Subgraph Isomorphism Search in Massive Graph Databases

Chemseddine Nabti, Hamida Seba

Subgraph Isomorphism Search in Massive Graph Databases

C. NABTI[1] and H. SEBA[1]
[1] *Université de Lyon, CNRS, Université Lyon 1, LIRIS, UMR5205, F-69622, France*
hamida.seba@univ-lyon1.fr

Keywords: Subgraph isomorphism, graph query, massive graph databases, graph summarizing, modular decomposition.

Abstract: Subgraph isomorphism search is a basic task in querying graph data. It consists to find all embeddings of a query graph in a data graph. It is encountered in many real world applications that require the management of structural data such as bioinformatics and chemistry. However, Subgraph isomorphism search is an NP-complete problem which is prohibitively expensive in both memory and time in massive graph databases. To tackle this problem, we propose a new approach based on concepts widely different from existing works. Our approach relies on a summarized representation of the graph database that minimizes both the amount space required to store data graphs and the processing time of querying them. Experimental results show that our approach performs well compared to the most efficient approach of the literature.

1 INTRODUCTION

Graphs are naturally used to represent data in several domains and applications such as protein interactions, regions of images, 2D and 3D shape recognition, etc. Graphs are also a first class solution for massive data representation. In fact, graph databases enable us to build interesting models that map closely to the problem domain. In this context, querying graphs is a fundamental problem studied by a large research community. Subgraph isomorphism search is the basic type of graph queries. Given a query graph Q and a data graph G, the subgraph isomorphism problem is to find all embeddings of Q in G. However, this problem is NP-complete (Garey and Johnson, 1979) and the problem of finding practical solutions for massive graph databases is a challenge. Subgraph isomorphism knows a lot of works that consist in finding the best way to match a query graph with the data graphs. We review here the main existing algorithms. For an exhaustive and detailed list of solutions, several surveys are available on the topic such as (Lee et al., 2013) and (Gallagher, 2006). Existing solutions are generally classified into two main categories:

1. Exact approaches: in this case, we are interested by returning the list of subgraphs of G that match exactly the query Q.
2. Inexact or error-tolerant approaches: in this case, we are interested by a ranked list of of subgraphs of G that are most similar to the query Q.

In both approaches, algorithms may be optimal or approximate. Optimal algorithms return a correct and complete solution but have, generally, an exponential time complexity. Approximate algorithms may not find the correct/complete solution but guarantee a polynomial time complexity.

The inexact approach knows a flourishing research activity in several application domains such as databases and pattern recognition relying on different tools such as genetic algorithms (Khoo and Suganthan, 2001), neural networks (Micheli, 2009), etc. Within this approach, we generally compute a distance between the graphs. This distance reflects the degree of similarity or dissimilarity of the graphs. We focus here on exact subgraph isomorphism search where existing algorithms can be classified within 3 categories:

1. Backtracking-based algorithms such as Ullman's (Ullmann, 1976), VF2 (Cordella et al., 2004), QuickSI (Shang et al., 2008), GraphQL (He and Singh, 2008), GADDI(Zhang et al., 2009), and SPath (Zhao and Han, 2010). This approach constructs a space search tree whose internal nodes correspond to partial solutions and leaves correspond to embeddings. The first exact subgraph isomorphism algorithm is due to Ullman (Ullmann, 1976). Ullmann's basic approach is to enumerate all possible mappings of vertices between the two graphs in a depth-first tree-search. Each node at level i of the search-tree maps a vertex of the query to a vertex of the data graph. Each

path from the root to leaf in the search-tree represents a complete mapping between the query and a subgraph of the data graph. This search-space increases exponentially with the size of the input graphs, so all the solutions proposed in this approach are based on some pruning rules that prevent developing non necessary paths.

2. Indexing-based algorithms such as Grep (Shasha et al., 2002), gIndex (Yan et al., 2004), FG-Index (Cheng et al., 2007), Tree+$\triangle$ (Zhao et al., 2007), gCode (Zou et al., 2008), SwiftIndex (Shang et al., 2008), and C-Tree (He and Singh, 2006). In this approach, indexes are used to minimize the number of candidate graphs in the database. Then, a subgraph isomorphism search is launched on the candidates.

3. Candidate Region selecting algorithms such as Turbo$_{ISO}$ (Han et al., 2013). In this approach, the idea is to target specified regions on the same graph for subgraph isomorphism search. These regions are selected according to the properties of the query.

   A candidate region for a query graph Q is a subgraph of the data graph G which may contain embeddings of the query graph. So, performing subgraph isomorphism search on all candidate regions will ensure that all embeddings can be obtained. However, minimizing the number of candidate regions and the size of each region is obviously important for faster matching. The main solution within this approach is called Turbo$_{ISO}$ (Han et al., 2013). In order to minimize the size of each candidate region, the authors of (Han et al., 2013) propose to :

   (a) Rewrite the query Q into an equivalent NEC (Neighborhood Equivalence Class) tree Q' . In Q' each set of vertices that have the same label and the same set of adjacent query vertices are merged into one NEC vertex. So, a NEC vertex is a compressed form of a set of vertices. Consequently, using Q' instead of Q, will accelerate the candidate region exploration process, since the number of vertices is smaller.

   (b) Construct 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 node 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 simply performing depth-first search using T_G and starting from v_s.

   Minimizing the number of regions comes through a careful choice of the root of the NEC tree. For this, Turbo$_{ISO}$ ranks every query vertex u by $Rank(u) = \frac{freq(G,L(u))}{deg(u)}$, 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.

   When exploring candidate regions, Turbo$_{ISO}$ also minimizes the number of enumerated partial solutions by ordering the NEC vertices by increasing sizes. Thus, paths involving fewer vertices are explored first, the space is pruned rapidly if no isomorphism is possible. In (Han et al., 2013), Turbo$_{ISO}$ is compared to the other approaches and its superiority in processing queries is attested via extensive experimentations.

   Another solution that can be classified in this category is STW proposed in (Sun et al., 2012). In this solution, the authors propose a graph decomposition into *STwigs*. An *STwig* is a two level tree structure, $q = (r, L)$, where r is the label of the root node and L is the set of labels of its child nodes. *STwigs* are non overlapping star structures, i.e., edge disjoint stars.

   Given a query graph Q, (Sun et al., 2012) first decomposes Q into a set of STwigs, then it uses exploration to find matches to each STwig. Candidate region exploration concerns the graphs that contain these STWings.

In this paper,we present a new approach for subgraph isomorphism search on large graphs. The proposed approach is completely different from all the previous approaches, and outperforms the most efficient existing algorithm in our experimentation. The main idea of the proposed approach is to enhance both time and space requirements of subgraph isomorphism search on large graphs. This is achieved by working on summarized graphs that are simpler and smaller than the original graphs.

The rest of the paper is organized as follows: Section 2 defines our notations and presents the compression algorithm used to summarize the graphs. Section 3 describes the proposed algorithm for subgraph isomorphism search on summarized graphs. Section 4 presents an experimental evaluation to show the effective performance of the proposed technique over the existing solutions. Finally, Section 5 concludes the paper with a summary of our work and its perspectives.

2 PRELIMINARIES

Basics. We consider data graphs defined as simple[1] vertex labeled graphs. We rely on the terminology used in (Basu and BBA, 2006; Fan et al., 2010).

Definition 1. [2] *A data graph G is a 3-tuple $G = (V, E, \ell)$, where V is a set of nodes (also called vertices), $E \subseteq V \times V$ is a set of edges connecting the nodes, $\ell : V \rightarrow \Sigma$ is a function labeling the nodes where Σ is the sets of labels that can appear on the nodes.*

In this paper, the notation $G = (V, E)$, with ℓ omitted means that we actually do not need the labels of the vertices but just their identifiers (i.e., indexes).

An undirected edge between vertices u and v is denoted indifferently by (u, v) or (v, u). For each $v \in V$, $d(v)$ 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 number of vertices of a graph is called the order of the graph. The number of edges of a graph is called the size of the graph. A graph that is contained in another graph is called a subgraph and can be defined as follows:

Definition 2. *A graph $G_1 = (V_1, E_1, f_{V_1})$ is a subgraph of a graph $G_2 = (V_2, E_2, f_{V_2})$, denoted $G_1 \subseteq G_2$, if $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, $f_{V_1}(x) = f_{V_2}(x)\ \forall x \in V_1$.*

Graph isomorphism is defined as follows:

Definition 3. *A graph $G_1 = (V_1, E_1, f_{V_1})$ and a graph $G_2 = (V_2, E_2, f_{V_2}, f_{E_2})$ are said to be isomorphic, denoted $G_1 \cong G2$, if there exists a bijective function $h : V_1 \rightarrow V_2$ such that the following conditions are met:*

1. $\forall x \in V_1 : f_{V_1}(x) = f_{V_2}(h(x))$
2. $\forall (x, y) \in E_1 : (h(x), h(y)) \in E_2$
3. $\forall (h(x), h(y)) \in E_2 : (x, y) \in E_1$

Given a query graph Q and a data graph G, the subgraph isomorphism search of Q in G consists to find all the subgraphs of G that are isomorphic to Q.

Graph Summarizing. Generally graph summarizing methods designate graph compression methods that aim to reduce the amount of storage space required for storing a graph so that the processing of the graph does not require its decompression. So, this summaries must retain an amount of the graph properties that are sufficient to the application. Actually, this line of research is very new and knows little works. We can cite for example (Chen et al., 2009) where the authors propose to summarize a graph by grouping the vertices that have the same label into super-vertices. In (Fan et al., 2012), the authors summarize graphs in a manner that preserves a class of queries, i.e., a query of this class returns the same result when applied to a graph G and when applied to the compression of G. They considered mainly reachability queries. In this kind of queries, we are interested in verifying if there is a path between two vertices within the graph. In (Lagraa et al., 2014), the authors propose a similarity measure between two large graphs based on a similarity measure between compressed versions of these graphs. They use modular decomposition (Gallai, 1967; Möhring, 1985a) to compress the graphs. A triangle listing algorithm is also proposed on graphs compressed by modular decomposition in (Lagraa and Seba, 2016). These two applications show that modular decomposition is a promising compression method for graphs. A modular decomposition of a graph consists to find within the graph all the sets of vertices that share the same neighboring-hood. These sets of vertices are called *modules*. In our framework, we also rely on modular decomposition to compress graphs. Modular decomposition is a graph representation method introduced by Gallai (Gallai, 1967) to solve optimization problems. It was also used to recognize some graph classes (Möhring, 1985a; Möhring, 1985b; Spinrad, 2003). For a survey of applications of modular decomposition see (Gallai, 1967; Möhring, 1985a; Dahlhaus et al., 1997). The basic concept of modular decomposition, used in the compression process, is the notion of *module* defined as follows:

Table 1: Notation

Symbol	Description
$G = (V, E, \ell)$	undirected vertex labeled graph, f is a labeling function
$V(G)$	vertex set of the graph G
$E(G)$	edge set of the graph G
$\overline{G}$	the complement of the graph G
$d(v)$	degree of vertex v
$G[X]$	the subgraph of G induced by the set of vertices X
$\mathcal{C}(G)$	compressed graph of G
$root(C(G))$	root of the tree corresponding to $\mathcal{C}(G)$
$Father(x)$	the module that contains vertex (or module) x
$Leaves(x)$	set of vertices contained in module x
$\ell(x)$	set of label of vertex (or module) x

Definition 4. *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 $V \setminus M$.*

A module M of G can take one of the following

[1] Simple graphs are graphs with no edges involving a single vertex.

[2] For presentation simplicity, we do not consider edge labels.

types:

- **Series:** if $G[M]$ is a clique.
- **Parallel:** if $\overline{G}[M]$ is a clique.
- **Neighborhood:** Both, $G[M]$ and $\overline{G}[M]$ are connected graphs.

Figure 1 presents a graph and its modules.

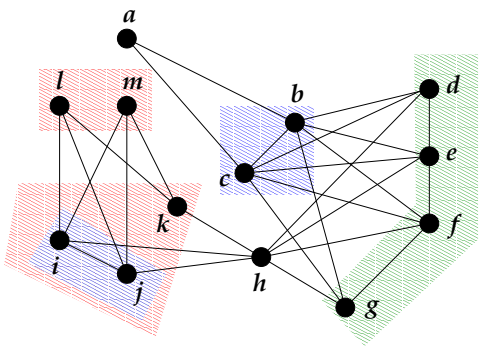


Figure 1: A Graph and its Modules (Lagraa et al., 2014).

The graph is compacted by replacing recursively each module by a supervertex as illustrated in Figure 2. To obtain a unique representation of the graph only the modules that do not overlap other modules are considered.

To retain all the properties of the original graph with the obtained compact representation of the graph, adjacency information for neighborhood modules must be stored. Series and parallel modules need no information about adjacency. For example, The obtained compressed graph illustrated in Figure 2(e) is itself a neighborhood module that can be denoted :

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

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 $N(d,e,f,g)$.

We note also that each module is a tree whose leaves are the vertices of the original graph as illustrated in Figure 3 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. Table 1 summarizes our notations.

Modular decomposition has been the subject of extensive research for years (Gallai, 1967; Möhring, 1985a; Dahlhaus et al., 1997; Habib and Paul, 2010; Quaddoura and Mansour, 2010). Several algorithms that compute the modular decomposition of a graph are proposed in the literature (Habib et al., 2004). The most efficient are linear time (Capelle et al., 2002; Habib et al., 2004; Tedder et al., 2008) and achieves in $O(n+m)$.

3 SUM$_{ISO}$: SUBGRAPH ISOMORPHISM SEARCH ON SUMMARIZED GRAPHS

As mentioned before, we propose here an algorithm that finds all the embeddings of a query graph Q in a data graph G. Both Q and G are compressed as described in the previous section. We show that by doing this we enhance both memory requirement for storing the data graphs and the processing time of the search. The algorithm, called Sum_{ISO}, 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. 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 embeddings by parsing the compressed graph data.

3.1 Candidate Supervertex Selection

The aim of this phase 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 vertices 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. Algorithm 1 details candidate supervertex selection. Figure 4 illustrates this step on our example. In this Figure, we can see that the query is compressed in a single supervertex labeled $S(P(b,c),a)$. $\ell(\mathcal{C}(Q)) = \{a,c,b\}$. Consequently, $Cand = \{1,2\}$, where 1, and 2 are the identifiers of the supervertices that are candidates to match

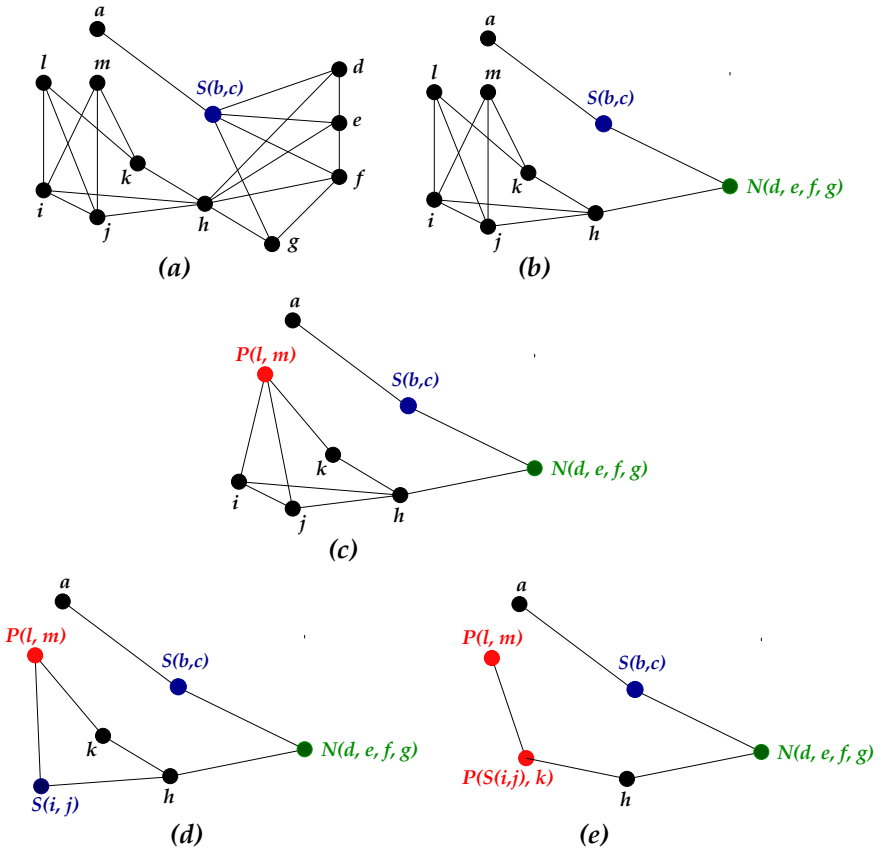


Figure 2: Compressing steps: *S*: series module. *P*: parallel module. *N*: neighborhood module (Lagraa et al., 2014).

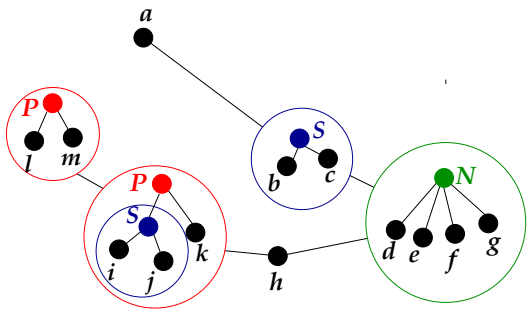


Figure 3: Tree representation of Modules (Lagraa et al., 2014).

the query. The partitioning of *Cand* yields to the set $\{\{1,2\}\}$. This means that there is only one possible region of the graph to explore for subgraph isomorphism.

Algorithm 1: Supervertex Selection.

Data: A summarized data graph $\mathcal{C}(G)$ and a summarized query $\mathcal{C}(Q)$.
Result: A set of candidate supervertices of $\mathcal{C}(G)$ that match $\mathcal{C}(Q)$.

```
begin
    Cand ← ∅;
    foreach m ∈ C(G) do
        if ℓ(Q) ∩ ℓ(m) ≠ ∅ then
            Cand ← Cand ∪ {m};
        end
    end
    C ← {s = {m_1, m_, ··· , m_j} | ℓ(Q) ⊆ ℓ(s)};
    foreach s ∈ C do
        P ← ∅;
        SubgraphSearch(C(Q), s, P);
    end
end
```

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.

3.2 SUBGRAPH SEARCH

The subgraph search phase 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 match u. Algorithm 2 shows all the details of this phase. 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 of neighbors of a vertex in a compressed graph, we use function *Neighbors* that takes advantage from the tree structure of the compressed graph (see Figure 3 to easily list the neighbors of a vertex as detailed in Algorithm 4. 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 backtrack to the preceding state by removing the match.

Algorithm 2: SubgraphSearch.

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
```

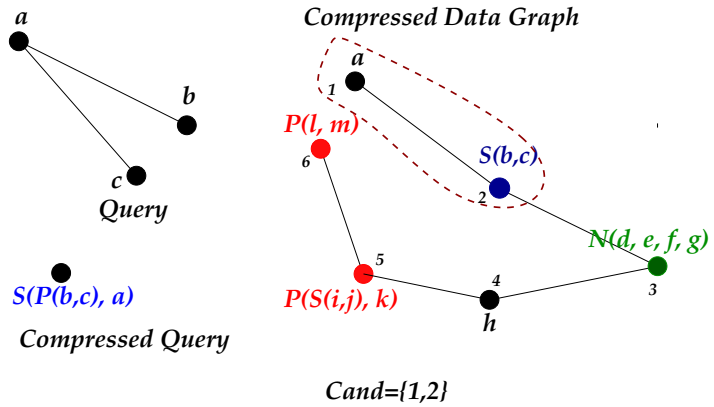


Figure 4: Example of Candidate Supervertex Selection.

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 $\mathcal{C}(G)$.
Result: The set of neighbors of u in g.
begin
  $N \leftarrow \emptyset$;
  $z \leftarrow u$;
  $x \leftarrow Father(u)$;
  while $x \neq root(\mathcal{C}(G))$ **do**
    switch *type of* x **do**
      case *a series module*
        foreach *child* $y \neq z$ *of* x **do**
          $N \leftarrow N \cup Leaves(y)$;
        end
      case *a Neighborhood module*
        foreach *edge* $(z,y) \in x$ **do**
          $N \leftarrow N \cup Leaves(y)$;
        end
    endsw
    $z \leftarrow x$;
    $x \leftarrow Father(x)$;
  end
  return N
end

4 EXPERIMENTAL RESULTS

We evaluate the execution time performance of our algorithm, 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 (Han et al., 2013). We recall that Turbo$_{ISO}$ is itself compared to the other existing solutions in (Han et al., 2013) and showed to be superior to them.

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

4.1 Datasets

We use the same datasets considered in (Han et al., 2013) for proving the superiority of Turbo$_{ISO}$ against the other algorithm of the literature described in Section 1. These datasets are referred to as AIDS, NASA, and Human. Their description is as follows:

- *AIDS database*: This dataset consists of graphs representing molecular compounds. It contains $10,000$ small 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 database*: This dataset consists of one large graph representing a protein interaction network. This graph has $4,675$ vertices and $86,282$ edges. The number of unique labels in the dataset is 90.

We present a summary of these graph databases in Table 2. 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.

Dataset	Number of graphs	$avg\|V\|$	$avg\|E\|$	CR
AIDS	10,000	26	27	56.8%
NASA	36,790	94	32	44.2%
HUMAN	1	4,675	86,282	61%

Table 2: Graph Dataset Characteristics. $avg|V|$: average number of vertices. $avg|E|$: average number of edges. CR average compression rate.

Graphs within the three datasets were preliminarily compressed using an extension of the algorithm proposed in (Capelle et al., 2002; Habib et al., 2004) 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 reports the size on disk of each dataset before and after compression. We also provide the time necessary to compress each dataset.

Dataset	Size on disk (Mb)	Size on disk after compression (Mb)	Compression time(ms)
AIDS	4.59	2.18	230
NASA	24	14.42	180
HUMAN	1.15	0.24	195

Table 3: Size on disk.

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++.

We use the same query sets as in (Han et al., 2013). These queries are constructed as follows (Han et al., 2013):

- AIDS and NASA query sets: For each of these datasets, (Han et al., 2013) 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, (Han et al., 2013) 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 (He and Singh, 2008).
  3. Path queries where the query subgraph is a path. A path query corresponds to transcriptional or signaling pathways (He and Singh, 2008).

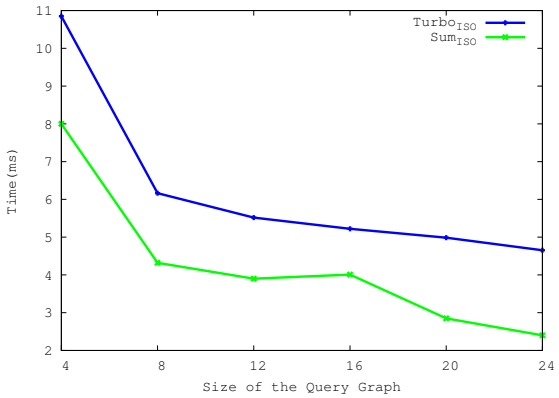


Figure 5: Aids dataset.

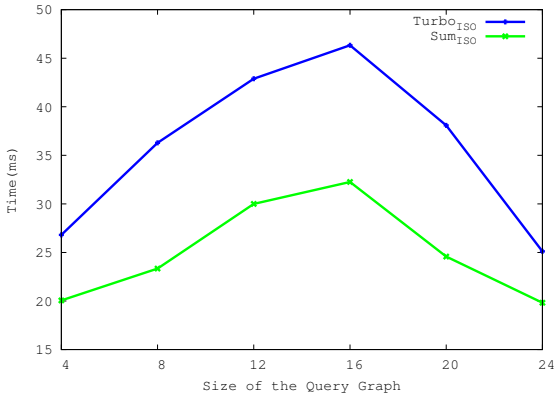


Figure 6: NASA dataset.

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

4.2 Results

Figure 5 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 (Han et al., 2013) 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 6 shows the experimental results for NASA. For this dataset, Sum$_{ISO}$ achieves significantly better than Turbo$_{ISO}$ for all the queries.

Figure 7 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 8 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 node in our approach. For path queries, we have also a better results than Turbo_{ISO} even if not significantly. We explain this by the fact that paths are not summarized by modular decomposition.

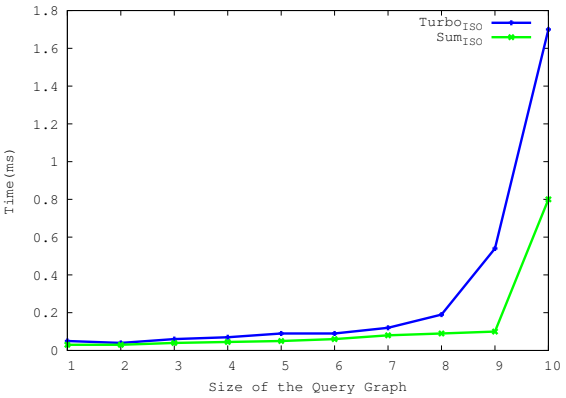


Figure 7: Human dataset.

5 CONCLUSIONS

In this paper, we presented a new approach to subgraph isomorphism search in massive graph databases. In our approach, data graphs are summarized to minimize storage requirement. Our subgraph isomorphism search algorithm, Sum_{ISO}, finds all the embeddings of a query graph in a summarized data graph without decompressing the graph. Our experimentations show that the proposed approach achieves good performance on both time processing of queries and space storage of data graphs. However, several enhancement issues are possible and merit investigation. First, more experiments are needed to attest the efficiency of the approach. In fact, we used the same datasets as in (Han et al., 2013) for our evaluation to compare with the Turbo_{ISO} algorithm. These datasets, namely AIDS, NASA and HUMAN are highly compressible with more than 40% of compression rate. However, not all the graphs are compressible with the same rate. So, it is interesting to study the behavior of the approach behaves with less compressible graphs and determine a compression rate threshold that conditions the use of compression. Second, it is interesting to compare the two approaches on larger graphs. Third, it is interesting to see if it is feasible to ran such an approach on a graph database such as Neo4j and also investigate how it can be implemented in a MapReduce framework. Finally, we have not used pruning methods in the SubgraphSearch phase and it may be possible to define some rules to prune the explored compressed paths by relying on the properties of the compression.

6 Acknowledgments

This work was supported by a grant from the ANR CAIR project.

REFERENCES

Basu, M. and BBA, T. K. H. (2006). *Data Complexity in Pattern Recognition*. Springer.

Capelle, C., Habib, M., and Montgolfier, F. D. (2002). Graph decompositions and factorizing permutations. *Discrete Mathematics & Theoretical Computer Science - DMTCS*, 5(1):55–70.

Chen, C., Lin, C. X., Fredrikson, M., Christodorescu, M., Yan, X., and Han, J. (2009). Mining graph patterns efficiently via randomized summaries. *Proc. VLDB Endow.*, 2(1):742–753.

Cheng, J., Ke, Y., Ng, W., and Lu, A. (2007). Fg-index: Towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 857–872, New York, NY, USA. ACM.

Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. (2004). A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1367–1372.

Dahlhaus, E., Gustedt, J., and McConnell, R. (1997). Efficient and practical modular decomposition. In *eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 26–35.

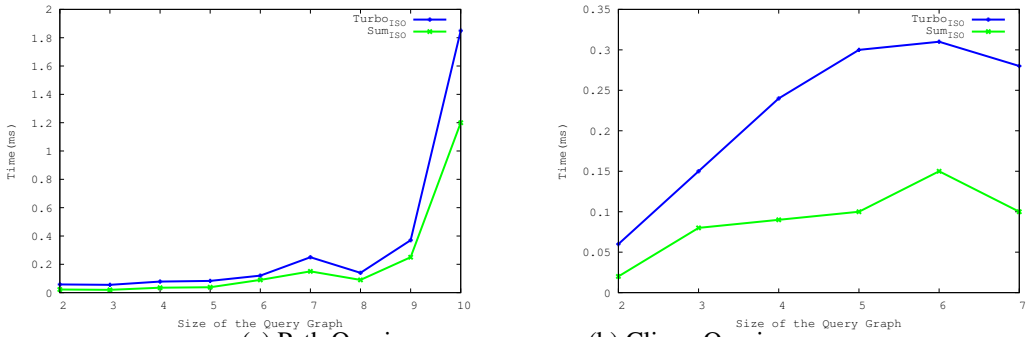
Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., and Wu, Y. (2010). Graph pattern matching: From intractable to polynomial time. *Proc. VLDB Endow.*, 3(1-2):264–275.

Fan, W., Li, J., Wang, X., and Wu, Y. (2012). 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. ACM.

Gallagher, B. (2006). Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 6:45–53.

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

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*.



(a) Path Queries. (b) Clique Queries.

Figure 8: Path and Clique Queries.

Habib, M., Montgolfier, F. D., and Paul, C. (2004). A simple linear-time modular decomposition algorithm for graphs. *Scandinavian Workshop on Algorithm Theory - SWAT*, pages 187–198.

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

Han, W.-S., Lee, J., and Lee, J.-H. (2013). 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. ACM.

He, H. and Singh, A. (2006). Closure-tree: An index structure for graph queries. In *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, pages 38–38.

He, H. and Singh, A. K. (2008). 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. ACM.

Khoo, K. G. and Suganthan, P. N. (2001). Multiple relational graphs mapping using genetic algorithms. pages 727–737.

Lagraa, S. and Seba, H. (2016). An efficient exact algorithm for triangle listing in large graphs. *Data Mining and Knowledge Discovery*, pages 1–20.

Lagraa, S., Seba, H., Khennoufa, R., M'Baya, A., and Kheddouci, H. (2014). A distance measure for large graphs based on prime graphs. *Pattern Recognition*, 47(9):2993 – 3005.

Lee, J., Han, W.-S., Kasperovics, R., and Lee, J.-H. (2013). 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.

Micheli, A. (2009). Neural network for graphs : A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511.

Möhring, R. (1985a). Algorithmic aspect of the substitution decomposition in optimization over relation, set system and boolean function. *Ann. Operations Research*, 4:195–225.

Möhring, R. (1985b). Algorithmic aspects of comparability graphs and interval graphs. *I. Rival. Graphs and Order (D. Reidel)*, pages 41–101.

Quaddoura, R. and Mansour, K. (2010). Classical graphs decomposition and their totally 2010 decomposable graphs. *International Journal of Computer Science and Network Security*, 10:1240–1250.

Shang, H., Zhang, Y., Lin, X., and Yu, J. X. (2008). Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375.

Shasha, D., Wang, J. T. L., and Giugno, R. (2002). Algorithmics and applications of tree and graph searching. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 39–52, New York, NY, USA. ACM.

Spinrad, J. P. (2003). *Efficient Graph Representation*. American Mathematical Society.

Sun, Z., Wang, H., Wang, H., Shao, B., and Li, J. (2012). Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799.

Tedder, M., Corneil, D. G., Habib, M., and Paul, C. (2008). Simpler linear-time modular decomposition via recursive factorizing permutations. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 634–645.

Ullmann, J. R. (1976). An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42.

Yan, X., Yu, P. S., and Han, J. (2004). 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. ACM.

Zhang, S., Li, S., and Yang, J. (2009). 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. ACM.

Zhao, P. and Han, J. (2010). On graph query optimization in large networks. *PVLDB*, 3(1):340–351.

Zhao, P., Yu, J. X., and Yu, P. S. (2007). Graph indexing: Tree + delta >= graph. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 938–949. VLDB Endowment.

Zou, L., Chen, L., Yu, J. X., and Lu, Y. (2008). 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. ACM.