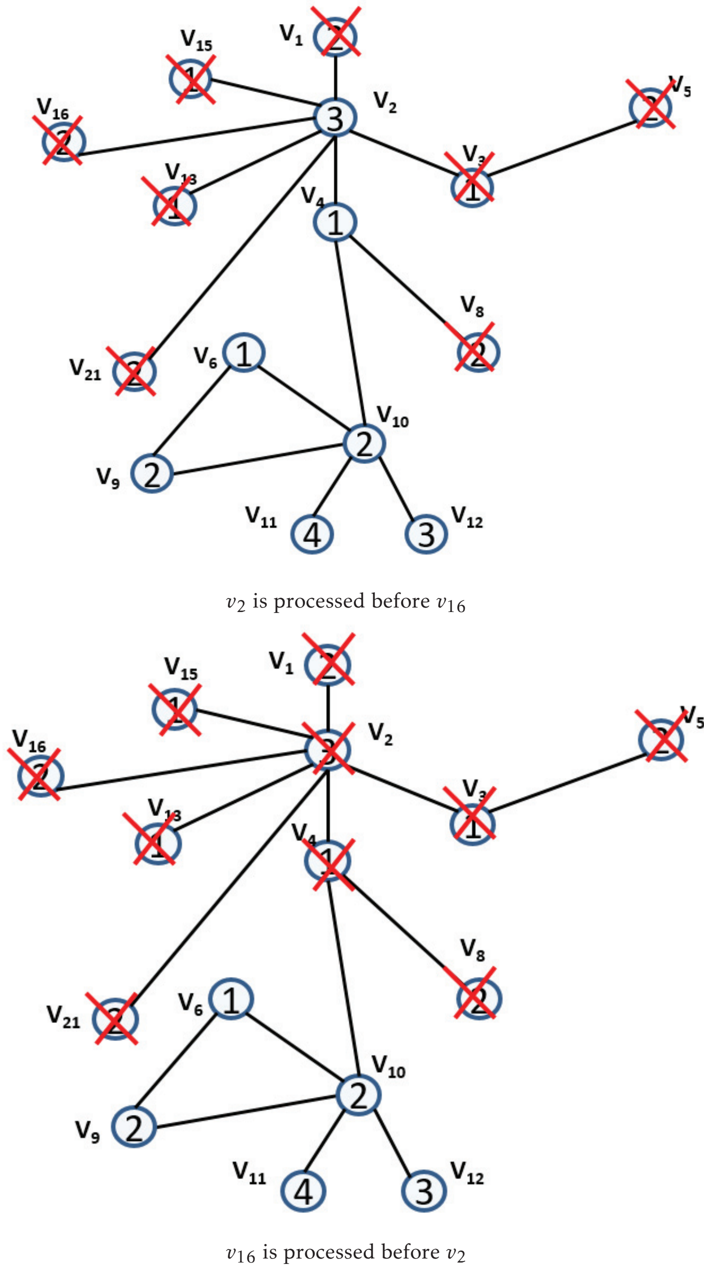
amount of achieved pruning depends on the order within which vertices are parsed. In our example, if v_2 is processed before v_{16} the amount of pruning is less than the one obtained with the reverse order. To get caught up, existing solutions rely on additional mechanisms and data structures during subgraph search such as NEC tree in Turbo_{Iso} [23] and *CPI* in CFL-mach [6] that both use path-based ordering during subgraph search. However, the underlying data structures are time and space exponential [6]. To avoid constructing and maintaining such data structures, we propose to achieve filtering solely prior to subgraph search. Our experiments show that this approach is as efficient as the state of the art algorithms.

Weakness 4: lack of scalability. This drawback results directly from the three above weaknesses. In fact, the lack of global filtering and the necessity to keep the data graph into memory for several passes make these backtracking-based solutions not suitable for graphs that do not fit into main memory. We aim to achieve a single parse of the data graph and reduce as early as possible the search space.

So, our contributions are:

- We propose a novel encoding of vertices, called Compact Neighborhood Index (CNI) that distills all the information around a vertex in a single integer leading to a simple but extremely efficient filtering scheme for processing subgraph isomorphism search. The whole filtering process is based on integer comparisons. CNIs are also easily updatable during filtering.

- We propose an Iterative Local Global filtering algorithm (ILGF) that relies on the characteristics of CNIs to ensure a global pruning of the search space before subgraph search.



v_2 is processed before v_{16}

v_{16} is processed before v_2

Figure 4.4: NLF filtering with two different vertex parsing orders

- Our encoding mechanism has the advantage to adapt to all graph access models: main memory, external memory and streams. By performing one sequential pass of the disk file (or the stream of edges) of the input graph. This avoids expensive random disk accesses if the graph does not fit into main memory.

- We conduct extensive experiments using synthetic and real datasets in different application domains to attest the effectiveness and efficiency of the proposed scheme.

4.2 Our approach

We propose a novel approach to subgraph isomorphism search that aims to reduce the cose of the filtering step. The approach is also adapted for all access methods and especially for big graphs that are accessed within a stream or in external memory. The main task of the proposed framework is a filtering step that relies on integer comparisons. This step is followed by Ullmann's matching subroutine. The efficiency of the filtering step relies on a novel method to encode a vertex. This encoding distills all the neighboring information that characterise a vertex into a single integer. Unlike existing methods that statically and invariably encode neighboring information, our vertex encoding integer can be dynamically updated leading to an iterative filtering process that allows a glabal pruning of the search space without additional data structures.

For presentation convenience, we do not show edge labels on our examples but these labels are considered in our algorithms and datasets. Table 4.1 summarises the notation used in this chapter.

Table 4.1: Notation

Symbol	**Description**
$G = (V, E, \ell, \Sigma)$	undirected vertex and edge labeled graph ℓ is a labeling function Σ is the set of labels
$V(G)$	vertex set of the graph G
$E(G)$	edge set of the graph G
$deg(v)$	degree of vertex v in G
$deg_S(v)$	number of neighbors of v that have a label in S
$G[X]$	the subgraph of G induced by the set of vertices X
$\mathcal{L}(Q)$	the set of unique labels in the query Q
$cni(v)$	compact neighborhood index of v

4.2.1 Compact Neighborhood Index (CNI)

In our method, the high-level idea is to put into a simple integer the neighborhood information that characterise a vertex. Matching two vertices is then a simple comparison between integers. Given a vertex u, the compact neighborhood index of u, denoted $cni(u)$, distills the whole structure that surrounds the vertex into a single integer. It is the result of a bijective function that is applied on the vertex's neighborhood information. This function ensures that two given vertices u and v will never have the same compact neighborhood index if they have the same number of neighbors and the same label unless they are isomorphic at one-hop. Let $x_1, x_2, x_3, \cdots, x_k$ be the list of u's neighbors' labels. The compact neighborhood index of u in the graph G is given by:
$cni(u) = \hbar(1, x_1) + \hbar(2, x_1 + x_2) + \cdots + \hbar(k, x_1 + x_2 + x_3 + \cdots + x_k)$. So, $cni(u) = \sum_{j=1}^{k} \hbar(j, x_1 + ... + x_j)$ where $\hbar(q, p) = \binom{q+p-1}{q} = \frac{(q+p-1)!}{q!(p-1)!}$

Theorem 1 states that $cni(u)$ is a bijection. Its proof is provided in Appendix

4.2.2.

Theorem 1. *$\forall(x_1, x_2, x_3, \cdots, x_k) \in \mathbb{N}^k$ and $k > 0$, g_k is a bijective function from $\mathbb{N}^k$ in $\mathbb{N}$, where:*

$$g_k(x_1, x_2, x_3, \cdots, x_k) = \sum\nolimits_{j=1}^{k} \hbar(j, x_1 + ... + x_j)$$

and

$$\hbar(q,p) = \binom{q+p-1}{q} = \frac{(q+p-1)!}{q!(p-1)!}$$

To use this bijection on vertices'labels, we need to assign a unique integer to each vertex label. This assignment can be simply achieved by numbering labels parting from 1 or by using an associative array to store the query labels. We use $ord(\ell(u))$ to retrieve the integer associated to the label of vertex u. $ord(\ell(u))$ will return 0 if vertex u has a label that does not belong to $\mathcal{L}(Q)$. This will systematically prune the neighbors that do not verify the label filter and avoid to consider them in the computation of the CNI of a vertex. Figure 4.5 illustrates the CNIs for our pruning example. In the computation of $cni(v)$ and $deg_{\mathcal{L}(Q)}(v)$, we do not consider the neighbors of the data vertex v that have not a label in $\mathcal{L}(Q)$. These vertices are illustrated in the figure with dotted lines. For example, $deg_{\mathcal{L}(Q)}(v_1 3) = 1$ because v_{17} has a label that does not belong to the query.

For filtering, We rely on three filters: the label filter, the degree filter and the CNI filter. The label and degree filters are the basis of all pruning methods. The CNI filter is based on the above bijection. So, we verify candidates for query vertices by the lemmas below.

Lemma 1 (Label filter)**.** *Given a query Q and a data graph G, a data vertex $v \in V(G)$ is not a candidate of $u \in V(Q)$ if $\ell(v) \neq \ell(u)$.*

Lemma 2 (Degree filter)**.** *Given a query Q and a data graph G, a data vertex $v \in V(G)$ is not a candidate of $u \in V(Q)$ if $deg_{\mathcal{L}(Q)}(v) < deg_{\mathcal{L}(Q)}(u)$.*

Lemma 3 (CNI filter)**.** *Given a query Q and a data graph G, a data vertex $v \in V(G)$ that verifies the label and degree filters is not a candidate of $u \in V(Q)$ if $cni(v) < cni(u)$.*

Lemmas 1 and 2 are straightforward. The proof of Lemma 3 is given in Appendix 4.3. We note also that the CNI of a vertex can also be defined to cover the k-hops neighborhood with $k > 1$.

4.2.2 Proof of Theorem 1

Proof. We need the following lemmas.

Lemma 4. $p < p' \Rightarrow \hbar(k, p) < \hbar(k, p')$

Proof. By deduction from the property of the binomial coefficient: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ (Pascal Formula) □

Lemma 5. $\forall k > 0, g_k(x_1, ..., x_k) < \hbar(k, x_1 + ... + x_k + 1)$

Proof. This inequality is trivial for $k = 1$: $g_1(x_1) = x_1$ and $\hbar(1, x_1 + 1) = x_1 + 1$. Assume that, for $k \geq 1$, the inequality holds and let us prove that it also holds for $k + 1$.
By definition of g_k, we have:
$g_{k+1}(x_1, ..., x_{k+1}) = g_k(x_1, ..., x_k) + \hbar(k+1, x_1 + ... + x_{k+1})$ $< \hbar(k, x_1 + ... + x_k + 1) + \hbar(k+1, x_1 + ... + x_{k+1})$ $< \hbar(k, x_1 + ... + x_k + x_{k+1} + 1) + \hbar(k+1, x_1 + ... + x_{k+1})$

By the property of Pascal's triangle, we know that:
$\hbar(k, x_1 + ... + x_k + x_{k+1} + 1) + \hbar(k+1, x_1 + ... + x_{k+1}) = \hbar(k+1, x_1 + ... + x_{k+1} + 1)$, we have $g_{k+1}(x_1, ..., x_{k+1}) < \hbar(k+1, x_1 + ... + x_{k+1} + 1)$

□

Lemma 6. *$\forall k > 0$, If $g_k(x_1, ..., x_k) = g_k(x'_1, ..., x'_k)$ then $x_1 + ... + x_k = x'_1 + ... + x'_k$*

Proof. Assume that $g_k(x_1, ..., x_k) = g_k(x'_1, ..., x'_k)$.
According to Lemma 5, we have:

$\hbar(k, x_1 + ... + x_k) < g_k(x_1, ..., x_k) = g_k(x'_1, ..., x'_k) < \hbar(k, x_1 + ... + x_k + 1$

we obtain then: $\hbar(k, x_1 + ... + x_k) < \hbar(k, x_1 + ... + x_k + 1$ According to Lemma 4, $\hbar(k, p)$ is strictly increasing. So, the inequality $x_1 + ... + x_k \leq x'_1 + ... + x'_k$ holds. Similarly, we prove the inverse inequality. This proves that $x_1 + ... + x_k = x'_1 + ... + x'_k$. □

To prove Theorem 1, we first prove that g_k is injective from $\mathbb{N}^k$ to $\mathbb{N}$. It is trivial for $k = 1$. In fact, $g_1 = \hbar(1, x_1) = \binom{x_1}{1} = \frac{x_1!}{1!(x_1-1)!} = x_1$ is the identity in $\mathbb{N}$. For $k \geq 2$, we assume that g_{k-1} is injective and we prove that g_k is also injective. Let $(x_1, ..., x_k)$ and $(x'_1, ..., x'_k)$ such that $g_k(x_1, ..., x_k) = g_k(x'_1, ..., x'_k)$. According to Lemma 6, $x_1 + ... + x_k = x'_1 + ... + x'_k$. We have also by definition of g_k:

$$\begin{cases} g_k(x_1, ..., x_k) = g_{k-1}(x_1, ..., x_{k-1}) + \hbar(k, x_1 + ... + x_k) \\ g_k(x'_1, ..., x'_k) = g_{k-1}(x'_1, ..., x'_{k-1}) + \hbar(k', x_1 + ... + x'_k) \end{cases}$$

By subtracting side by side, we obtain $g_{k-1}(x_1, ..., x_{k-1}) = g_{k-1}(x'_1, ..., x'_{k-1})$ which is our induction hypothesis that gives $(x_1, ..., x_{k-1}) = (x'_1, ..., x'_{k-1})$. This implies that $x_k = x'_k$.

Conclusion: g_k is injective.

To show that g_k is also surjective, we recall that $\hbar(k, x_1 + ... + x_k) \leq g_k(x_1, ..., x_k) < \hbar(k, x_1 + ... + x_k + 1)$. As $p \to \hbar(k, p)$ is a strictly increasing sequence, we deduce that each n in $\mathbb{N}$ have an antecedent in $\mathbb{N}^k$.
So, g_k is a bijection from $\mathbb{N}^k$ to $\mathbb{N}$ which proves Theorem 1. □

4.3 Proof Sketch of Lemma 3

We prove the lemma by contradiction. Assume v is a candidate of u with $cni(v) < cni(u)$. That is, there is an embedding M that maps u to v. This means that $\ell(v) = \ell(u)$ and $deg(v) \geq deg(u)$ and $\ell(N(u)) \subseteq \ell(N(v))$. Let $deg(u) = k$ and $deg(v) = k + t$, $t \geq 1$. Let $(l_1, l_2, \cdots, l_k)$ be the labels of the neighbors of u according to the order given by function $ord()$. Similarly, let $(l_1, l_2, \cdots, l_k, l_{k+1}, \cdots, l_{k+t})$ be the labels of the neighbors of v. By construction of, we have $cni(v) = g_{k+t}(l_1, l_2, \cdots, l_{k+t})$=$g_k(l_1, l_2, \cdots, l_k)$+$\hbar(k+1, l_1 + ... + l_{k+1})$+$\cdots$+$\hbar(k+t, l_1 + ... + l_{k+t})$. So, $cni(v) = cni(u)$+$\hbar(k+1, l_1 + ... + l_{k+1})$+$\cdots$+$\hbar(k+t, l_1 + ... + l_{k+t})$. as $t > 0$, we reach a contradiction. Thus, the lemma holds.

Note that, the CNI filter can be verified in constant time; that is, verifying one candidate vertex v for a query vertex u takes $\mathcal{O}(1)$ time versus $\mathcal{O}(\mathcal{L}(Q))$ for NLF.

4.3.1 Iterative Local Global Filtering Algorithm (ILGF)

The aim of the Iterative Local Global Filtering Algorithm (ILGF) is to reduce globally the search space using CNIs. It relies on the fact that $cni(v)$ can be easily updated after a local filtering giving rise to new filtering opportunities. Algorithm 6 details this iterative filtering process. To verify the CNI filter on a candidate data vertex, the algorithm uses the *cniMatch()* subroutine that implements Lemma 3 and consequently allows to verify that a data vertex is a candidate for a given query vertex according to the CNI filter. The ILGF algorithm removes iteratively from G the vertices that do not match a query vertex using the label, the degree and the CNI filters (see lines 5-7) of the algorithm. Each time a vertex is removed by the filtering process the degree and CNI of its neighbors are updated (lines 8-10) giving rise to new filtering

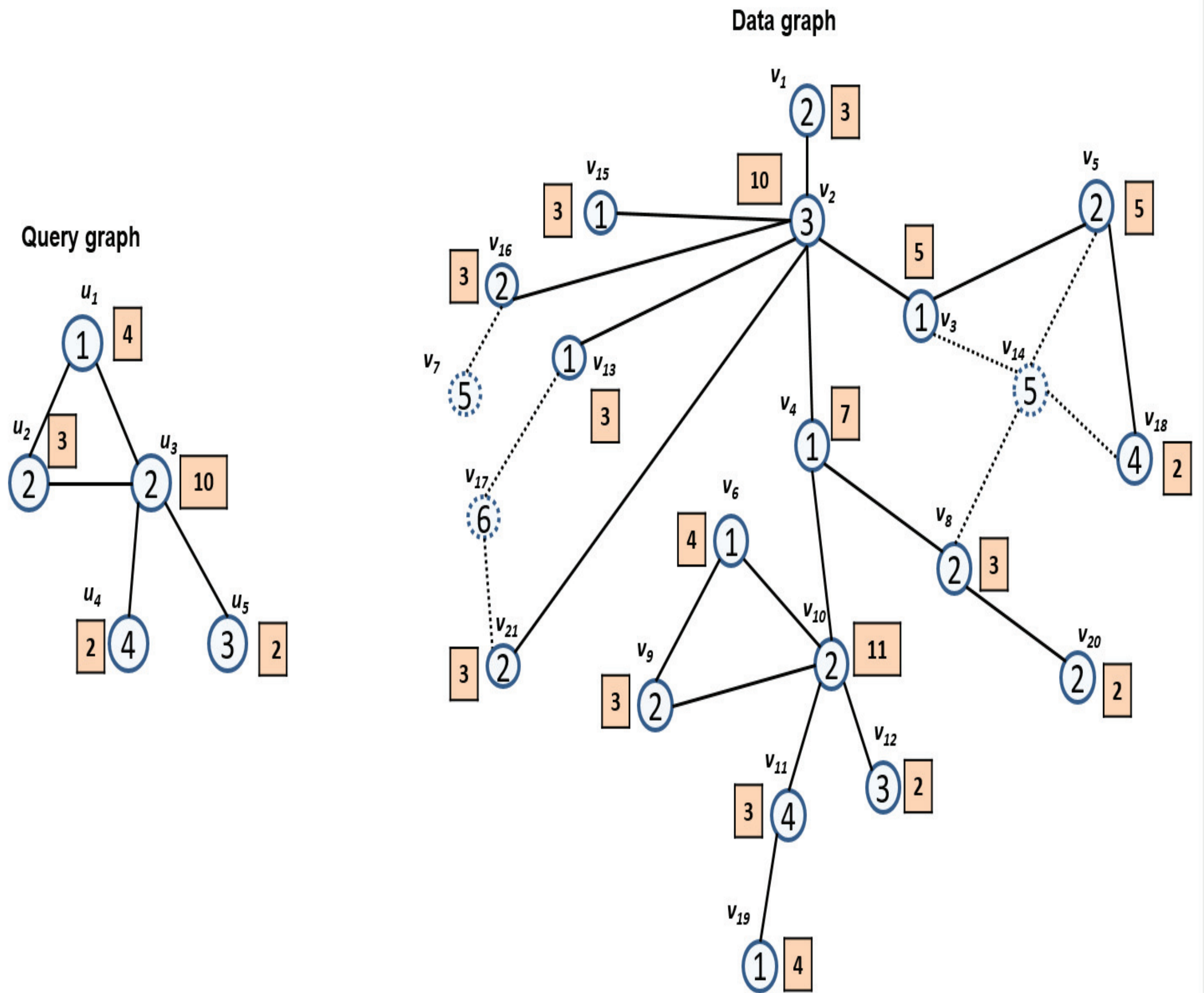


Figure 4.5: CNIs of the Query graph and the Data graph.
Vertices (and the corresponding edges) in dotted lines are not considered in the computation of $deg_{\mathcal{L}(Q)}(u)$ and $cni(u)$.

opportunities. Filtering stops when no further vertices are removed. This is implemented by the boolean variable *stopFilter*. This iterative filtering leads to an early global filtering of the search space.

```
Algorithm 6: ILGF.
Data: A data graph G
Result: A filtered version of G
begin
    stopFilter ← FALSE;
    cpt ← |V(G)|;
    repeat
        foreach vertex v ∈ V(G) do
            if ∀u ∈ V(Q), !cniMatch(v, u) then
                remove v from V(G) and the corresponding edges from E(G);
                foreach x ∈ N(v) do
                    update cni(x);
                end
            end
            else
                cpt;
            end
        end
        if cpt=0 then
            stopFilter ← TRUE;
        end
    until stopFilter;
    foreach vertex u ∈ V(Q) do
        C(u) ← {v ∈ V(G) such that cniMatch(v, u)};
        if C(u) = ∅ then
            return (∅);
        end
    end
    M ← ∅;
    SubgraphSearch(M);
end
```

Figure 4.6 illustrates the ILGF algorithm on our example. We can see in these

Algorithm 7: Function *cniMatch*(v,u).

Data: A data vertex v and a query vertex u.
Result: returns true if v is a candidate for u according to the label, degree and CNI filters.
begin
  return $(\ell(v) = \ell(u) \bigwedge deg_{\mathcal{L}(Q)}(v) < deg_{\mathcal{L}(Q)}(u) \bigwedge cni(v) < cni(u))$ or $(\ell(v) = \ell(u) \bigwedge deg_{\mathcal{L}(Q)}(v) = deg_{\mathcal{L}(Q)}(u) \bigwedge cni(v) = cni(u)))$
end

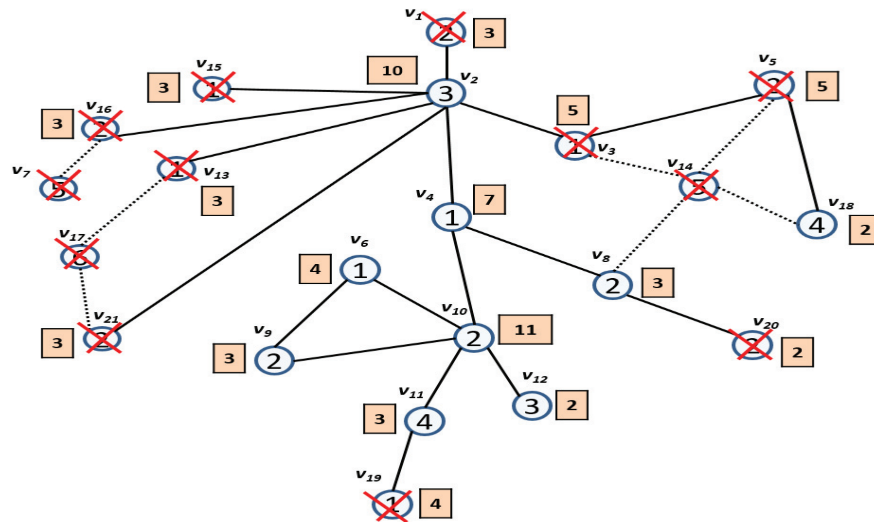
figures that using our three filters we have the following possible mappings betwen data vertices and query vertices:

- u_1 has candidates v_4 and v_6,
- u_2 has candidates v_8, v_9 and v_{10},
- u_3 has candidate v_{10},
- u_4 has candidates v_{11}, and
- u_5 has candidates v_2 and v_{12}.

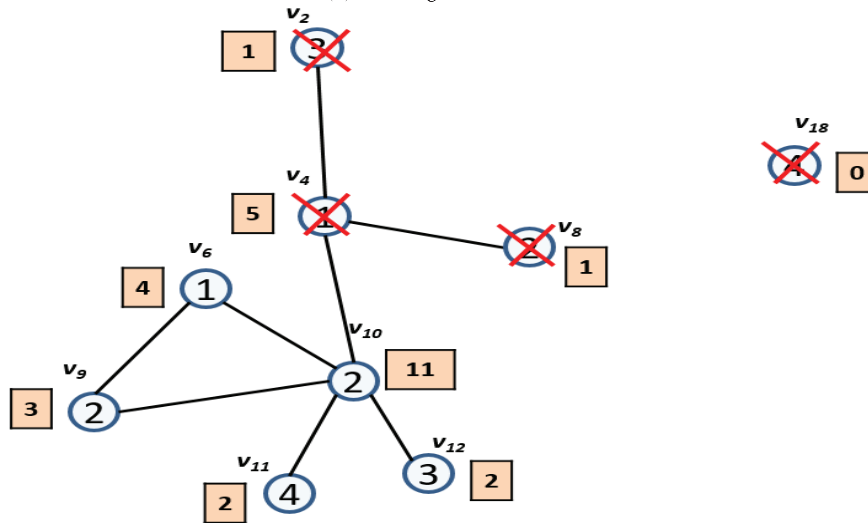
In fact, the first iteration of the ILGF algorithm, finds out that vertices v_1, v_3, v_5, v_7, v_{13}, v_{14}, v_{15}, v_{16}, v_{17}, v_{19}, v_{20} and v_{21} cannot be mapped to any query vertex because:

- v_7, v_{14} and v_{15} do not pass the label filter.
- v_1, v_{13}, v_{15}, v_{16}, v_{19}, v_{20} and v_{21} do not pass the degree filter.
- v_3 and v_5 do not pass the CNI filter.

After removing these vertices and updating the degree and CNI of their neighbors a new filtering iteration is triggered (see Figure 4.6 (b). This second filtering iteration reveals that vertices v_2, v_4, v_8 and v_{18} can also be pruned. In fact, v_2



(a) Filtering iteration 1



(b) Filtering iteration 2

Figure 4.6: Filtering iterations of our running example.

and v_4 do not pass the CNI filter and v_8 and v_{18} do not pass the degree filter. The final filtered graph is illustrated in Figure 4.6 (b).

4.3.2 Subgraph Search

After filtering, the data graph contains only the vertices that are candidates for query vertices, i.e., the vertices map at one-hop according to the CNI filter. Subgraph search allows to verify the mapping at k-hops. Algorithm 8 implements this step. It is the depth first search subroutine of Ullmann's algorithm. It lists the subgraphs of the filtered data graph that are isomorphic to the query by verifying the adjacency relationships. This step allows also to handle edge labels by discarding those that do not match the query labels. The subroutine *neighborCheck()* verifies that a mapping (v, u) is added to the current partial embedding M only if v and u have neighbors that also map.

Algorithm 8: SubgraphSearch.

```
Data: a partial embedding M.
Result: All embeddings of Q in G.
begin
    if |M| = |V(Q)| then
        Report M;
    end
    Choose a non matched vertex u from V(Q);
    C(u) ← { non matched v ∈ V(G) such that cniMatch(v, u));
    foreach v ∈ C(u) do
        if neighborCheck(u,v, M) then
            M ← M ∪ {(u, v)};
            SubgraphMatch(M);
            Remove (u, v) from M ;
        end
    end
end
```

Algorithm 9: Function *neighborCheck*(u, v, M).

Data: a partial embedding M, a query vertex u and a data vertex v.
Result: returns true if u and v have neighbors that match.
begin
  return
  $\forall (u', v') \in M, ((u, u') \in E(Q) \rightarrow (v, v') \in E(G) \bigwedge \ell((u, u')) = \ell((v, v'))$
end

4.3.3 Extension to Larger Graphs

For large data graphs, we aim to keep in memory as few vertices and edges as the three filters can achieve. So, filtering begins while reading the data graph. For this, we compute vertex degrees and CNIs incrementally during graph parsing. Only a single pass of the graph is needed. This is important if we deal with a graph stream or a sequential read of a graph from disk, i.e., a graph that does not fit into main memory and that is loaded part by part. We keep in memory only the vertices (and the corresponding edges) that verify the label, degree and CNI filters. These are the vertices and edges that will be used during subgraph search. As we parse the data graph, the label filter is straightforward. However, the degree and the CNI can be used when their values, computed incrementally, are sufficient for pruning. However, this depends on how the stream of edges arrives. If edges are sorted, i.e., we access all the edges involving vertex i, then all the edges involving vertex $i+1$ and so on, the amount of pruning will be larger during the parse than in the case edges arrive randomly.

Algorithm 10 presents the filtering actions performed during the data graph reading in the case where edges are sorted. In this case, the three filters can be applied as the edges of a vertex are accessed avoiding to store them. When all the edges incident to the current vertex are available (see lines 14-20), we can compute the CNI of the current data vertex and compare it with the CNIs

of the query vertices (see lines 21-25), the vertex and all its edges are pruned. The filtered data graph, denoted G_Q obtained at the end of the reading-filtering process contains only the data vertices that are candidates to query vertices.

Algorithm 10: Large Data Graph Filtering .

```
Data: A Data Graph G (stream of edges).
Result: A filtered data graph G_Q.
begin
    //processing a stream of sorted edges
    V(G_Q) ← ∅;
    E(G_Q) ← ∅;
    read edge (x, y);
    repeat
        current ← −1;
        if x ∉ V(G_Q) and ℓ(x) ∈ ℒ(Q) then
            V(G_Q) ← V(G_Q) ∪ {x};
        end
        if x ∈ V(G_Q) then
            current ← x;
        end
        while x = current do
            if ℓ(y) ∈ ℒ(Q) and y ∉ V(G_Q) then
                V(G_Q) ← V(G_Q) ∪ {y};
                E(G_Q) ← E(G_Q) ∪ {(x, y)};
            end
            read edge (x, y);
        end
        compute cni(current);
        if ∀u ∈ V(Q), !cniMatch(current, u) then
            remove current from V(G_Q);
            remove all the edges of current from E(G_Q);
        end
    until end of stream;
end
```

4.4 Experiments

We evaluate the performance of our algorithm, *CNI* (for Compact Neighborhood Index), over various types of graphs, sizes of queries, number of labels and their distribution on vertices. We also compare it with three state of the art algorithms, CFL-match [6], Turbo_{ISO} [23] and Sum_{ISO} [40] a representative algorithm for compressed based subgraph search approaches developed in [31], [43], and [40]. Note that each of CFL-match, Turbo_{ISO} and Sum_{ISO} are compared to the other existing solutions, such as QuickSI and SPath, and showed to be more efficient in [23, 43, 6, 40].

All experiments are performed on an *Intel i*5 2.40 GHz, 64 bits laptop with 8 GB of RAM running windows 7. Algorithms are implemented in C++. For the state of the art algorithms, we used the binaries provided by the authors.

We first describe the datasets used in the experiments, then we present our results.

4.4.1 Datasets

We use seven datasets of real-world graphs to undertake the experiments. We also used synthetic graphs to evaluate the scalability of the algorithms. These datasets can be classified into three categories:

1. Small graphs: these graphs are known datasets used by almost all existing methods in their evaluation process. So, we mainly use them as comparative datasets. The underlying graphs represent protein interaction networks coming from three main organisms: human (HUMAN and HPRD datasets), yeast (YEAST dataset) and fish (DANIO-RERIO dataset). The HUMAN and DANIO-RERIO datasets are available in the RI database of biochemical

data[1] [9]. The HPRD and YEAST come from the work of [33] and [6].

- *HUMAN*: This dataset consists of one large graph representing a protein interaction network. This graph has 4,675 vertices and 86,282 edges.
- *HPRD*: This is a graph that contains 37,081 edges and 9,460 vertices. The number of unique labels in the dataset is 307.
- *YEAST*: This graph contains 12,519 edges, 3,112 vertices, and 71 distinct labels.
- *DANIO-RERIO*: This graph contains 51,464 edges and 5,720 vertices. We used it with different number of labels (32, 64, 128 and 512) and distributions of them.

To query the HUMAN, HPRD and YEAST datasets, we use the sets of queries generated in [6]. Each query is a connected subgraph of the data graph obtained using a random walk on the data graph. For HPRD an YEAST, the authors of [6] provide 8 query sets, each containing 100 query graphs of the same size. The 8 query sets are denoted $25s$, $25n$, $50s$, $50n$, $100s$, $100n$, $200s$, and $200n$, where is and in denote query sets with i vertices and, respectively, average degree ≤ 3 (i.e., sparse) and > 3 (i.e., non-sparse). For HUMAN which is the smallest graph among the considered datasets, the authors constructed smaller queries denoted $10s$, $10n$, $15s$, $15n$, $20s$, $20n$, $25s$, and $25n$.

We used the DANIO-RERIO dataset to evaluate the algorithms in function of the number of labels and their distribution on the vertices. So, we used this dataset with 4 different number of unique labels 32, 64, 128, and 512

[1] http://ferrolab.dmi.unict.it/ri/ri.html#description

provided by the *RI* database [9]. We use 2 distributions of the labels on the vertices: a uniform distribution and a Gaussian distribution (normal distribution). The obtained graphs are denoted $32u$, $64u$, $128u$, $512u$, $32g$, $64g$, $128g$ and $512g$ where iu and ig denote a DANIO-RERIO data graph with i distinct labels and respectively a uniform distribution and a normal distribution of labels. For all these graphs, we use two sets of queries sparse queries and non sparse queries with the same number of vertices: 128.

2. Large graphs: In this category, we considered a real graph from the Stanford Large Network Dataset Collection [2] called LiveJournal. It is a graph representing an on-line social network with almost 5 million members and over 68 million friendship relations, i.e., edges. We used 200 distinct labels and 4 sets of queries with $100k$, $200k$, $400k$ and $500k$ (with $k = 10^3$) vertices. Each set contains 10 query graphs of the same size.

3. Big Graphs (stream of edges): In this category, we considered two graphs from the Stanford Large Network Dataset Collection. They are Twitter and Friendster.

   - Twitter is a snapshot of the twitter microblogging social network that corresponds to the period of June-Dec 2009. The vertices represent users and edges correspond to user-follower relationships.
   - Friendster is an on-line social network where edges correspond to friendship relations. It contains more than 65 million vertices and more than 180 billion edges.

These graphs do not fit in the main memory of the computer used for

[2] http://snap.stanford.edu/

the experiments: Twitter is 7.5 GB and Friendster is 30GB. We used 200 unique labels with a uniform distribution with Twitter and 512 unique labels for Friendster. For these two graphs, we also constructed 4 sets of queries of $100k$, $200k$, $400k$ and $500k$ (with $k = 10^3$) vertices. Each set contains 10 query graphs of the same size. Each query graph is a connected subgraph obtained by a random walk in the data graph. We processed these big graphs as a stream of edges by partitioning each disk file into several sequential files that fit into main memory.

We also constructed 3 synthetic graphs with 5 billion, 20 billion and 70 billion vertices respectively. Edges are added following a power law distribution of the degree according to the characteristics of real big graphs. For each of these graphs, we used 512 labels distributed uniformly on the vertices. We queried these graph with a set of 10 queries of $500k$ vertices each.

Table 4.2 summarises the characteristics of the datasets. For each graph, we report the number of vertices, the number of edges, the number of unique labels and the compression rate which is the ratio between the number of edges of the compressed graph on the number of edges of the original graph using modular decomposition of graphs as a compression tool of graphs [18, 21, 31]. Modular decomposition compresses graphs by aggregating vertices that have the same neighbors into one single vertex. The compression ratio is used to show how well the datasets are compressible and consequently how well they are suitable for a subgraph isomorphism search algorithm such as Turbo$_{Iso}$, its boost version developed in [43] or Sum$_{Iso}$ [40]. For instance, we can see that the HUMAN dataset is highly compressible, i.e., compression rate of 61%.

Table 4.2: Graph Dataset Characteristics.

Dataset	$\|V\|$	$\|E\|$	Number of labels	Compression rate(%)
HUMAN	4,675	86,282	44	61
HPRD	9,460	37,081	307	25
DANIO-RERIO	5,720	51,464	32/64/128/512	25
YEAST	3,112	12,519	71	43
LIVEJOURNAL	4,847,571	68,993,773	200	30
TWITTER	17,069,982	476,553,560	200	-
FRIENDSTER	65,608,366	180,606,731,005	512	-

4.4.2 Results

In this subsection, we report and comment the results obtained by comparing our algorithm with the state of the art algorithms CFL-match [6], Turbo$_{ISO}$ [23] and Sum$_{ISO}$ [40] the representative algorithm for compression-based subgraph search approaches developed in [43], [31] and [40]. Our main metric is the time performance by varying $|V(Q)|$, i.e., the number of vertices in the query, $|\Sigma|$, i.e., the number of unique labels, the sparsity of the queries, the distribution of labels, and $|V(G)|$, i.e., the number of vertices in the data graph. We present the obtained results according to this metrics and by category of graphs (small, large and big). We note also that all the algorithms output the same sets of isomorphic subgraphs for each query graph.

Against Existing Algorithms by Varying $|V(Q)|$ ***within the small datasets***: Figure 4.7 shows the average total processing time for each query graph on the HUMAN dataset (subfigure (a)), YEAST dataset (subfigure (b) and the HPRD datset (subfigure (c)) for the four algorithms. First of all, it is interesting to see that our results are completely different from those obtained in [6] as none of the algorithms exceed 5 hours of execution on the large queries: 100s, 200s, 100n

and $200n$. In fact, the experiments undertaken in [6] on the same data graphs and the same set of queries report that Turbo$_{Iso}$ exceeds 5 hours running time on these large queries on almost the all three datasets on an Intel i5 3.20 GHz CPU and 8GB memory. We recall that, we used the binaries provided by the authors and consequently no modifications have been done on these algorithms. According to our results plotted on Figure 4.7, there is practicably no difference between the four algorithms on the HUMAN data graph whatever is the size of the query and its sparsity. We note that this dataset is highly compressible and is suitable for algorithms such as Turbo$_{Iso}$ and Sum$_{Iso}$.

For YEAST and HPRD, we clearly see that *CNI* outperforms CFL-match and Sum$_{Iso}$, which behave almost similarly on all queries, and both perform better than Turbo$_{Iso}$ that obtains the worst time performance. This is due to our new neighborhood encoding that allows an easy global pruning step.

Against Existing Algorithms by Varying $|\Sigma|$ ***within the small datasets***: Figure 4.8 shows the average total processing time for each query graph on the DANIO-RERIO for the four algorithms with various numbers of query labels and also two distributions (uniform and Gaussian) of these labels on vertices. These graphs are queried by 2 sets of queries: sparse and non sparse queries. Each set contains 100 query graphs of the same size (128 vertices). We can see on this Figure that the worst results are obtained by Turbo$_{Iso}$. This can be explained by the complexity of its data structures when we list all the embeddings [6]. CFL-match and sum$_{Iso}$ have very close results on sparse queries on all the considered label numbers and with the two distributions. However, sum$_{Iso}$ behaves better with non sparse queries mainly because the corresponding grahs are more likely compressible. CNI clearly outperforms the three other algorithms which confirms the importance to reduce filtering cost.

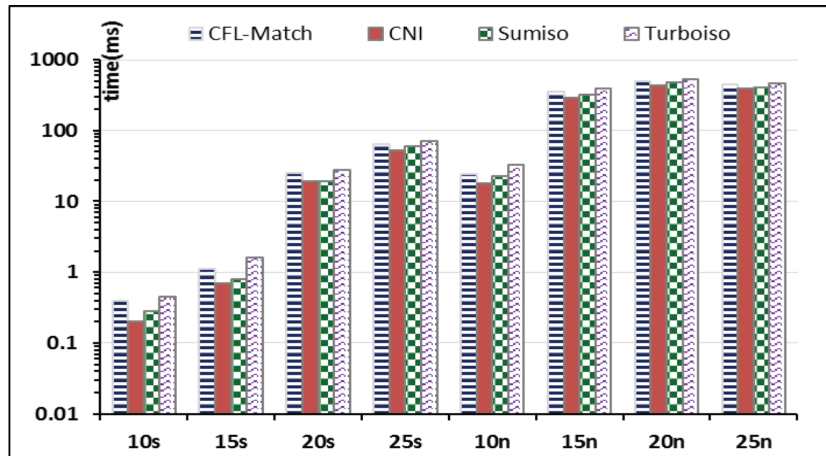
Against Existing Algorithms by Varying $|V(Q)|$ ***within the large dataset***:

Figure 4.10 shows the average total processing time for each query graph our large dataset LIVEJOURNAL. The obtained results have the same pattern as the results obtained on small graphs. However, the difference between the four algorithms is less pronounced than with small graphs. This can be explained by the fact that the small graphs are more difficult instances for subgraph isomorphism search with denser graphs.

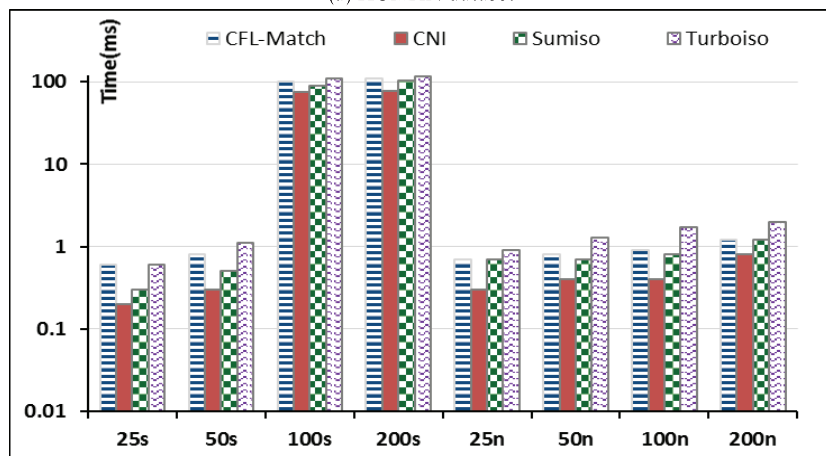
Against Existing Algorithms by Varying $|V(Q)|$ within the big datasets: It was not possible to use CFL-match, Turbo_{Iso}, and Sum_{Iso} with big graphs. So, the results concern only *CNI*. Figure 4.9 shows the total processing time of *CNI* on the two big graphs. We can mainly see that even with a query graph of $500,000$ vertices we cannot perceive any exponential shape which confirms the scalability of the approach. This tendency is also confirmed when we vary the number of vertices of the data graph on Figure 4.11. These results definitely settle the scalability of the proposed approach.

4.5 $cni(v)$ at $(k > 1)$-hops Neighborhood

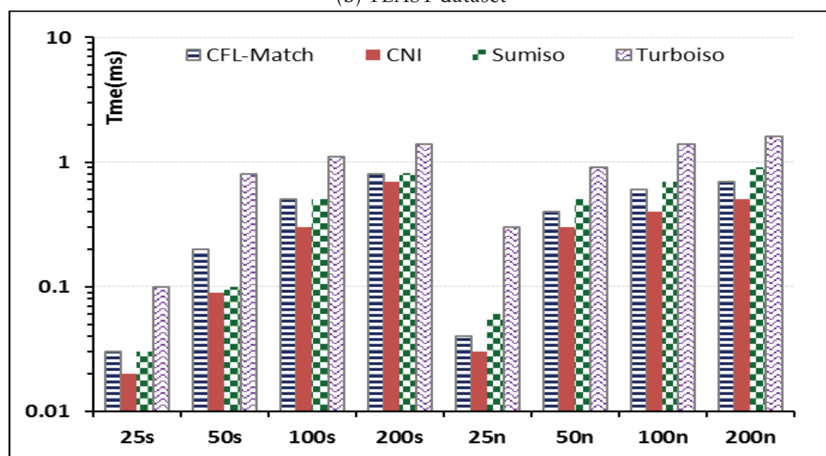
The compact neighborhood index can also be computed for the k-neighborhood with $k > 1$ and can be extended to cover edge labels. The CNI of vertex v featuring its neighborhood at k-hops can be computed using the same formula: $cni_k(v) = \sum_{j=1}^{s} \hbar(j, x_1 + ... + x_j)$ where $\hbar(q,p) = \binom{q+p-1}{q} = \frac{(q+p-1)!}{q!(p-1)!}$, s is the number of k-hops neighbors of v, i.e, number of vertices of G that are reachable from v with exactly k-hops in a shortest path from v, and $x_1, \cdots, x_s$ are the numeric labels of these vertices. For instance, the CNI at $k = 2$ of the query vertex u_1 of our running example (see Figure 4.1 comprises vertices u_4 and u_5 and can be computed as: $cni_2(u_1) = \hbar(1,3) + \hbar(2,4) = 7$. The CNI at k-hops can be used to prune the data vertices that are not candidate for a query vertex but that passe



(a) HUMAN dataset

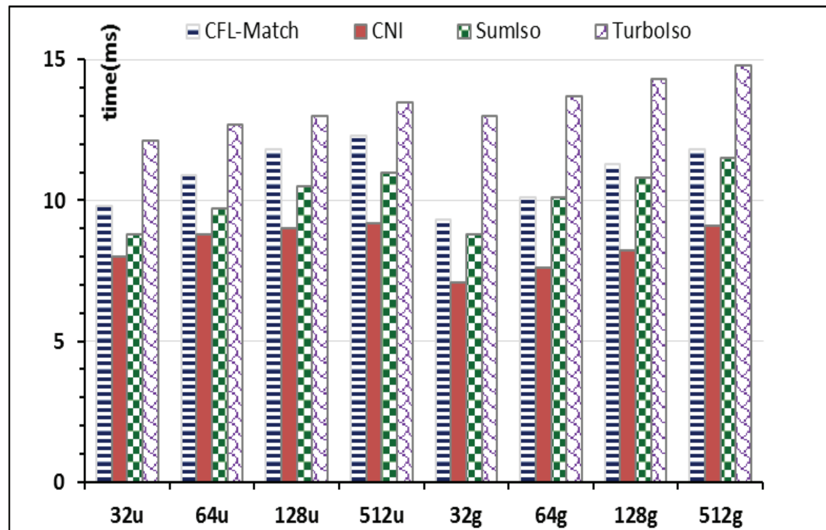


(b) YEAST dataset

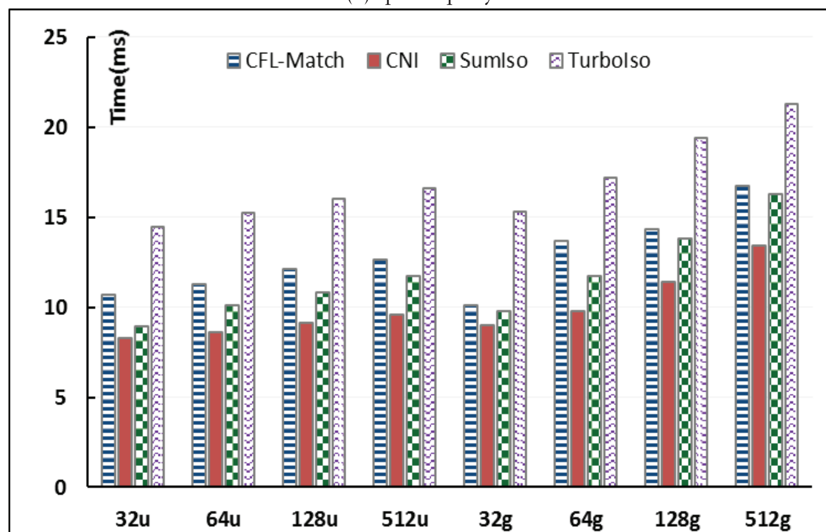


(c) HPRD dataset

Figure 4.7: Time performance on small datasets (varying $|V(Q)|$). Results are in logscale.

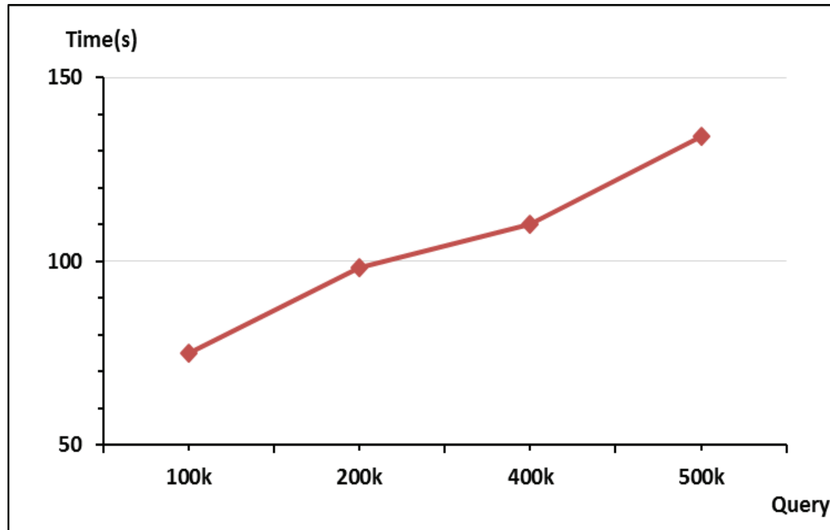


(a) sparse query

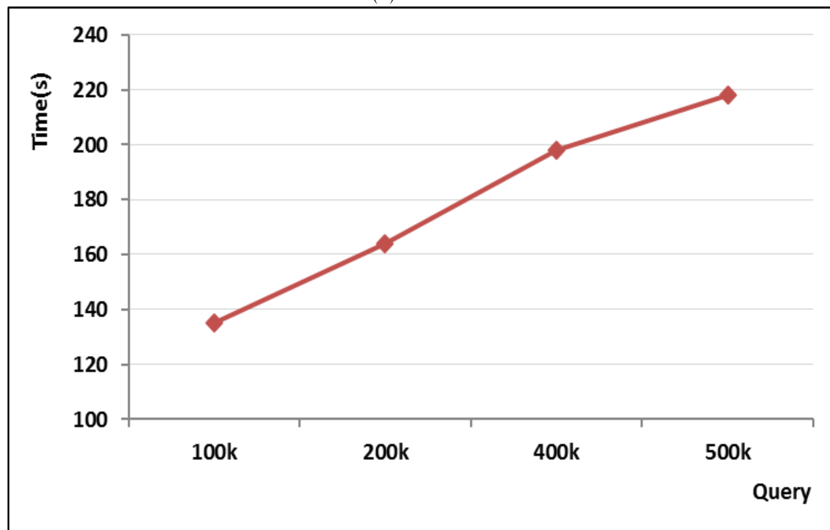


(b) non sparse query

Figure 4.8: Time performance on the small dataset DANIO-RERIO (varying $|\Sigma|$ and the label distribution).



(a) Twitter



(b) Friendster

Figure 4.9: Scalability testing (varying $|V(Q)|$).

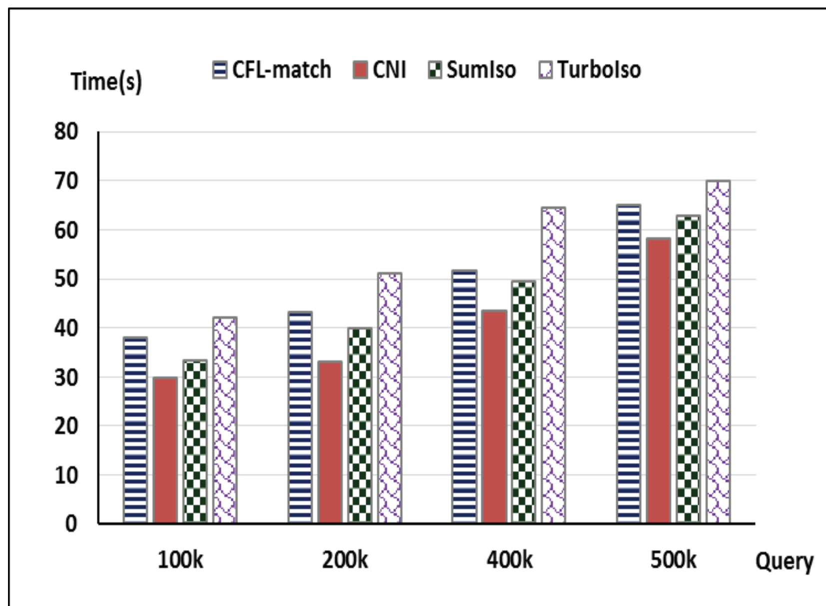


Figure 4.10: Scalability testing on large graphs (varying $|V(Q)|$).

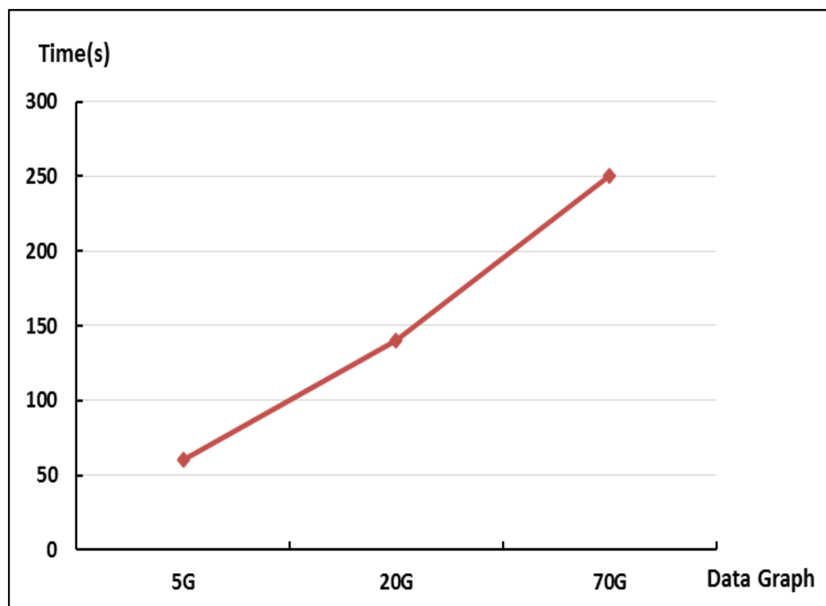


Figure 4.11: Scalability testing (varying $|V(G)|$).

through the $(k-1)$-hop CNI as follows:

Lemma 7 (k-hop Degree filter)**.** *Given a query Q and a data graph G, a data vertex* $v \in V(G)$ *is not a candidate of* $u \in V(Q)$ *if* $deg^k_{\mathcal{L}(Q)}((v) < deg^k_{\mathcal{L}(Q)}(u)$ *where* $deg^k_{\mathcal{L}(Q)}(u)$ *is the number of vertices reachable from u with exactly k-hops in a shortest path from u and have a label in* $\mathcal{L}(Q)$*.*

Lemma 8 (CNI_k filter)**.** *Given a query Q and a data graph G, a data vertex* $v \in V(G)$ *that verifies the* CNI_k *filter and the* $(k+1)$*-hops Degree filter is not a candidate of* $u \in V(Q)$ *if* $cni_{k+1}(v) < cni_{k+1}(u)$*.*

Lemma 7 is straightforward. The proof of Lemma 8is similar to the proof of Lemma 3.

To cover edge labels, a CNI can also be computed for the edges at several hops as for vertices. A CNI for edges can be used as a first filter before testing the compatibility of labels edge by edge.

4.6 Conclusion

Subgraph isomorphism search is an NP-complete problem. This means a processing time that grows with the size of the involved graphs. Pruning the search space is the pilar of a scalable subgraph isomorphism search algorithm and has been the main focus of proposed approaches since Ullmann's first solution. In our contribution, we proposed *CNI*, a simple subgraph isomorphism search algorithm that relies on a compact representation of the neighborhood, called Compact Neighborhood Index (*CNI*), to perform an early global pruning of the search space. *CNI* distills topological information of each vertex into an integer. This vertex encoding is easily updatable and can be used to prune globally the search space using an iterative algorithm. Furthermore *CNI* does not require

that the entire data graph is loaded into main memory and can be used with a graph stream. Our extensive experiments validate the efficiency of our approach.

As part of future work related to this second contribution, it will be interesting to extend CNI to construct a graph index that allows to handle a graph database. For this issue, we plan to compute a vertex CNI that includes the vertex label: $cni(u) = \sum_{j=1}^{k} \hbar(j, x_1 + ... + x_j)$ where the label of u is among the x_i and then compute a compact neighborhood index for the graph using the same formula as follows: $cni(G) = \sum_{j=1}^{k} \hbar(j, x_1 + ... + x_j)$ where each x_i is the CNI of a vertex of G. This resulting graph CNI can be used to index a graph in a database of graphs defined on the same set of labels.

Chapter 5: Conclusion and Perspectives

Contents

5.1 Conclusion 111
5.2 Perspectives 114

5.1 Conclusion

To conclude this manuscript, we present, in the following, a summary of the work that we achieved during our thesis. The research perspectives that could be considered following this work are also discussed.

In this thesis, we studied the problem of subgraph isomorphism search in massive data. Subgraph isomorphism search is the main tool used for graph querying. It is an NP-complete problem. Basically, this problem consists to determine an equality between two graphs in terms of structure and labels. It also finds a mapping between all the vertices and/or edges of the query graph and the target graph while respecting the labeling functions. Graph querying can be very usefull. For example, in chemistry, scientists usually aim to find a small complex molecule in a big one during their tests. Such a problem can be solved using subgraph isomorphism seach with a graph representation of

molecules.

As presented in the state of the art (see Chapter 2), many algorithms and solutions were proposed to solve subgraph isomorphism search efficiently. The main problem is how to reduce the search space to save memory space and time processing.
A search space is generally a tree that the algorithm has to parse to search for the query. The very first and most used technique to browse the search space is backtracking, proposed first by Ullmann [49]. Algorithms that followed, rely on Ullmann's solution and try to outperform it by further reducing the size of the search space. This is done by filtering unpromising vertices, that can not answer the query, as soon as possible.
Many techniques were proposed to reduce the search space, some of them use paths as patterns of comparison. Instead of checking for isomorphism with all vertices, the search will be performed on a shorter list of candidates. A candidate is a pattern that shows more probability to answer the query. Some techniques use score functions to determine if a candidate is relevant or not. After reducing the search space by returning a list of relevant candidates a second phase, called verification phase, is performed on the final list of relevant candidates to check for subgraph isomorphism.

In the second chapter of this thesis, we presented the subgraph isomorphism search problem deeply by showing its utility to query graphs, and by presenting existing methods that are the most related to what we have done in our contributions.
We catigorized the subgraph matching problem into two categories: the first one consists to find all graphs, in a graph database, that contain the query. This category is called subgraph containement search over a graph database. The second category, in which we focused our work, is subgraph matching over a

single large data graph. This problem is more dificult than the first one, because here we aim to find all the occurrences of the query in the data graph, instead of checking for the exsistance of the query on each graph in a database.

After analyzing our state of the art, we presented our two contributions that globaly aim to reduce the search space by compression.
In the first contribution, we compress the whole data graph. In fact, a smaller representation of the graph will definitely lead to a smaller search space, which gives less time and memory storage complexity. Graph compression (or summarizing), is a well known technique that is effective when dealing with massive data.

The best compression algorithm is the one that retains all the properties of the original graph. We surveyed several approaches to compress graph and the one that responds to this criteria is a concept from graph theory called modular decomposition and that dates back to the work of Gallai in 1967 [17].
Modular decomposition of graphs consists to highlight a set of vertices that have the same neighbors and so are not distinguishable from outside. These sets of vertices are called modules. Each module is compressed as a single vertex depending on how vertices are connected within a module. We showed that we can query these compressed graphs without decompressing them. Our experimentations show that the proposed approach achieves good performance on both time processing of queries and space storage of data graphs.

In our second contribution, we compress the neighborhood of each vertex. In this contribution, we focused on the best way to filter the search space. We proposed a new constant time pruning mechanism. The main idea in this contribution was to avoid comparing all vertex's neighbors, in order to check if two vertices are identical or not. To do so, we regroup all information that surround a vertex on one simple integer.

We used a bijective function that basically performs specific mathematical operations on vertex's neighbors to obtain one integer. Because this function is bijective, we are sure that two vertices with the same label and number of neighbors, that get the same result with the bijective function, are the same. This simple neighborhood encoding reduces the time complexity of vertex filtering.

Our encoding mechanism is also adapted to massive graphs, that do not fit into memory. We perform one sequantial pass of the disk file, and retain all needed information. This avoids expensive random disk accesses. The new encoding is called Compact Neighbourhood Index (CNI), with which the filtering phase is processed with integer comparisions, and relies on the characteristics of CNIs to ensure a global prunning of the search space.
After this filtering, the input graph used to perform the final subgraph search contains only vertices that are candidates for query vertices. Finally, the depth first search subroutine of Ullmann's algorithm is used to list the subgraphs of the filtered data graph that are isomorphic to the query by verifying the adjacency relationships.
Our extensive experiments validate our approach.

5.2 Perspectives

Our work can be extended according to several axes:

- In our first contribution, we relied on graph compression. It is interessting to do more research on the existing compression methods, to propose new methods for graph compression. To be efficient, a compression method must preserve all graph's information, without losing any significant structure information. The idea will be to find a way to compact a larger number of vertices on one module, without losing information. Such technique

must also facilitate the subgraph search on modules without decompressing them by storing clear and usefull information related to each vertex. The challenge in our first contribution was that two modules with different types are not necessarily storing different structures and we had to do a lot of testing to decide whether to prune a module or not. So, the data stored for each module must be helpfull on filtering modules.

Another futur issue, that will help handling large graphs, will be to find a way to test these representations on a parralel programming model such as MapReduce.
It is also interesting to see if it is feasible to run such approach on a graph database like Neo4j by designing and developing all the necessary database operations such as create, delete, and insert on the compressed dataset.

- In our second contribution, the idea was to propose an effecitve and very tight filtering technique to filter-out unpromising vertices as soon and effective as possible. Our filtering was based on the vertex's neighborhood information.

  Even if our method shows good performance, the issue is that we have to compute the CNI for each vertex. It will be interesting to extend this technique by constructing a subgraph neighborhood index, instead of doing it for each vertex. The idea will be to divide the target graph into candidate subgraphs. A candidate subgraph is a subgraph that have more possibility to match the query. Unlike our contribution, the idea will be to compact all neighborhood information of the subgraph on one integer.
  For example: if we have a target graph of 100 vertices divided into 10 candidate subgraphs, the index will contain 10 integers storing all necessary information, instead of calculating 100 integeres. The last comparison

will be between the query, which will also have an integer regrouping all neighborhood information, and 10 other integers (representing the target graph). This will largely reduce the processing time, and the search space. If the query and the target graph are large graphs, the query will be also divided into subgraphs, each one with its CNI and the target graph's subgraphs in this case will be candidates for the query subgraphs instead of being candidate for the whole query.

Another issue will be to extend CNI to construct a graph index that allows to handle a graph database. For this, we plan to compute a vertex CNI that includes the vertex label. The resulting graph CNI can be used to index a graph in a database of graphs defined on the same set of labels.

- Another perspective that seems interesting is to developp a hybrid method that takes the advantages of the first contribution (the power of compression), and the effectiveness of the second method's filtering. Such work, according to each contribution's advantages, will be effective, with less time consuming, less memory space usage, and less searching space.

List of Publications

Journal paper

[1] Chemseddine Nabti, Hamida Seba. Querying Massive Graph Data: a Compress and Search Approach. *Future Generation Computer Systems (FGCS), Volume 74, September 2017, Pages 63-75, Elsevier*

International conferences with proceeding

[2] Chemseddine Nabti, Hamida Seba. Subgraph Isomorphism Search in Massive Graph Databases. In *The International Conference on Internet of Things and Big Data – IoTBD 2016, 23-25 Avril 2016, Rome (Italie). HAL : hal-01313922.* ***Best paper award***.

International conferences without proceeding

[3] Chemseddine Nabti, Hamida Seba. Querying Massive Graph Data: a Compress and Search Approach. In *High Quality Journals Forum, in IoTbds2017, 24-26 Avril 2017, Porto (Portugal).* ***Best Presentation Award***.

Paper Under Submission

[4] Chemseddine Nabti, Hamida Seba. Compact Neighborhood Index for Subgraph Queries in Massive Graphs. *A preliminarily version is available in Arxiv https://arxiv.org/abs/1703.05547*.

References

[1] http://socialnetworks.mpi-sws.org/data-imc2007.html.

[2] Micah Adler and Michael Mitzenmacher. Towards compressing web graphs. In *Proceedings of the Data Compression Conference*, DCC '01, Washington, DC, USA, 2001. IEEE Computer Society.

[3] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[4] Boanerges Aleman-Meza, Christian Halaschek-Wiener, Satya Sanket Sahoo, Amit Sheth, and I Budak Arpinar. Template based semantic similarity for security applications. In *International Conference on Intelligence and Security Informatics*, pages 621–622. Springer, 2005.

[5] Endika Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Sup´erieure des Telecommunications Departement Traitement du Signal et des Images Ecole doctorale Edite, University of the Basque Country Computer Engineering Faculty Intelligent Systems Group, Paris, FRANCE, 2002.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1199–1214, New York, NY, USA, 2016. ACM.

[7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214. ACM, 2016.

[8] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM.

[9] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(Suppl 7)(S13), 2013.

[10] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed xml. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 141–152. VLDB Endowment, 2003.

[11] Christian Capelle, Michel Habib, and Fabien De Montgolfier. Graph decompositions and factorizing permutations. *Discrete Mathematics & Theoretical Computer Science - DMTCS*, 5(1):55–70, 2002.

[12] Chen Chen, Cindy X. Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. Mining graph patterns efficiently via randomized summaries. *Proc. VLDB Endow.*, 2(1):742–753, August 2009.

[13] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18:265–298, 2004.

[14] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the vf graph matching algorithm. In *Image Analysis and Processing, 1999. Proceedings. International Conference on*, pages 1172–1177. IEEE, 1999.

[15] Wenfei Fan and Jin-Peng Huai. Querying big data: Bridging theory and practice. *Journal of Computer Science and Technology*, 29(5):849–869, 2014.

[16] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 157–168, New York, NY, USA, 2012. ACM.

[17] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18:25–66, 1967.

[18] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18:25–66, 1967.

[19] Michael Randolph Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[20] S Greenblatt, S Marcus, and T Darr. Tmods-integrated fusion dashboard-applying fusion of fusion systems to counter-terrorism. In *Proc. International Conference on Intelligence Analysis*, 2005.

[21] M. Habib and C. Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.

[22] Michel Habib, Fabien De Montgolfier, and Christophe Paul. A simple linear-time modular decomposition algorithm for graphs. *Scandinavian Workshop on Algorithm Theory - SWAT*, pages 187–198, 2004.

[23] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, NY, USA, 2013. ACM.

[24] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, NY, USA, 2013. ACM.

[25] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[26] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[27] Harry T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *J. ACM*, 22(1):11–16, January 1975.

[28] Sharanya Jayaraman. Fg-index: Towards verification-free query processing on graph databases. 2013.

[29] Donald E Knuth. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129):122–136, 1975.

[30] Sofiane Lagraa and Hamida Seba. An efficient exact algorithm for triangle listing in large graphs. *Data Mining and Knowledge Discovery*, pages 1–20, 2016.

[31] Sofiane Lagraa, Hamida Seba, Riadh Khennoufa, Abir M'Baya, and Hamamache Kheddouci. A distance measure for large graphs based on prime graphs. *Pattern Recognition*, 47(9):2993 – 3005, 2014.

[32] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the VLDB Endowment*, volume 6, pages 133–144. VLDB Endowment, 2012.

[33] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 133–144. VLDB Endowment, 2013.

[34] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6(1) 29–123, 2009*, 6(1):29–123, 2009.

[35] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 641–650. ACM, 2010.

[36] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 177–187, New York, NY, USA, 2005. ACM.

[37] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 533–542, New York, NY, USA, 2010. ACM.

[38] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *CoRR*, abs/1301.1493, 2013.

[39] Ryan Boyd William Lyon Michael Hunger. Rdbms graphs: Sql vs. cypher query languages. https://neo4j.com/blog/sql-vs-cypher-query-languages/, 2015.

[40] Chemseddine Nabti and Hamida Seba. Querying massive graph data: A compress and search approach. *Future Generation Computer Systems*, 74:63 – 75, 2017.

[41] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, October 2004.

[42] K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener. The link database: fast access to graphs of the web. In *Data Compression Conference, 2002. Proceedings. DCC 2002*, pages 122–131, 2002.

[43] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proc. VLDB Endow.*, 8(5):617–628, January 2015.

[44] Hamida Seba, Sofiane Lagraa, and Elsen Ronando. Comparison issues in large graphs: State of the art and future directions. *CoRR*, abs/1502.07576, 2015.

[45] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, August 2008.

[46] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, August 2008.

[47] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[48] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*. Lomza, Poland, May 2012.

[49] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[50] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 913–924, New York, NY, USA, 2011. ACM.

[51] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346, New York, NY, USA, 2004. ACM.

[52] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[53] TARA SAFAVI DANAI KOUTRA YIKE LIU, ABHILASH DIGHE. Graph summarization: A survey. ACM Computing Surveys, 2017.

[54] Bob Yirka. Computer scientist claims to have solved the graph isomorphism problem. https : / / phys . org / news / 2015-11-scientist-graph-isomorphism-problem.html/tutoriel/, 2016.

[55] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 192–203, New York, NY, USA, 2009. ACM.

[56] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 192–203, New York, NY, USA, 2009. ACM.

[57] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340–351, September 2010.

[58] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

[59] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: Tree + delta graph. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 938–949. VLDB Endowment, 2007.

[60] Xiang Zhao, Chuan Xiao, Xuemin Lin, and Wei Wang. Efficient Graph Similarity Joins with Edit Distance Constraints. In *IEEE 28th International*

Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, pages 834–845, 2012.

[61] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. Treespan Efficiently computing similarity all-matching. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 529–540, New York, NY, USA, 2012. ACM.

[62] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. A novel spectral coding in a large graph database. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 181–192, New York, NY, USA, 2008. ACM.