Journal of Computer and System Sciences ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Fast processing of graph queries on a large database of small and medium-sized data graphs

Dipali Pal, Praveen Rao *, Vasil Slavov, Anas Katib

Department of Computer Science and Electrical Engineering, University of Missouri-Kansas City, United States

ARTICLE INFO

Article history:
Received 23 May 2013
Received in revised form 19 February 2016
Accepted 19 March 2016
Available online xxxx

Keywords:
Graph indexing
Query processing
Subgraph matching
Approximate graph matching
Line graphs

ABSTRACT

We propose a new way of indexing a large database of small and medium-sized graphs and processing *exact subgraph matching* (or subgraph isomorphism) and *approximate (full) graph matching* queries. Rather than decomposing a graph into smaller units (*e.g.*, paths, trees, graphs) for indexing purposes, we represent each graph in the database by its graph signature, which is essentially a multiset. We construct a disk-based index on all the signatures via bulk loading. During query processing, a query graph is also mapped into its signature, and this signature is searched using the index by performing multiset operations. To improve the precision of exact subgraph matching, we develop a new scheme using the concept of *line graphs*. Through extensive evaluation on real and synthetic graph datasets, we demonstrate that our approach provides a scalable and efficient disk-based solution for a large database of small and medium-sized graphs.

1. Introduction

Graphs are widely used to model data in a variety of domains such as biology, chemistry, computer vision, and the World Wide Web. As an example, in biology, protein interaction networks and phylogenies can be modeled as graphs. In chemistry, chemical compound structures can be conveniently represented by graph structures. Graphs are also useful in pattern recognition and computer vision, for example, to represent hierarchical image features. On the World Wide Web, social networks naturally fit a graph data model.

Pattern matching over graphs is an important task in a variety of applications and falls under two categories, namely, *subgraph matching* and *entire graph matching*. Further, each category can be classified into exact matching and approximate (or inexact) matching. Exact subgraph matching (or isomorphism) [43,29] has been a problem of interest for several decades and has been widely used in areas such as chemical informatics, circuit design and verification, scene analysis in computer vision, and so forth. One example is the task of substructure matching in chemical databases. By posing a query based on *exact subgraph matching*, we can identify those compounds in a database that contain a particular functional group (*e.g.*, a phenyl group C_6H_5) [17]. Another application of exact subgraph matching is for finding structural motifs in 3D protein structures using protein contact maps [23], which can be represented as labeled graphs.

Approximate graph matching [45] is useful in applications where graphs similar to a query graph are desired. Such a matching is useful for tasks such as similarity searching of chemical compounds [46], comparing phylogenies and biopathway graphs, and so forth.

* Corresponding author.
E-mail addresses: dp244@umkc.edu (D. Pal), raopr@umkc.edu (P. Rao), vgslavov@mail.umkc.edu (V. Slavov), anaskatib@mail.umkc.edu (A. Katib).

http://dx.doi.org/10.1016/j.jcss.2016.04.002
0022-0000/

Subgraph isomorphism is an NP-complete problem [15], for which backtracking techniques [43,20] were developed in the past. Unfortunately, these techniques become intractable in the worst case. In practical applications a large number of graphs exist in the database, and it is infeasible to test for subgraph isomorphism pairwise between a query graph and every data graph. A common approach adopted by most prior techniques is to first *filter* and identify candidate matches, and then *verify* these candidates to retain only the true matches. The key challenge in the filtering stage is to construct an index over graphs, so that a large number of graphs that do not contain a match for a query can be pruned away quickly, thereby reducing the number of actual pairwise tests needed in the verification stage.

Many of the previous approaches for exact subgraph matching queries are main-memory based *i.e.*, they load the index into memory before processing graph queries (*e.g.*, gIndex [49], Tree + Δ [56], GDIndex [47], QuickSI [39], GCoding [57]). A recent technique called FG-index [12,13] maintains part of the index in memory and the rest on disk.

Previous approaches commonly leverage frequent pattern mining to extract features from graphs (*e.g.*, trees, subgraphs) for indexing (*e.g.*, gIndex [49], Tree + Δ [56], QuickSI [39], FG-index [12,13]). GraphGrep [16] and SING [31] use paths in graphs as features but do not use frequent pattern mining. C-tree [21] and GCoding [57] are two other techniques that do not rely on mining to construct indexes on graphs.

Approximate graph matching requires the computation of graph edit distance between graphs. Graph edit distance provides a measure of dissimilarity between two graphs. However, computing graph edit distance between two graphs is computationally expensive and the cost is exponential in the number of vertices in the graphs [37]. Therefore, to process an approximate graph matching query, it would be inefficient to compute the graph edit distance between a query and every graph in the database. A *filter* and *verification* approach seems to be a viable alternative. Several techniques have been proposed for approximate subgraph matching such as SAGA [40], TALE [41], PIS [51], Grafil [50], GADDI [54], APPPSUB [52], GrafD-index [38], SAPPER [55], and NeMa [26]. The authors of APPSUB also proposed a technique called APPFULL [52] for processing approximate full graph matching queries. Among these techniques, SAGA and TALE are disk-based indexing techniques and can work with limited amount of main memory.

In this work, we are particularly interested in dealing with a large database of small and medium-sized data graphs that arise in domains such as chemical informatics [2] and proteomics [1]. The nature of graphs that we tackle with can contain up to a few hundred vertices and a few thousand edges. We aim to build disk based indexes for fast filtering of exact subgraph and approximate graph matching queries. This will allow us to process large datasets wherein the graph indexes cannot fit in main memory. We aim to avoid the use of mining for extracting frequent patterns as this preprocessing step can become very expensive for large, dense graphs. Finally, holistic XML pattern matching approaches (*e.g.*, TwigStack [8]) are regarded to be superior for querying XML documents than those that decompose a tree pattern query into smaller units and process them separately. In a similar vein, we aim to develop a holistic graph pattern matching approach, without decomposing a graph (data or query) into its features during indexing and query processing.

To achieve our aforementioned aims, we propose a system called GiS (**G**raph **i**ndexing via **S**ignatures). We assume that the database D has a large number of *undirected* graphs with or without edge labels. We address two types of queries over D.

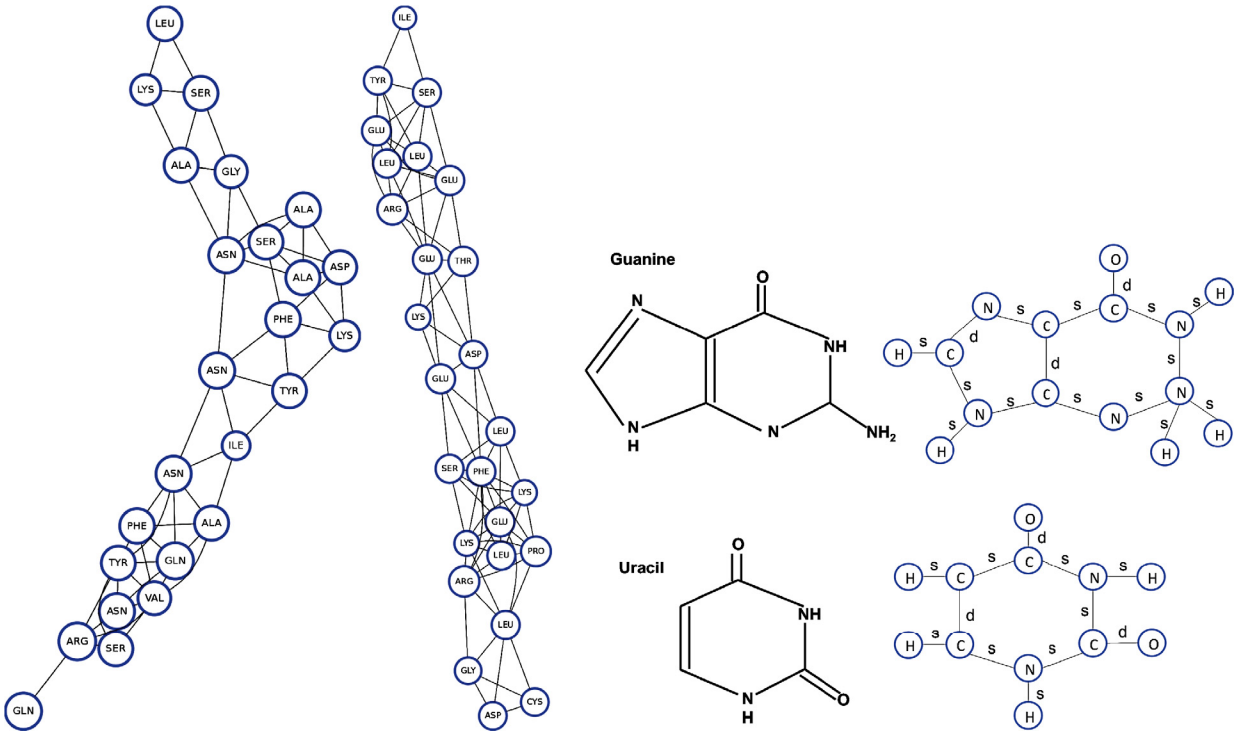
> **Exact subgraph matching:** Given a query graph Q, an exact subgraph matching query finds all graphs in D that contain a subgraph that is isomorphic to Q. (This query is also referred to as a subgraph containment query in prior work.)
> **Approximate (full) graph matching:** Given a query graph Q and a distance threshold d, an approximate graph matching query finds all graphs in D whose edit distance with Q is at most d.

GiS is designed to efficiently process high selectivity queries on graphs such as those shown in Fig. 1. The first kind is shown in Fig. 1(a), which are undirected graphs without edge labels and are called contact map graphs [14]. A contact map graph is used to represent the 3D structure of a protein, where the amino acid residues in the protein are represented by vertices. An edge exists between two residues if the distance between their c_α atoms is below a particular threshold [14]. The second kind is shown in Fig. 1(b). These graphs are undirected graphs with edge labels to model chemical compounds such as Guanine and Uracil. The vertices of each graph denote the atoms in the compound, and the edge labels denote the type of bond between two atoms. One can observe that the contact map graphs in Fig. 1(a) are relatively larger and denser than the graphs for chemical compounds in Fig. 1(b).

Below we summarize the key contributions of our work.

- We propose a new way of representing a graph holistically (*i.e.*, as a whole) by its signature, thereby avoiding its decomposition into smaller units that are separately indexed. A *graph signature* is essentially a multiset that captures the vertex and edge characteristics of a graph.
- To systematically expose more structural properties of a graph that can be captured by a graph signature, and therefore, improve the precision of exact subgraph matching, we propose a new method based on the concept of *line graphs*. Line graph computation can be applied successively to a graph and provides a way to tune the extent of structural information captured by signatures. We study the benefits and tradeoffs of our method.
- We propose an approach for approximate graph matching using signatures on the original graphs. (Line graphs are not used for approximate graph matching.) We study the relationship between graph edit distance and the properties of graph signatures and propose a method for processing approximate graph matching queries.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002



(a) Protein contact map graphs (b) Graph representation of Guanine and Uracil

Fig. 1. Examples of graphs that GiS is designed to process efficiently.

- We show that both exact subgraph matching and approximate graph matching query can be processed by first computing the signature of a query and then by applying multiset operations on the data and query signatures. Unlike GiS, many existing methods decompose a query into features and then process individual features and combine the results.
- To speed up query processing, we develop a disk-based indexing strategy for graph signatures, *i.e.*, multisets. We propose a bulk-loading approach for efficient indexing.
- We conducted extensive performance evaluation of GiS on real-world and synthetic datasets containing small and medium-sized data graphs. The graphs in these datasets had up to a few hundred vertices and a few thousand edges. (The number of distinct vertex labels in these datasets was between 38 to 235.) We conclude that GiS's holistic query processing approach provides an efficient disk-based solution for exact subgraph matching and approximate (full) graph matching on high selectivity queries.

The rest of the paper is organized as follows: Section 2 provides preliminaries in graph theory and multisets; Section 3 describes related work; Section 4 provides the overall architecture of GiS; Section 5 presents graph signatures and line graphs; Section 6 describes how signatures can be used for exact subgraph matching; Section 7 describes approximate graph matching using signatures; Section 8 describes the performance evaluation; and Section 9 describes our conclusions and future work.

A preliminary version of this work appeared in the 20th International World Wide Web Conference [33].

2. Preliminaries

We state the basic terminologies from graph theory literature. A graph G consists of a non-empty finite vertex set $V(G)$ called *vertices*, and a finite set $E(G)$ of unordered pairs of vertices called *edges*. We denote such a graph by $G(V, E)$ or $G = (V, E)$. A simple graph, which we consider in this paper, does not contain multi-edges or self-loops. We assume undirected graphs, where there is no direction assigned to an edge. Edges and vertices may be labeled. We assume that each vertex of a graph is assigned a unique integer id from 1 to n, where n denotes the number of vertices. We use v_i to denote the id of a vertex. Let $l(v_i)$ denote the vertex label of v_i drawn from some domain Δ. Two vertices are *adjacent* if there is an edge joining them. An edge is said to be incident on the vertices that are its end points. We will frequently use (v_i, v_j) to denote an edge between vertex v_i and vertex v_j. Two edges are *adjacent* if they share a common vertex.

A graph G_1 is isomorphic to G_2, if there exists a bijective mapping $f : V(G_1) \rightarrow V(G_2)$ between the vertex sets in G_1 and G_2, and two vertices u_1 and v_1 are adjacent in G_1, if and only if, their corresponding vertices u_2 and v_2 are adjacent in G_2. A subgraph $G_i(V_i, E_i)$ of a graph G has a vertex set $V_i \subseteq V$ and $E_i \subseteq E$. Subgraph isomorphism is a decision

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

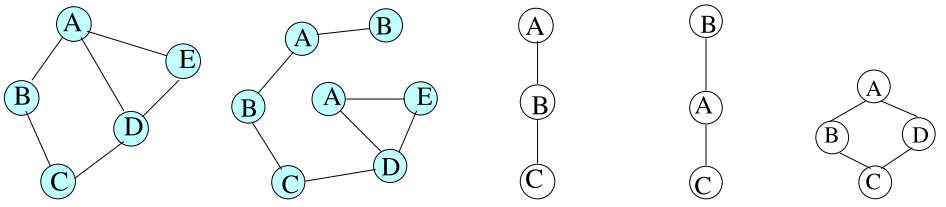


Fig. 2. Examples of data and query graphs.

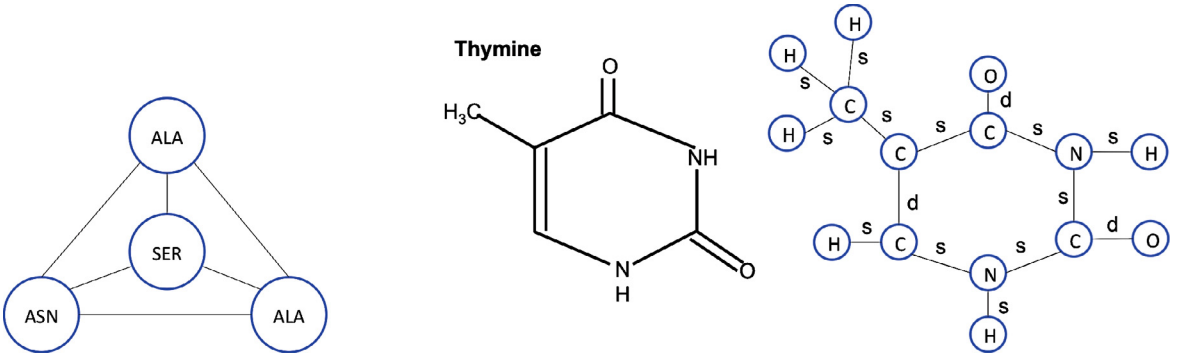


Fig. 3. Examples of query graphs on real datasets.

problem to test if a graph has a subgraph that is isomorphic to a given graph. For labeled graphs, we impose an additional condition that the labels of corresponding vertices in the bijective mapping should also be identical for graph and subgraph isomorphism.

Edit distance between two graphs denotes the minimum cost of transforming one graph to another by applying a sequence of edit operations (*e.g.*, insertion, deletion, and relabel of vertices and edges). In approximate graph matching, we find those graphs that are within a user-specified edit distance from a query graph.

Example 1. Fig. 2 shows two data graphs, namely, $\mathbb{G}_1$ and $\mathbb{G}_2$, and three query graphs, namely, Q_1, Q_2, and Q_3. The domain of vertex labels $\Delta = \{A, B, C, D, E\}$. Let us first consider the task of exact subgraph matching. In $\mathbb{G}_1$, there exists a subgraph that is isomorphic to Q_1. Neither $\mathbb{G}_1$ nor $\mathbb{G}_2$ has a subgraph that is isomorphic to Q_2. Finally, $\mathbb{G}_1$ also has a subgraph that is isomorphic to Q_3. Next, let us consider the task of approximate (full) graph matching and focus on Q_3. Graph $\mathbb{G}_1$ and query Q_3 have an edit distance of three (assuming unit cost for each edit operation). One vertex deletion and two edge deletions are needed to transform $\mathbb{G}_1$ to Q_3. The edit distance between $\mathbb{G}_2$ and Q_3, however, is greater than three.

Example 2. Fig. 3 shows example query graphs on the data graphs shown in Fig. 1. Consider the query in Fig. 3(a) for exact subgraph matching on the protein contact map graphs in Fig. 1(a). This query denotes the desired structural relationship among the amino acids ALA, SER, ASN, and ALA. Only the data graph on the left-hand side of Fig. 1(a) contains a subgraph isomorphic to the query. Consider the query in Fig. 3(b) for approximate (full) graph matching on the chemical compounds in Fig. 1(b). Suppose we want to find the compounds that are similar to Thymine within an edit distance of 7 (assuming unit cost for each edit operation). Uracil shown in Fig. 1(b) is a match for the query as one vertex relabel, three vertex additions, and three edge insertions are needed to transform Uracil to Thymine. Note that Uracil and Thymine belong to a particular family of compounds called Pyrimidine.[1]

Next, we discuss multisets and operations over them. Given a domain $\Delta = \{l_1, \ldots, l_n\}$, a multiset M over Δ can be represented by defining a family of functions $\mu_M(l_i) = f_i$ for each l_i, where f_i denotes the frequency of l_i in M [9]. Next, we present the operators $+$, $-$, $\subseteq$, $\cap$, $\cup$, and $|\cdot|$ over multisets. Consider two multisets M and N. The multiset $L = M + N$ is defined by the family of functions $\mu_L(l_i) = \mu_M(l_i) + \mu_N(l_i)$. The multiset $L = M - N$ is defined by the family of functions $\mu_L(l_i) = max\{0, \mu_M(l_i) - \mu_N(l_i)\}$. The multiset $L = M \cup N$ is defined by the family of functions $\mu_L(l_i) = max\{\mu_M(l_i), \mu_N(l_i)\}$.

[1] https://en.wikipedia.org/wiki/Pyrimidine.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

The multiset $L = M \cap N$ is defined by the family of functions $\mu_L(l_i) = min\{\mu_M(l_i), \mu_N(l_i)\}$. If $L \subseteq M$, then $\forall_{i=1\ to\ n}\{\mu_L(l_i) \leq \mu_M(l_i)\}$. The cardinality of a multiset M denoted by $|M|$ is equal to the number of elements in the multiset, *i.e.*, $\sum_{i=1}^{n} \mu_M(l_i)$.

3. Related work and motivations

We briefly survey existing graph indexing approaches and highlight their differences compared to our approach. We broadly group the approaches into two categories based on the nature of graph matching, namely, exact subgraph matching and approximate subgraph/graph matching. Some approaches belong to both categories.

3.1. Approaches for exact subgraph matching

Many approaches have been developed for efficient processing of exact subgraph matching queries. A number of these approaches differ in what they select as features for indexing the data graphs (*e.g.*, paths, trees, subgraphs). This in turn decides how a query is processed. Some of the approaches enumerate features up to a certain size and others use frequent pattern mining to select the features for indexing. A few approaches like ours transform the entire data graph into a representation that facilitates a compact index structure. Our approach aims to facilitate holistic graph pattern matching technique by avoiding the decomposition of data and query graphs into features for indexing and query processing.

3.1.1. Paths as features for indexing

One of the earliest approaches is GraphGrep [16], which decomposes graphs into paths of up to a certain length l, thereby resulting in an index size exponential in l. More recently, Natale et al. [31] proposed SING for exact subgraph matching on large graphs. SING, a main-memory based approach designed for exact subgraph matching on graphs that are medium-sized or large-sized. SING uses a path up to some length k (along with its lower bound on the frequency of occurrence in a graph) as a feature for indexing. (SING can be considered as a non-holistic approach.) SING also maintains the position of a feature in a graph to speed up the filtering phase and the verification phase. SING supports the finding of all occurrences of an exact subgraph match in the graph database.

3.1.2. Subgraphs as features for indexing

Later, Yan et al. [49] proposed gIndex to reduce the index size by selecting discriminative frequent substructures (*i.e.*, subgraphs) in the data as the indexing feature. GDIndex [47] proposed the use of graph decomposition by maintaining two separate data structures: a directed acyclic graph of unique induced subgraphs in the database and a hash table for the subgraphs. The overhead of computing these decompositions for indexing large graphs can be significant. In the worst case, the index size can be exponential in the maximum graph size. A decomposition approach was also suggested by Messmer and Bunke [29] for efficient subgraph isomorphism detection.

Subsequently, Cheng et al. proposed FG-index [12,13], which uses frequent subgraphs as the features for indexing. It builds two indexing data structures: one is stored in main memory and the other is stored on disk. When a query pattern is a frequent subgraph, then no verification is needed during exact subgraph matching. However, when a query pattern is not frequent, which is the focus of our work, then the subgraphs of the query that are indexed are first found along with data graphs that contain these subgraphs. Edges in the query that are infrequent are used to identify another set of data graphs. Finally, the ids of data graphs are intersected and matches are then verified. Therefore, in this situation, FG-index can be viewed as a non-holistic approach. However, the preprocessing cost via frequent pattern mining could be non-trivial.

3.1.3. Trees as features for indexing

TreePi [53] and Tree + Δ [56] use frequent trees in graphs as indexing features to reduce the index size and construction cost. These approaches require non-trivial preprocessing cost via frequent tree mining. GString [24] proposes a semantic based approach by encoding basic structures in graphs into strings. The authors focus primarily on chemical compounds. Recently, QuickSI [39] was proposed to reduce the cost of the verification phase (*e.g.*, using Ullmann's algorithm [43]) and leverages a feature-based index (*i.e.*, using trees) to speed up the filtering phase. This approach is suitable when queries return a large number of matches and therefore, the processing time is dominated by the verification cost.

3.1.4. Non-mining based approaches

He et al. proposed C-tree [21] to overcome the enumeration overhead of approaches like GraphGrep and gIndex. C-tree organizes graphs into a hierarchical index by computing *graph closures*. The index is constructed by hierarchical clustering. C-tree indexes graph closures but applies different levels of pseudo-isomorphism tests while searching the index to achieve high precision. Zou et al. proposed a scheme called GCoding to map the structure information of graphs into a numerical space [57] using vertex signatures and graph codes. GCoding does not use frequent pattern mining for indexing purposes. While a graph code computed by GCoding captures local tree structures around the neighborhood of vertices, in GiS, we attempt to capture subgraph structures in a graph. (See Section 5.2.) GCoding is implemented using C++ STL and hence, is a main-memory based approach.

Unlike C-tree and GCoding, our work employs the concept of line graphs to capture the structure and labels of the data graphs for efficient processing of exact subgraph matching queries.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

3.2. Approaches for approximate subgraph/graph matching

3.2.1. Approximate subgraph matching

Many approaches have been developed for approximate subgraph matching (*e.g.*, SAGA [40], TALE [41], PIS [51], Grafil [50], GADDI [54], GrafD-index [38], SAPPER [55]). Grafil [50] and PIS [51] were developed for similarity based substructure matching and rely on gIndex, a technique that indexes discriminative frequent subgraphs in the data. Unlike approaches that use frequent subgraphs for indexing a large database of graphs, GADDI [54] proposed a new measure called neighboring discriminative substructure (NDS) distance between the neighboring vertices for indexing a single large graph like a biological network.

SAGA [40] and TALE [41] are disk based approaches for approximate subgraph matching. SAGA is designed for small query graphs and TALE is designed for large query graphs with hundreds of nodes. SAGA is suited for biological data and breaks a graph into fragments of a user-specified size and uses fragments as the indexing units. In TALE, the indexing unit is the neighborhood of each graph node. The neighborhood information captures the local structure around a graph node, and the index size grows linearly with the database size. TALE can be regarded as a non-holistic approach. SAPPER is another technique for approximate subgraph matching on large graphs. Unlike, TALE which is faster but may miss approximate matches, SAPPER returns all matches.

Zeng et al. [52] developed APPSUB for approximate subgraph matching. For each input graph, they compute a star structure on each vertex. A star structure is rooted at a vertex and includes the adjacent vertices and edges [52]. They define three distance measures to lower and upper bound the graph edit distance in polynomial time. During query processing, every graph in the database is scanned and the distance measures are used to process different types of graph matching queries. Bipartite graph matching is required to compute one of the distance measures and requires $\Theta(n^3)$ time, where n is the number of vertices in the data graph.

More recently, Khan et al. [26] proposed a novel technique called NeMa for approximate subgraph matching on large real-life graphs. NeMa adopts a neighborhood vector representation to represent nodes in a large graph and develops a new subgraph matching cost function. It allows both structure and node label mismatches during subgraph matching and employs an inference algorithm to find optimal approximate subgraph matches.

Note that GiS is not designed for approximate subgraph matching.

3.2.2. Approximate (full) graph matching

Few approaches have been proposed to efficiently retrieve data graphs that are similar to the query graphs. One of the earliest approaches was C-tree [21], which builds an index on graph closures. It supports similarity queries (a.k.a. range queries) on small graphs. Later, Zeng et al. [52] developed APPFULL for approximate (full) graph matching using the notion of star structures, which they used for approximate subgraph matching. Due to the lack of indexes, APPFULL may become expensive when the number of graphs is very large.

Similar to APPFULL, GiS aims to support approximate (full) graph matching queries based on graph edit distance. GiS aims to rely on the same disk-based index for both types of matching. Unlike NeMa, GiS does not support node label mismatches.

3.3. Other related approaches

The concept of a line graph has been studied in the context of biological networks. Pereira-Leal et al. [34] used the line graph of a protein interaction network to improve graph clustering. Ucar et al. [42] used the line graph for preprocessing to eliminate redundant false positives from protein–protein interactions before performing clustering. Nacher et al. [30] studied the effect of line graph computation on scale-free networks. Bayir et al. [5] computed the line graph of protein interaction network to remove edges in the graph before computing topological measures. Unlike these approaches, GiS uses line graphs to speed up exact subgraph matching queries.

Recently, Cheng et al. [11] proposed a new approach for processing supergraph queries, wherein the database graphs that are contained by the query graph are output. GiS does not support supergraph queries. Bleco et al. [6] proposed an approach for graph analytics on massive collections of small (directed) graphs with numerical values on the vertices and edges. Their approach decomposes both the data and query graphs into simpler structural elements for efficient query processing. It uses column-oriented storage, bitmap indexes, and materialized views for fast processing of analytical queries. While we focus primarily on undirected graphs in this work, we discuss later in Section 8.10 how GiS can be extended to support directed graphs.

3.4. Key motivations

The key motivations of our work are as follows: When a dataset containing a large number of small/medium-sized data graphs is indexed, the index size can become very large and may not fit in limited amount of main memory. So we aim to build disk based indexes for fast processing of exact subgraph and approximate (full) graph matching queries. We aim to avoid the use of mining for extracting frequent patterns as this preprocessing step can become very expensive for even medium-sized, dense data graphs. Motivated by the success of holistic XML pattern matching approaches

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

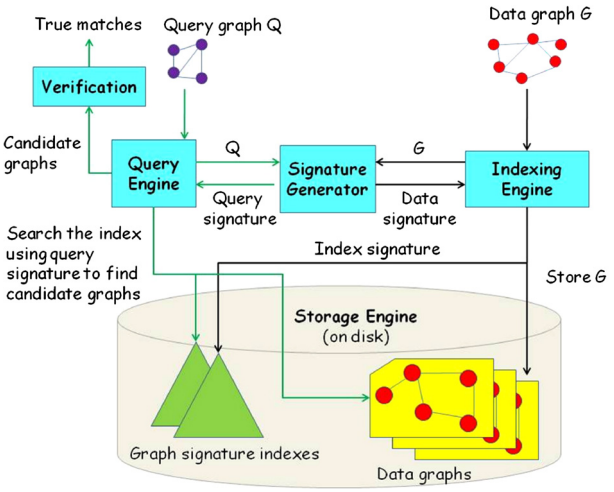


Fig. 4. Architecture of GiS.

(*e.g.*, TwigStack [8]), we aim to develop a holistic graph pattern matching approach, without decomposing a graph (data or query) into its features (*e.g.*, paths, trees, subgraphs) during indexing or query processing. Thus, a query will be processed as a single unit. We believe this will lead to small index sizes, fast index construction, and fast query processing by reducing the cost of searching the index and any post-processing (*e.g.*, computing the partial results of features in a query followed by the intersection of the results as done by a non-holistic approach). We also argue that as the size of a query increases, more features need to be processed using the index, which can lead to increased filtering cost. We aim to employ line graphs on small and medium-sized data graphs for exact subgraph matching. The goal is to improve the precision of exact subgraph matching using our index, which will lead to faster query processing. Another appealing feature is that using line graphs, we can preserve the concept of holistic matching even when the original (data or query) graphs are transformed.

4. Architecture of GiS

We introduce the overall architecture of GiS before delving into the details of its design in Sections 5, 6, 7, and 6.2. The key components of GiS include the *Signature Generator*, *Indexing Engine*, *Query Engine*, and *Storage Engine* as shown in Fig. 4. The Signature Generator constructs a signature of a graph, which is essentially a multiset representation of the graph and captures the vertex and edge characteristics of the graph. The Signature Generator can employ the idea of line graphs to tune the extent of structural information captured by the signature of a graph. Further, the use of line graphs facilitates holistic query processing without breaking a query into smaller units.

To index a graph, the Indexing Engine obtains the signature of a graph from the Signature Generator and then indexes the signature in a disk-resident hierarchical index. The index, which we refer to as graph signature index, speeds up the filtering phase during query processing. The leaf nodes of the graph signature index contain the graph signatures and the ids of the graphs. The Storage Engine maintains the indexes and the data graphs on disk. A user can pose either an exact subgraph matching query or an approximate (full) graph matching query.

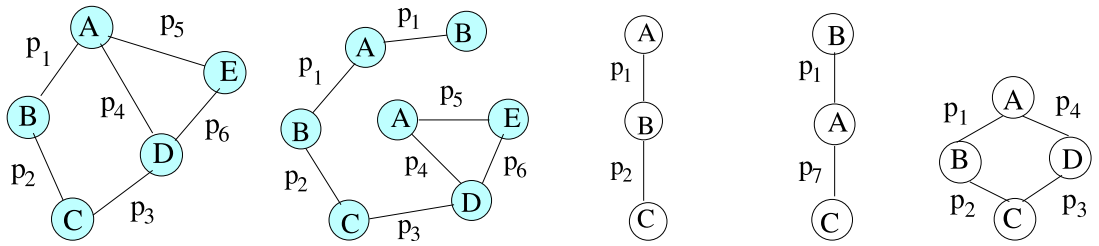
The Query Engine obtains the signature of a query graph from the Signature Generator and then uses it as a key to search through the signature index. Candidates graph ids are obtained during the filtering phase. Finally, the verification step is carried out on the candidate graphs to obtain the true matches for the query. Note that the way the signature index is searched during the filtering phase depends on the nature of the query. For example, subset operation between multisets (or signatures) is utilized for exact subgraph matching; multiset difference operation is utilized during approximate graph matching.

5. A new signature representation of graphs

In this section, we first introduce the key idea of graph signatures in GiS that forms the basis for indexing a large database of graphs and processing graph queries efficiently. Then, we present the concept of line graphs to tune the extent of structural information captured by a graph signature. This concept is specifically designed for exact subgraph matching.

GiS supports graphs that have vertex labels or edge labels or both. Unlabeled graphs, however, are not currently supported. In this section, we first assume that the graphs have only vertex labels. Later, we discuss how edge labels in graphs can be handled by GiS.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002



(a) Graph $\mathbb{G}_1$ (b) Graph $\mathbb{G}_2$ (c) Query Q_1 (d) Query Q_2 (e) Query Q_3

Fig. 5. The data and query graphs in Fig. 2 are shown again. The annotation aside each edge is the hash value of its ordered-label pair.

5.1. Signature of a graph

Consider an edge (v_i, v_j) in a graph. Without loss of generality, we define a function $lex(l(v_i), l(v_j))$ that returns the input arguments (*i.e.*, vertex labels) in lexicographic order. We call the output of $lex(\cdot)$ the *ordered-label pair* of an edge.

Algorithm 1: Construct a signature for a graph

```
1  Global S ← ∅ /* initialize signature */
proc constructSignature(vertex v)
2  foreach edge (v, v_j) not visited so far do
3  |   mark edge (v, v_j) as visited /* Depth first search */
4  |   p ← h(lex(l(v), l(v_j))) /* compute hash value */
5  |   add p to partial signature S
6  |   constructSignature(v_j)
   end foreach
7  return
endproc
```

A *graph signature* is essentially a multiset of hash values computed over each ordered-label pair of its edges. Algorithm 1 shows the steps involved in computing a graph signature. Starting from any vertex, the graph is traversed in depth first order. Each time an unvisited edge is reached, it is marked as visited, and the ordered-label pair of the edge is constructed using $lex(\cdot)$. Each ordered-label pair is hashed to an integer using a hash function $h(\cdot)$ and the hash value is added to the partial signature S. When the traversal is complete, the graph signature, denoted by $sig(\cdot)$ hereinafter, is essentially a multiset of hash values for all ordered-label pairs of a graph.

The hash function $h(\cdot)$ that we use is based on Rabin's fingerprinting technique [35], which has been well studied both in theory and practice and can also be computed efficiently. Essentially, we represent an ordered-label pair using its bit string representation, which corresponds to a polynomial in Galois Field 2 with coefficients 0 or 1. Let us denote this polynomial by p. We define $h(\cdot) = p \mod r$, where r is an irreducible polynomial in Galois Field 2 picked at random. This hash function has low degree of collision. Suppose k bit fingerprints are computed using an irreducible polynomial of degree $k-1$, then given two ordered-label pairs x and y, s.t. $x \neq y$, $P(h(x) = h(y)) \leq \frac{max(|x|,|y|)}{2^{k-1}}$, where $|\cdot|$ denotes the number of bits [7]. Suppose we chose r to be an irreducible polynomial of degree 31 and generated 32 bit hash values. Let each ordered-label pair require at most 1024 bits. Then the probability of collision is less than 2^{-21}. (In fact, we required less than 1024 bits to store each ordered-label pair in our experiments.)

Remark 1. The $lex(\cdot)$ function ensures that irrespective of the starting vertex chosen for depth-first traversal during signature construction, the constructed signature is the same.

Example 3. Let us consider the example graphs introduced in Fig. 2. The same graphs $\mathbb{G}_1$ and $\mathbb{G}_2$ are shown in Figs. 5(a) and 5(b), respectively, with a value against each edge denoting the hash value of its ordered-label pair. Suppose the ordered-label pairs (A, B), (B, C), (C, D), (A, D), (A, E), and (D, E) are hashed to integers p_1, p_2, p_3, p_4, p_5, and p_6, respectively, using $h(\cdot)$. Using Algorithm 1, $sig(\mathbb{G}_1) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ and $sig(\mathbb{G}_2) = \{p_1, p_1, p_2, p_3, p_4, p_5, p_6\}$. For queries Q_1, Q_2, and Q_3 shown in the figure, we construct their signatures also using Algorithm 1. Thus, $sig(Q_1) = \{p_1, p_2\}$, $sig(Q_2) = \{p_1, p_7\}$, and $sig(Q_3) = \{p_1, p_2, p_3, p_4\}$.

We will frequently refer to the signature of a data graph and a query as *data signature* and *query signature*, respectively.

5.2. Line graphs: a method to tune graph signatures

The above scheme for signature construction solely considers adjacent vertices and their labels. It ignores other structural properties such as the adjacency between edges in the graph, which can affect the precision of exact subgraph matching.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

We propose a new method that allows GiS to tune the extent of structural information captured by graph signatures based on the concept of line graphs (a.k.a. interchange graphs) [48].

Definition 1. A line graph of a graph $G = (V, E)$, denoted by $L(G) = (V_L, E_L)$, is a graph whose vertex set V_L contains one vertex for every edge in G. The edge set E_L is constructed as follows: two vertices in $L(G)$ are adjacent, if and only if, their corresponding edges in G are adjacent *i.e.*, share a vertex.

Although the standard definition of a line graph does not specify how the vertex labels of a line graph should be represented, we follow a precise way, because this affects the graph signatures that will be constructed. For an edge (v_i, v_j) in G, we label the corresponding vertex in $L(G)$ by $lex(l(v_i), l(v_j))$.

Algorithm 2 outlines the steps involved in constructing $L(G)$ from G. It has two phases: In the first phase, the vertex set of $L(G)$ is created by traversing G in depth-first order (Lines 3-10). In the second phase, we examine the adjacency between edges in G and create the edge set of $L(G)$ (Lines 11-20).

Algorithm 2: Line graph construction

```
proc constructLineGraph(V(G), E(G))
1   An array of lists (or sets) called adj is maintained, where each array entry is indexed using a vertex id
2   Let V_L and E_L denote the vertex set and edge set L(G)
    /* Phase 1 */
3   foreach edge (v_i, v_j) ∈ E(G) do
4       adj[v_i] ← adj[v_i] ∪ v_j
5       adj[v_j] ← adj[v_j] ∪ v_i
6       if v_i < v_j then
7           V_L ← V_L ∪ (v_i, v_j)
8           l((v_i, v_j)) ← lex(l(v_i), l(v_j))
        else
9           V_L ← V_L ∪ (v_j, v_i)
10          l((v_j, v_i)) ← lex(l(v_j), l(v_i))
        end if
    end foreach
    /* Phase 2 */
11  for each vertex v_i ∈ V(G) do
12      for j = 0; j < adj[v_i].size − 1; j++ do
13          for k = j + 1; k < adj[v_i].size; k++ do
14              if v_i < adj[v_i][j] then
15                  v_L1 ← (v_i, adj[v_i][j])
                else
16                  v_L1 ← (adj[v_i][j], v_i)
                end if
17              if v_i < adj[v_i][k] then
18                  v_L2 ← (v_i, adj[v_i][k])
                else
19                  v_L2 ← (adj[v_i][k], v_i)
                end if
20              E_L ← E_L ∪ (v_L1, v_L2)
            end for
        end for
    end for
endproc
```

The signature of $L(G)$ is constructed using Algorithm 1. The ordered-label pair of each edge in the line graph is computed using the function $lex(\cdot)$ and is then hashed to an integer using $h(\cdot)$ during signature construction. Note that the signature of a line graph contains the hash values of the ordered-label pairs in it.

Example 4. $L(\mathbb{G}_1)$ and $L(\mathbb{G}_2)$ in Figs. 6(a) and 6(b) denote the line graphs of $\mathbb{G}_1$ and $\mathbb{G}_2$ in Figs. 5(a) and 5(b), respectively. Suppose the ordered-label pairs of $L(\mathbb{G}_1)$ and $L(\mathbb{G}_2)$ are hashed to integers $r_0, \ldots, r_9$. For example, let the ordered-label pair ((A, B), (B, C)) be hashed to r_1. The signatures $sig(L(\mathbb{G}_1)) = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9\}$ and $sig(L(\mathbb{G}_2)) = \{r_0, r_1, r_2, r_3, r_4, r_7, r_8, r_9\}$. Similarly, for the line graphs of queries shown in Figs. 6(a), 6(b), and 6(c), $sig(L(Q_1)) = \{r_1\}$, $sig(L(Q_2)) = \{r_{10}\}$, and $sig(L(Q_3)) = \{r_1, r_2, r_6, r_9\}$.

A line graph of a graph provides a systematic way to expose more structural properties of a graph that can be captured during signature construction: $L(G)$ explicitly provides the information about the adjacency of edges in G. While a line graph provides a way to tune the extent of structural information captured by a graph signature, the trade-off is that the length of signature increases in most cases. Given a graph $G = (V, E)$ and its $L(G) = (V_L, E_L)$, $|V_L| = |E|$ and $|E_L| = \sum_{i=1}^{|V|} \binom{deg(v_i)}{2}$, where $deg(v_i)$ is the degree of a vertex v_i in G. The time complexity of constructing a line graph is $O(|V_L| + |E_L|)$.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

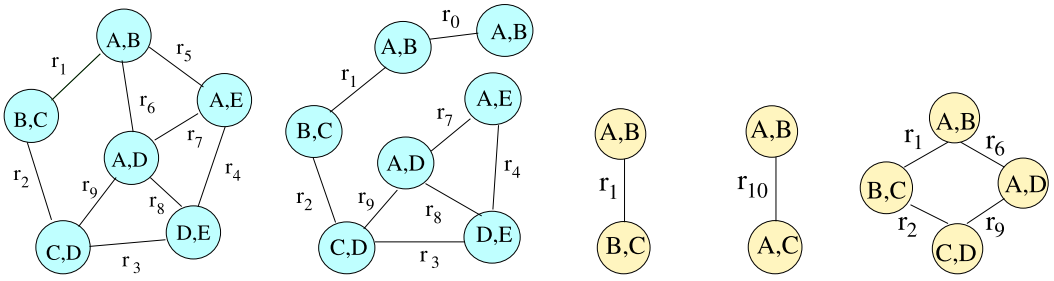


Fig. 6. Line graphs of the data and query graphs.

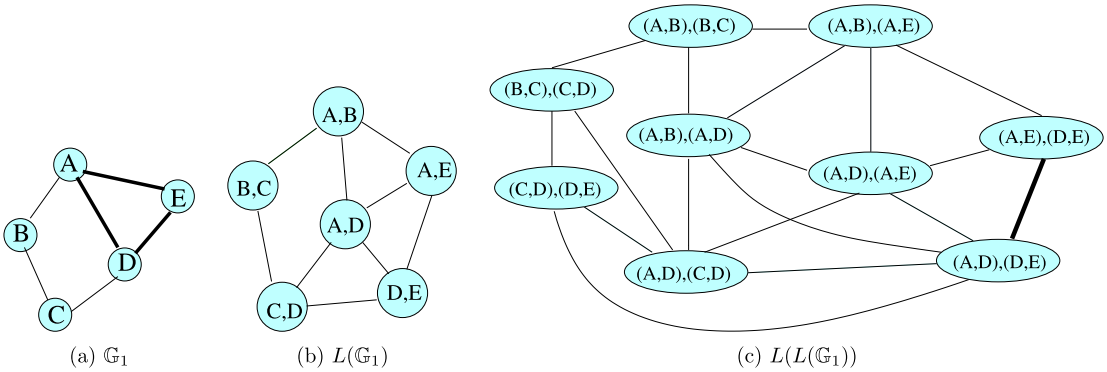


Fig. 7. Successive application of line graph computation on $\mathbb{G}_1$ is shown. Note that the signature of a line graph only contains the hash values of the ordered-label pairs, and the vertex labels are not explicitly stored.

Note that there are two exceptions as shown by Rooij et al. [44]. If a graph is a path graph (*i.e.*, contains only one path) with path length n, then its line graph is a path graph with path length $n-1$. In this case, the size of the signature will decrease as fewer edges will be considered. If a graph is a cycle graph, then its line graph is also a cycle graph. In this case, the size of the signature remains the same.

5.3. Successively applying line graph computation

Interestingly, line graph computation is similar to a composable function. We can successively apply the line graph computation n times on an input graph. We denote this operation and the resulting graph by $L^n(G)$, where G is the input graph. For example, $L^3(G)$ is equivalent to $L(L(L(G)))$. If $n=0$, $L^0(G)=G$. We simply refer to each $L^i(G)$ as a line graph in our subsequent discussions. Except for path graphs, all other input graphs will yield a graph for $L^n(\cdot)$ as $n \to \infty$.

Our main purpose of using $L^n(G)$ is to tune the extent of structural information captured by the signature for G. To compute the signature of $L^n(G)$, we apply Algorithm 1 by hashing the ordered-label pairs in it, as we did for the original graph G. Note that we do not need to store the vertex labels of a line graph. The size of the signature of a line graph grows with the number of edges in the line graph and does not depend on the size of the vertex labels. (A more detailed discussion is provided in Section 6.5.)

Example 5. Fig. 7 shows the successive application of line graph computation on $\mathbb{G}_1$. The ordered-label pairs (output by $lex(\cdot)$) during signature construction of $L(L(\mathbb{G}_1))$ are: ((A, B), (B, C)), ((B, C), (C, D)), ((A, B), (A, E)), ((A, B), (B, C)), ((A, B), (A, D)), ((A, B), (B, C)), ((B, C), (C, D)), ((C, D), (D, E)), ((A, D), (C, D)), ((B, C), (C, D)), ((A, D), (C, D)), ((C, D), (D, E)), ((A, D), (D, E)), ((C, D), (D, E)), ((A, D), (D, E)), ((A, E), (D, E)), ((A, D), (A, E)), ((A, D), (D, E)), ((A, B), (A, D)), ((A, D), (A, E)), ((A, B), (A, D)), ((A, D), (D, E)), ((A, B), (A, D)), ((A, D), (C, D)), ((A, B), (A, D)), ((A, B), (A, E)), ((A, B), (A, E)), ((A, E), (D, E)), ((A, D), (A, E)), ((A, E), (D, E)), ((A, D), (A, E)), ((A, D), (C, D)), ((A, B), (A, E)), ((A, D), (A, E)), and ((A, D), (C, D)), ((A, D), (D, E)). These are hashed, and thus $sig(L(L(G)))$ will contain 18 hash values.

With successive application of line graph computation, an edge in $L^n(G)$ represents a fingerprint of some (connected) subgraph structure in the original graph. Thus, when $sig(L^n(G))$ is computed, it captures richer structural components in G than $sig(G)$. Note that this notion of fingerprint does not have a one-to-one correspondence with a subgraph. As a result, we cannot always reconstruct the actual subgraph from this fingerprint in $L^n(G)$. For example, if we have an edge in $L^2(G)$ with vertices '(A, A), (A, A)' and '(A, A), (A, A)', the corresponding subgraph may or may not be a cycle graph.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

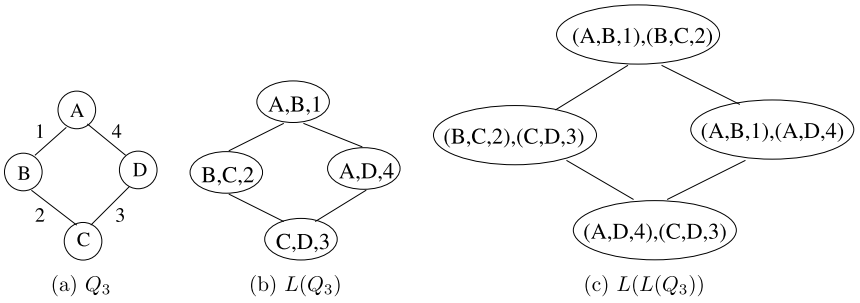


Fig. 8. Line graph computation on an edge labeled graph.

Example 6. In Fig. 7, the thick edge in $L(L(\mathbb{G}_1))$ has vertex labels '(A, D), (D, E)' and '(A, E), (D, E)'. This edge is a fingerprint of a subgraph shown with thick edges (A, D), (D, E), and (A, E) in the original graph $\mathbb{G}$.

Next, we point out some special cases of line graph computations in the context of GiS.

Remark 2. For a path graph G of length n, its graph signature can be constructed for $L^i(G)$, where $0 \leq i < n$, because $L^n(G)$ yields a graph with just one vertex.

Remark 3. Suppose we want to apply line graph computations n times over each graph in a database. If the database contains path graphs whose signatures cannot be computed for the chosen value of n, then a simple way is to separate out such path graphs and process them separately.

5.4. Edge labeled graphs

GiS can support edge labeled graphs. The signature construction process is modified as follows. Rather than constructing an ordered-label pair of an edge, we construct an ordered-label triple of an edge. For example, given an edge (u, v) with edge label e, we construct an ordered-label triple $(lex(l(u), l(v)), e)$. The ordered-label triple is then hashed during signature generation.

Suppose we compute the line graph of an edge labeled graph. The vertex labels of the line graphs are ordered triples that include the edge labels of the original graph. The line graph itself does not contain edge labels. Successive application of line graph computations will result in graphs without edge labels. An example is shown in Fig. 8.

6. Exact subgraph matching using signatures

We begin by presenting a key property of graph signatures that forms the basis for exact subgraph matching. We then describe the construction of an index on the data signatures and how to efficiently process exact subgraph matching queries using this index. Finally, we present a detailed cost analysis of signatures, discuss the benefits and tradeoffs of using line graphs, and highlight the differences between GiS and prior approaches.

6.1. Key property of graph signatures

Suppose we construct the signatures of a data graph and a query graph. We assume that a query graph has at least 2 nodes.[2] We establish a necessary condition that forms the basis for processing exact subgraph matching queries. We state the following theorem.

Theorem 1. *Given a data graph G and a query graph Q, if Q is a subgraph of G, then $sig(Q) \subseteq sig(G)$.*

Proof. By definition, if Q is a subgraph of G, then every edge of Q appears in G. While applying Algorithm 1 on Q, all the ordered-label pairs in Q were also considered during the signature construction for G. Thus each hash value p that was added to the signature $sig(Q)$ (Line 5 in Algorithm 1), was also added to the signature $sig(G)$. Therefore, all the hash values in $sig(Q)$ are present in $sig(G)$, and hence $sig(Q) \subseteq sig(G)$. □

The intuition is as follows. If a query is a subgraph of a data graph, then its edge set is a subset of the edge set of the data graph. Therefore, the hash values that appear in the signature of a query will definitely appear in the signature of the data graph.

[2] It is straightforward to process queries with one vertex, for example, by building an inverted index on the vertex labels.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Example 7. Consider the data and query graphs in Fig. 5. Q_1 is a subgraph of $\mathbb{G}_1$ and $\mathbb{G}_2$. Recall that $sig(\mathbb{G}_1) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $sig(\mathbb{G}_2) = \{p_1, p_1, p_2, p_3, p_4, p_5, p_6\}$, and $sig(Q_1) = \{p_1, p_2\}$. Indeed, $sig(Q_1) \subseteq sig(\mathbb{G}_1)$ and $sig(Q_1) \subseteq sig(\mathbb{G}_2)$. For Q_2, $sig(Q_2) = \{p_1, p_7\}$. Clearly, $sig(Q_2) \nsubseteq sig(\mathbb{G}_1)$ and $sig(Q_2) \nsubseteq sig(\mathbb{G}_2)$. This is due to the value p_7 in $sig(Q_2)$. Indeed, Q_2 is neither a subgraph of $\mathbb{G}_1$ nor $\mathbb{G}_2$.

Suppose we apply line graph computation once on a data graph and a query graph. We state the following theorem that forms a necessary condition for exact subgraph matching queries using line graphs.

Theorem 2. *Suppose G and Q denote a data graph and a query graph, and their line graphs $L(G)$ and $L(Q)$ have at least two vertices each. If Q is a subgraph of G, then $sig(L(Q)) \subseteq sig(L(G))$.*

Proof. We first show that if Q is a subgraph of G, then $L(Q)$ is a subgraph of $L(G)$. Let us focus on the vertex set of $L(Q)$ and $L(G)$. Without loss of generality, consider an edge (v_{qi}, v_{qj}) in Q. Because Q is a subgraph of G, there exists an edge with labels $L(v_{qi})$ and $L(v_{qj})$ in G. Let (v_{gk}, v_{gl}) denote such an edge in G. Edge (v_{qi}, v_{qj}) has a vertex in $L(Q)$ according to the definition of a line graph. Similarly, (v_{gk}, v_{gl}) has a vertex in $L(G)$. Further their vertex labels are the same because $lex(l(v_{qi}), l(v_{qj})) = lex(l(v_{gk}), l(v_{gl}))$.

The above condition is true for every edge in Q. Therefore the vertex set of $L(Q)$ is a subset of the vertex set of $L(G)$.

Let us focus on the edge set of $L(Q)$ and $L(G)$. Without loss of generality, suppose edges (v_{qi}, v_{qj}) and (v_{qi}, v_{qk}) are adjacent in Q. Since Q is a subgraph of G, there exists a corresponding pair of edges in G say (v_{gl}, v_{gm}) and (v_{gl}, v_{gn}) that are adjacent. Then $l(v_{qi}) = l(v_{gl})$, $l(v_{qj}) = l(v_{gm})$, and $l(v_{qk}) = l(v_{gn})$. In $L(Q)$, there exists an edge $(lex(v_{qi}, v_{qj}), lex(v_{qi}, v_{qk}))$. In $L(G)$, there exists an edge $(lex(v_{gl}, v_{gm}), lex(v_{gl}, v_{gn}))$. This edge in $L(Q)$ has the same vertex labels as that of the edge in $L(G)$. (This is because $l(v_{qi}) = l(v_{gl})$, $l(v_{qj}) = l(v_{gm})$, and $l(v_{qk}) = l(v_{gn})$.) This is true for every edge in $L(Q)$, and therefore the edge set of $L(Q)$ is a subset of the edge set of $L(G)$.

Hence, it is proved that $L(Q)$ is a subgraph of $L(G)$. Now we can apply the ideas in Theorem 1 to show that $sig(L(Q)) \subseteq sig(L(G))$. □

Example 8. In this example, we demonstrate the benefit of using line graphs for exact subgraph matching. Consider the data and query graphs shown in Fig. 5. For query Q_3, $sig(Q_3) = \{p_1, p_2, p_3, p_4\}$. Q_3 has a subgraph match in $\mathbb{G}_1$ but not in $\mathbb{G}_2$. Theorem 1 holds for $sig(Q_3)$ and $sig(\mathbb{G}_1)$. It also holds for $sig(Q_3)$ and $sig(\mathbb{G}_2)$. This means $\mathbb{G}_2$ will be treated as having a match for Q_3, which is in fact a false positive. By constructing line graphs, we can avoid this situation. The line graphs of $\mathbb{G}_1$, $\mathbb{G}_2$, and Q_3 are shown in Fig. 6. (Their signatures are shown in Example 4.) Theorem 2 holds for $sig(L(Q_3))$ and $sig(L(\mathbb{G}_1))$ but not for $sig(L(Q_3))$ and $sig(L(\mathbb{G}_2))$, because $sig(L(Q_3)) \nsubseteq sig(L(\mathbb{G}_2))$. Therefore, only $\mathbb{G}_1$ is treated as having a match for Q_3 but not $\mathbb{G}_2$.

We generalize Theorems 1 and 2 as follows.

Theorem 3. *Suppose G and Q denote a data graph and a query graph. Let n (≥ 0) denote the number of line graph computations on the data and query graphs. Suppose $L^n(G)$ and $L^n(Q)$ have at least two vertices. If Q is a subgraph of G, then $sig(L^n(Q)) \subseteq sig(L^n(G))$.*

Proof. We prove the theorem by Induction on the number of line graph computations. Let $P(k)$ denote the proposition that *if Q is a subgraph of G, then $sig(L^k(Q)) \subseteq sig(L^k(G))$*. Note that $1 \leq k \leq n$.

(1) Basis of induction: $P(0)$ is true. If Q is a subgraph of G, then $sig(L^0(Q)) \subseteq sig(L^0(G))$. This was proved in Theorem 1.

(2) Induction hypothesis: Assume that $P(i)$ is true for $1 \leq i \leq k$. We will show that $P(k+1)$ is true. Now $L^{k+1}(Q) = L(L^k(Q))$ and $L^{k+1}(G) = L(L^k(G))$. If $L^k(Q)$ is a subgraph of $L^k(G)$, then $L^{k+1}(Q)$ is a subgraph of $L^{k+1}(G)$ (by Theorem 2). By induction hypothesis, if Q is a subgraph of G, then $L^k(Q)$ is a subgraph of $L^k(G)$. By transitivity, if Q is a subgraph of G, then $L^{k+1}(Q)$ is a subgraph of $L^{k+1}(G)$. Therefore, $sig(L^{k+1}(Q)) \subseteq sig(L^{k+1}(G))$. Thus $P(k+1)$ is true. □

Because the above theorem provides a necessary condition for an exact subgraph match, GiS finds a superset of true matches. False positives can occur and can be discarded during the verification phase by applying a subgraph isomorphism algorithm [43]. The recall, however, is always one; there are no false dismissals.

6.2. The graph signature index

We propose a disk-based index called the Graph Signature Index to organize the data signatures. This enables us to quickly test the necessary condition for exact subgraph matching (Theorem 3) on a large number of data signatures. Because we are dealing with signatures, which are multisets, and perform operations on them, prior techniques for set indexing cannot be employed directly (*e.g.*, RD-tree [22]).

Our index is similar to an R-tree [18] in the sense that it hierarchically groups similar signatures together, just like an R-tree that hierarchically groups nearby rectangles together. Each non-leaf index node contains (sig, ptr) entries where sig denotes a multiset, and ptr denotes a reference to a child index node. Each leaf node also contains (sig, ptr) entries where

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

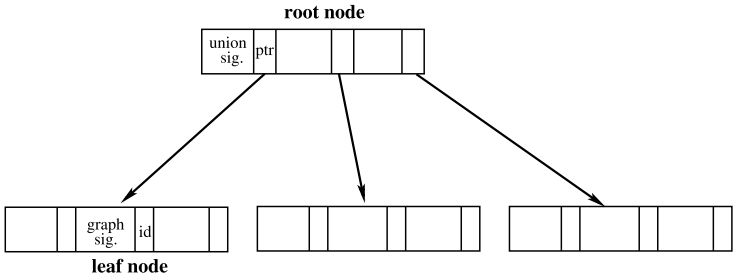


Fig. 9. An example of the Graph Signature Index with two levels.

sig denotes a graph signature, and *ptr* denotes a graph id.[3] The multiset in a non-leaf node entry denotes *the union of the signatures* in the child node pointed by the entry. Fig. 9 shows an illustration of the Graph Signature Index with the root and leaf nodes.

Algorithm 3: Bulk loading the Graph Signature Index

Global: int i = 1;
proc *bulkLoad*($DList$, k, f)
/* $DList$ - signature list; k - a positive integer; f - fanout */
1 Suppose *idx* denotes the index position of a signature s in $DList$. Create $SList$ by sorting $(idx, |s|)$ for each $s \in DList$
2 *split*($DList$, k, f, $SList$)
3 Create upper level index nodes with increasing node ids containing k (sig, ptr) entries (or less for the last index node) by grouping k lower level nodes in the order of their node ids. Each sig value in a node's entry is the union of the signatures in the lower level node.
4 Repeat Line 3 till the upper level has only one index node *i.e.*, root.
endproc
proc *split*($DList$, k, f, $SList$)
5 Among the k highest cardinality signatures in $sList$ find the pair that is most dissimilar
6 Let s_A and s_B denote the two seed signatures from groups A and B
7 Let s_{UA} and s_{UB} denote the union of the signatures in each group
8 **foreach** $s \in DList$ **do**
9 if $\frac{|s \cap s_{UA}|}{|s \cup s_{UA}|} > \frac{|s \cap s_{UB}|}{|s \cup s_{UB}|}$ **then**
10 add s to group A and update s_{UA}
else
11 if $\frac{|s \cap s_{UA}|}{|s \cup s_{UA}|} < \frac{|s \cap s_{UB}|}{|s \cup s_{UB}|}$ **then**
12 add s to group B and update s_{UB}
else
13 add s to the group with fewer signatures and update its union
end if
end if
end foreach
14 **if** *A has more than f signatures* **then**
15 create $SList_A$ for signatures in A
16 split(A, k, f, $SList_A$)
else
17 create leaf index node i containing the signatures in A
18 $i \leftarrow i + 1$
end if
19 **if** *B has more than f signatures* **then**
20 create $SList_B$ for signatures in B
21 split(B, k, f, $SList_B$)
else
22 create leaf index node i containing the signatures in B
23 $i \leftarrow i + 1$
end if
endproc

Next, we propose a bulk-loading algorithm to speed up the indexing process rather than inserting one signature at a time into the index. Suppose we compute $sig(L^n(\cdot))$ on each data graph, where $n \geq 0$. Algorithm 3 outlines the steps involved. Given a group of data signatures, we recursively split the group into two groups, until each group has at most f signatures, where f is the desired node fanout. For a group, we start by picking two dissimilar seeds. Similarity between two

[3] Because signatures are of variable lengths, each entry in the index node can instead contain a $(recid, ptr)$, where the *recid* denotes the id of the signature stored in a heap file.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

signatures is defined as follows: Given two signatures S_1 and S_2, their similarity is given by $\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$. Rather than computing the similarity over all possible pairs to select the seeds, we only examine the k highest cardinality signatures. The intuition is that if two signatures with high cardinalities share the least number of elements, then the remaining signatures may be more likely to be similar to either one of them.

Each group is initialized with a seed signature and the union ($\cup$) of the signatures in each group is maintained. Each remaining graph signature is assigned to the group whose union of signatures has the highest similarity with the graph signature. In case of ties, we assign the signature to the group with fewer signatures. If a group has more than f signatures, it is split again. Otherwise, the group forms a leaf node. Each time a leaf is created, it is assigned an id one greater than the id assigned to a previously created leaf node. Once all the leaf index nodes are created, we are ready to create the upper level nodes in a bottom-up fashion.

Starting with the node with the smallest id, we group f lower level nodes in the order of their ids to create an upper level node. Each entry contains the union of the signatures in a lower level node and the node's id. The upper level nodes are also assigned increasing ids as they are created. We continue creating upper level nodes until only one upper level node is required, *i.e.*, the root of the index. The ids assigned to the index nodes resemble a reverse of breath first search ordering. (The design of the index in GiS draws inspiration from the *psi*X system [36].)

The following remark highlights the pruning property of the index.

Remark 4. Suppose sig_p denotes the signature value in a node's entry and sig_c denotes the signature value in the child node pointed to by the node's entry. If a query signature is a subset of sig_c, then the query signature is also a subset of sig_p. This property allows us to prune the search space of signatures during query processing.

6.3. Processing an exact subgraph matching query

To process an exact subgraph matching query Q, we compute $sig_Q = sig(L^n(Q))$ and search the index as follows: Starting from the root node of the index, at each node, we test if $sig_Q \subseteq sig$ in each (sig, ptr) entry in the node. If the test succeeds and the node is not a leaf node, then we traverse the child pointed by ptr and continue the search. However, if the subset test succeeds and the node is a leaf node, we output ptr as a candidate match. Algorithm 4 outlines the steps involved. Candidate matches are tested during the verification phase to discard false matches [43].

Algorithm 4: Exact subgraph matching

```
proc exactSubgraph(nodeId, sig_Q)
  /* nodeId - index node id; sig_Q - query signature; */
1 (nodetype, S) ← fetchNode(nodeId) /* read node */
2 foreach (sig, ptr) in S do
3   if sig_Q ⊆ sig then
4     if nodetype = LEAF then
5       Output ptr as a candidate match
      else
6       exactSubgraph(ptr, sig_Q) /* Traverse child */
      end if
    end if
  end foreach
endproc
```

6.4. Disk-based implementation of the index

We can conveniently implement the Graph Signature Index on disk using Oracle Berkeley DB [3]. Berkeley DB provides $put()$ and $get()$ APIs to store/insert and retrieve variable length records, respectively. Each record is assigned a recid. During bulk loading, we use these APIs to store the index nodes. When the signatures belonging to a leaf node are determined, we can create a new Berkeley DB file and store each (sig, ptr) entry in the node as a variable length record in that file. Similarly, when a non-leaf/root node is constructed, it can be stored in a separate Berkeley DB file. Each (sig, ptr) entry in the node is stored as a variable length record in that file. Berkeley DB provides a configurable buffer pool cache to speed up disk reads and writes. Thus, we can leverage the efficient implementation and performance of Berkeley DB to maintain the index.

One may wonder if our index can support updates. While we target applications where the graphs are not frequently updated, it is possible to extend the current design to either insert a new data signature or replace an existing signature. Suppose a new graph signature sig_n with id id_n is to be inserted into the index with say two levels. We read the root node of the index and identify the (sig, ptr) entry whose sig has the highest similarity with the signature of the new graph. We pick the child node pointed by ptr and insert (sig_n, id_n) as a record into the leaf node. We then compute $sig \cup sig_n$ in linear time as the signatures are kept sorted. Using the same recid of (sig, ptr), we can reinsert $((sig \cup sig_n), ptr)$ into the node. This will overwrite the previous record value.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 1
Cost of multiset operations.

Operations	Time
$S_1 \cap S_2$, $S_1 \cup S_2$, $S_1 \subseteq S_2$, $S_1 - S_2$	$O(\|S_1\| + \|S_2\|)$

Table 2
Cost of signature construction.

Type	Time	Space
G or $L^0(G)$	$O(\|V_{L^0}\| + \|E_{L^0}\| \cdot log(\|E_{L^0}\|))$	$O(\|E_{L^0}\|)$
$L^n(G)$ $(n \geq 1)$	$O(\sum_{i=1}^{n} \|V_{L^i}\| + \sum_{i=1}^{n} \|E_{L^i}\| + \|E_{L^n}\| \cdot log(\|E_{L^n}\|))$	$O(\|E_{L^n}\|)$

Similarly, if a particular graph in the database is updated, then we should compute its new signature, say sig_n. Using the exact subgraph matching algorithm discussed earlier, we search the index and identify the (sig, ptr) entry in the leaf node such that ptr matches the graph's id. We reinsert (sig_n, ptr) into the leaf node and overwrite the previous (sig, ptr). We should also update the parent entry's signature value after computing its union with sig_n. (This is similar to updating the minimum bounding rectangles of ancestor nodes after inserting a key into an R-tree.) To delete a signature, a similar procedure can be followed, where the deleted (sig, ptr) entry in the leaf can be assigned a null value, so that it can be skipped during query processing. In essence, we can rely on the efficient APIs in Berkeley DB to update the index in GiS.

6.5. Cost analysis

We implement a multiset as a list of sorted values.[4] Suppose two signatures S_1 and S_2 are sorted. The operations difference, subset, union, and intersection over these signatures (multisets) can be computed by scanning each signature once. The cost is shown in Table 1.

Let E_{L^n} and V_{L^n} denote the vertex and edge set of $L^n(G)$, respectively. The space complexity of a graph signature and time complexity to construct it is shown in Table 2. The number of edges E_{L^n} can be expressed by the following recurrence relations,

$$|V_{L^n}| = |E_{L^{n-1}}|, \text{ and} \tag{1}$$

$$|E_{L^n}| = \sum_{i=1}^{|V_{L^{n-1}}|} \binom{deg(v_i)}{2} \tag{2}$$

where v_i is a vertex of $L^{n-1}(G)$, $E_{L^0} = E(G)$, and $V_{L^0} = V(G)$. We observe that the number of edges in the computed line graph grows faster when the degree of the original graphs are higher. We expect the fastest growth in signature size upon line graph computations for complete graphs. Fortunately, most real world graphs are not complete graphs.[5]

Note that signatures are sorted and therefore, the *log* term appears in the time complexity. The time complexity for $L^n(G)$ depends on the cost of applying line graph computations n times. The space required for the signature, however, depends on the size of final line graph.

6.6. Benefits and tradeoffs of using line graphs

GiS allows a user to choose the extent of structural information captured by signatures for exact subgraph matching. By increasing n, we can improve the precision of the filtering phase for exact subgraph matching. This reduces the cost of the verification phase during query processing to discard false matches. However, by increasing the value of n, the size of signatures and the cost of signature construction increases. Longer signatures increase the size and the construction time of the index.

Suppose a graph dataset has a high number of distinct vertex labels. Then the vertex labels in the queries tend to be discriminative enough to achieve high precision for exact subgraph matching. Thus, in this case, the benefit of applying line graphs in terms of the improvement in precision may be small compared to the cost of applying line graphs.

Suppose a graph dataset has a small number of distinct vertex labels. (A real dataset with such characteristics is reported in Section 8.) Then the vertex labels in the queries may not be discriminative enough and can lead to poor precision. In this case, applying line graphs offers an attractive solution to enhance the extent of structural information captured by the data and query signatures. The benefit obtained (*i.e.*, high precision of exact subgraph matching thereby reducing the cost of verification) may far outweigh the extra cost involved in constructing and indexing longer signatures.

[4] We could have implemented a multiset by keeping (frequency, value) pairs in sorted order by value. This would facilitate the use of a binary search for some of these operations. A list representation was, however, sufficient for GiS to obtain high performance.

[5] For a complete graph G with m vertices, the size of $L^n(G)$ is upper bound by the size of a complete graph with m^{2n} vertices.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 3
Index construction.

Method	Index construction	
	Index key	# of index keys per graph
GiS	A graph signature (or multiset) on $L^n(G)$	1
Path-based	Representative paths in a graph	# of distinct representative paths
Feature-based (*e.g.*, trees, subgraphs)	Representative/frequent features in a graph	# of distinct representative/frequent patterns

Table 4
Query processing.

Method	Query processing	
	Strategy	# of index lookups
GiS	A query is not decomposed but is mapped to a single signature	1
Path-based	A query is decomposed into candidate paths	# of candidate paths
Feature-based (*e.g.*, trees, subgraphs)	A query is decomposed into candidate features	# of candidate features

We assume a typical case when queries dominate the workload. A suitable value of n can be recommended to a user using the following approach. A sample of the dataset can be used to incrementally construct a test index by increasing the value of n (starting from 0). A query workload, if available, can be used or can be generated using available tools (*e.g.*, PAFI [27]) on the sampled data. Once the queries are processed using the test index, we can compute the ratio of the improvement in the precision and the change in the total query processing time (filtering + verification), over the previous test index. When a desired ratio is reached, the value of n can be recommended to the user.

In our experiments, we show that signature construction and indexing in GiS is very fast compared to the time spent by indexing approaches that use frequent pattern mining. Therefore, we argue that the preprocessing cost involved in deciding n is still significantly smaller than performing frequent pattern mining.

6.7. GiS vs prior approaches for exact subgraph matching

Here we distinguish GiS from other approaches to highlight its novelty. In C-tree, histograms are computed for each index node and stored in the parent, by counting the number of each distinct attribute of vertices and edges [21]. Although the histogram-based pruning condition used by C-tree and subsets on multisets used by GiS seem similar, there is a difference: each element of a multiset in GiS can potentially represent a graph pattern (Fig. 7) instead of just a vertex or edge label.

By design, GiS is different from techniques that enumerate structural components (*e.g.*, paths, trees, subgraphs) up to a particular size in a graph for indexing purposes (*e.g.*, GraphGrep [16]) or use pattern mining to index frequent features (*e.g.*, trees, subgraphs). We show in Tables 3 and 4 the fundamental differences between GiS and other approaches. Note that while GiS requires a single key for a data graph, the size of the key is proportional to the number of edges in the line graph, which can be much larger than the number of edges in the data graph (Section 6.5).

Example 9. Consider the graph $\mathbb{G}_1$ shown in Fig. 7(a). Let us compare GraphGrep and GiS. Suppose the maximum length of enumerated paths in GraphGrep is 3. It is fair to compare this setting of GraphGrep with GiS for $L^2(\mathbb{G}_1)$. GraphGrep enumerates more than 40 label paths, which translates to more than 40 hash values or index keys. For GiS, $sig(L^2(\mathbb{G}_1))$ is a multiset with 18 hash values because of 18 edges in $L^2(\mathbb{G}_1)$. This multiset serves as an index key.

Another important distinction is that GiS does not enumerate all the subgraphs up to a particular size in a graph when constructing line graphs. One may wonder how indexing a signature of a line graph is different from indexing subgraphs of a particular size in a graph. The primary difference is that GiS uses the entire graph signature as an indexing unit rather than relying on subgraphs in a graph as indexing units. This enables holistic query processing in GiS where a query is processed as a whole without decomposing it into smaller units.

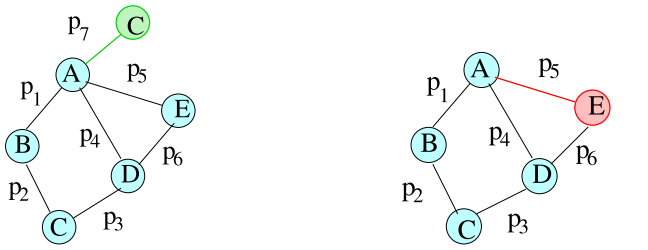
Example 10. Consider $\mathbb{G}_1$ in Fig. 7(a). Any technique that enumerates subgraphs up to size 3 for indexing would have 6 subgraphs of size 1 (*i.e.*, 1 edge), 8 subgraphs of size 2, and 12 subgraphs of size 3. This is a total of 26 subgraphs. On the contrary, $L^2(\mathbb{G}_1)$ has 18 edges. As explained earlier, each edge is a fingerprint of a subgraph in $\mathbb{G}_1$, and therefore, the $sig(L^2(\mathbb{G}_1))$ is representative of only 18 subgraphs in $L^2(\mathbb{G}_1)$.

7. Approximate (full) graph matching using signatures

Approximate (full) graph matching [45] is useful in applications where graphs similar to a query graph are desired.[6] Similarity between graphs is typically measured by *graph edit distance* that denotes the minimum cost to transform one

[6] We drop the term "full" whenever it is obvious.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002



(a) Inserting a new vertex into $\mathbb{G}_1$ (b) Deleting a vertex from $\mathbb{G}_2$

Fig. 10. An example to illustrate the influence of vertex insert and delete operations. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

graph into another by applying a sequence of edit operations (*e.g.*, insertion, deletion, relabel of vertices and edges). The cost of each edit operation depends on the application domain, and because we do not target any particular application, we assign unit cost to each edit operation. However, computing graph edit distance between two graphs is computationally expensive and the cost is exponential in the number of vertices in the graphs [37]. Therefore, to process an approximate graph matching query, it would be inefficient to compute the graph edit distance between a query and every graph in the database. A *filter* and *verification* approach seems to be a viable alternative.

7.1. Using signatures for approximate graph matching

Can the properties of graph signatures be leveraged to infer the graph edit distance between graphs? We show that this is possible and propose a scheme for approximate graph matching using graph signatures of the original graphs. (Line graphs are not used for approximate graph matching.) Given a query graph and an edit distance threshold d, we wish to find those data graphs that have an edit distance of at most d with the query graph. We begin by introducing the notion of the *influence of an edit operation* on a graph signature.

Definition 2. Suppose an edit operation e is applied on a graph G to produce a graph Q. The influence of the edit operation e on $sig(G)$, denoted by $Inf(e)$, is the maximum number of items (or hash values) that can be added, removed, or modified in $sig(G)$ to produce $sig(Q)$.

Suppose v_R denotes the relabel of a vertex v, v_I and v_D denote the insertion and deletion of a vertex v respectively, and e_I and e_D denote the insertion and deletion of an edge e, e_R denotes the relabel of an edge e, respectively. If we relabel a vertex v of degree m in a graph G, then the m items (or hash values) assigned to its edges incident at v during signature construction can change. Thus, m items can be modified in $sig(G)$ and hence $Inf(v_R) = deg(v)$. To insert (or delete) a vertex, we do the following because we are dealing with *connected graphs*: When a vertex is inserted (or deleted), it is implicitly followed by an edge insert (or delete). Thus, it causes one item to be added to (or removed from) $sig(G)$. Hence, $Inf(v_I) = Inf(v_D) = 1$. When an edge is inserted (or deleted) explicitly, it causes one item to be added to (or removed from) $sig(G)$. Hence, $Inf(e_I) = Inf(e_D) = 1$. An edge relabel will modify one item in $sig(G)$. Hence, $Inf(e_R) = 1$.

Example 11. Let us consider the graph $\mathbb{G}_1$ from Fig. 5(a). Recall that $sig(\mathbb{G}_1) = \{p_1, p_2, p_3, p_4, p_5, p_6\}$. Suppose we insert a new vertex C into $\mathbb{G}_1$ as shown (in green) in Fig. 10(a). This will add one item (*i.e.*, p_7) to $sig(\mathbb{G}_1)$. Now consider Fig. 10(b). Suppose we delete the vertex E followed by an implicit edge delete in $\mathbb{G}_1$ as shown (in red) in the figure. This will remove one item (*i.e.*, p_5) from $sig(\mathbb{G}_1)$. The remaining edge incident on E can be removed by an edge delete operation. This will remove one more item (*i.e.*, p_6) from $sig(\mathbb{G}_1)$.

Definition 3. The total influence of an edit path (*i.e.*, a sequence of edit operations to transform one graph to another) is the sum of the influence of each edit operation in the edit path.

Next, we establish a necessary condition that forms the basis for processing approximate graph matching queries in GiS. (Because of this, false matches can arise during the verification phase, which can be discarded using a graph edit distance algorithm [32].) We use the notation $ged()$ to denote the edit distance between two graphs. We state the following theorem.

Theorem 4. *Given a query graph Q and a data graph G, let d (≥ 1) denote a user specified maximum edit distance. Let $m_1, \ldots, m_n$ denote the degrees of n vertices in Q ranked in descending order. Let $|\cdot|$ denote the cardinality of a signature. If Q has an approximate graph match with G such that $ged(Q, G) \leq d$, then*

$$|sig(Q) - sig(G)| \leq maxInf_d$$

where $maxInf_d = (\sum_{i=1}^{min\{d,n\}} m_i) + max\{0, d - n\}$.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 5
Influence of edit operations.

$Inf(v_R)$	$Inf(ve_I)$	$Inf(ve_D)$	$Inf(e_I)$	$Inf(e_D)$	$Inf(e_R)$
$deg(v)$	1	1	1	1	1

Proof. Suppose $d \leq n$. There are many different edit paths of cost d that could transform G to Q. Among these, the one that maximizes the total influence is the edit path with d vertex relabels. This is because $Inf(v_R) \geq 1$. We denote this maximum total influence for edit distance d by $maxInf_d$. By computing $|sig(Q) - sig(G)|$, we are computing the count of those items that are in Q but do not have a counterpart in G. The value $maxInf_d$ provides an upper bound for this count. As we do not know which vertices are to be relabeled when the query is posed, we compute the value of $maxInf_d$ by summing the d highest vertex degrees of Q, *i.e.*, $\sum_{i=1}^{d} m_i$.

Suppose $d > n$. Then at most n relabels can be present as there are only n vertices in Q. The remaining $d-n$ edit distance should be contributed by the remaining edit operations. Each of these edit operations has an influence of 1. Therefore, an additional term $max\{0, d-n\}$ is needed. □

Note that $sig(Q) - sig(G)$ denotes the multiset difference and can be computed by scanning $sig(Q)$ and $sig(G)$ once. We refer to $maxInf_d$ as the maximum total influence for edit distance d.

7.2. Tighter bound computation

To compute a tighter bound for $|sig(Q) - sig(G)|$ in Theorem 4, we replace v_I and v_D by two new edit operations, namely, ve_I and ve_D, respectively. We define them as follows: ve_I (or ve_D) is the insertion (or deletion) of a vertex v followed by the insertion (or deletion) of the edge e incident on v. If a user specifies an upper bound on the number of each edit operation, we can compute a more precise value for $maxInf_d$. Suppose c_{v_R}, c_{ve_I}, c_{ve_D}, c_{e_I}, c_{e_D}, and c_{e_R} denote the upper bounds for each of the six edit operations, respectively. (Note that $c_{v_R} \leq d$.) Now the allowed edit distance for approximate graph matching is:

$$d = c_{v_R} + 2 \times c_{ve_I} + 2 \times c_{ve_D} + c_{e_I} + c_{e_D} + c_{e_R} \tag{3}$$

The factor of 2 is necessary for ve_D and ve_I, because each constitutes an edit operation on a vertex and an edge. The influence of each edit operation is shown in Table 5. The following value of $maxInf_d$ will be used in Theorem 4.

$$maxInf_d = \sum_{i=1}^{c_{v_R}} m_i + \sum_{i=1}^{c_{ve_I}+c_{ve_D}+c_{e_I}+c_{e_D}+c_{e_R}} 1 \tag{4}$$

Essentially, in the RHS of Equation (4), we maximize the influence of each edit operation by using Table 5.

7.3. Signature-Cardinality test

Theorem 4 provides a necessary condition. Interestingly, we can discard some false matches by focusing on the difference in the cardinality of a data graph signature and a query signature. This is because the term $|sig(Q) - sig(G)|$ in Theorem 4 is oblivious of the actual cardinality of $sig(G)$. One scenario that may arise due to this is as follows: G is output as a match when Q is a subgraph of G but G is a much larger graph than Q.

We propose the *Signature-Cardinality* test to overcome the above problem. Suppose we do not know the bounds on the edit operations, then our test is defined as follows:

$$|sig(G)| - |sig(Q)| \leq \sum_{i=1}^{d} 1 \tag{5}$$

$$|sig(Q)| - |sig(G)| \leq \sum_{i=1}^{d} 1 \tag{6}$$

The intuition is as follows. Among the different edit operations, d edge deletes can maximize the term $|sig(G)| - |sig(Q)|$. But if this term exceeds the RHS value in Equation (5), then clearly $ged(G, Q) > d$. Similarly, d edge inserts can maximize the term $|sig(Q)| - |sig(G)|$. But if this term exceeds the RHS value in Equation (6), then clearly $ged(G, Q) > d$.

Suppose we know the bounds on the edit operations, then our test is defined as follows:

$$|sig(G)| - |sig(Q)| \leq \sum_{i=1}^{c_{ve_D}} 1 + \sum_{i=1}^{c_{e_D}} 1 \tag{7}$$

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

$$|sig(Q)| - |sig(G)| \leq \sum_{i=1}^{c_{ve_I}} 1 + \sum_{i=1}^{c_{e_I}} 1 \tag{8}$$

The intuition is that the edit operations ve_D and e_D when applied to G remove one item from $sig(G)$. Therefore, if $|sig(G)| - |sig(Q)|$ exceeds the RHS value in Equation (7), then clearly we need more ve_D or e_D edit operations to transform G to Q. This would make $ged(G, Q) > d$. Similarly, ve_I and e_I edit operations can add one item to $sig(Q)$. Therefore, if $|sig(Q)| - |sig(G)|$ exceeds the RHS value in Equation (8), then clearly we need more ve_I and e_I edit operations to transform G to Q. This would again make $ged(G, Q) > d$.

7.4. Processing an approximate graph matching query

To speed up approximate graph matching, we build an index on the data signatures as in Section 6.2. Note that we compute the signatures on the original graphs (*i.e.*, $n = 0$) and do not use line graphs. This is because the influence value of an edit operation increases if $L(G)$ is used and can lead to poor pruning.

The Signature-Cardinality test discussed above is applied between the query signature and the data signature. Because we build a hierarchical index on the data signatures, this test is applied only in the leaf nodes of the index. The following remark highlights the pruning property of the index.

Remark 5. Suppose sig_p denotes the signature value in a node's entry and sig_c denotes the signature value in the child node pointed to by the node's entry. If the cardinality of the (multiset) difference between a query signature and sig_c is at most some non-negative value k, then the cardinality of the multiset difference between the query and sig_p is also at most k. (This is because $sig_c \subseteq sig_p$.) This property allows us to prune the search space of signatures while processing approximate graph matching queries.

Algorithm 5: Approximate graph matching

```
proc approxGraph(nodeId, sig_Q, maxInf_d)
  nodeId - index node id; sig_Q - query signature;
  maxInf_d - maximum influence for edit distance d
1 (nodetype, S) ← fetchNode(nodeId) /* read node */
2 foreach (sig, ptr) in S do
3   if |sig_Q − sig| ≤ maxInf_d then
4     if nodetype = LEAF then
5       if Signature-Cardinality test succeeds then
6         Output ptr as a candidate match
        end if
      else
7       approxGraph(ptr, sig_Q, maxInf_d) /* Traverse child */
      end if
    end if
  end foreach
endproc
```

Given a query Q and a maximum edit distance of d, we compute $sig_Q = sig(Q)$ and the value of $maxInf_d$. Algorithm 5 outlines the steps involved in processing Q. Starting from the root node of the index, at each node, we test if $|sig_Q - sig| \leq maxInf_d$, for each (sig, ptr) entry in it. If the test succeeds and the node is not a leaf, we traverse the child pointed by ptr and continue the search. However, if the test succeeds and the node is a leaf, we check if the Signature-Cardinality test succeeds (Equations (5) and (6)). If so, we output ptr as a candidate match. Candidate matches are tested during the verification phase to discard false matches by computing graph edit distance [32].

If the upper bounds on edit operations are available, then we compute $maxInf_d$ using Equation (4). For the Signature-Cardinality test, we use Equations (7) and (8). Note that we do not apply the Signature-Cardinality test when non-leaf nodes are searched, because the signatures in these nodes are constructed by computing the union of the child node signatures.

8. Performance evaluation

In this section, we present the performance evaluation of GiS and compare it with existing approaches for exact subgraph matching and approximate (full) graph matching. Our goal is to show that GiS is an efficient approach for processing queries on a large database of small and medium-sized data graphs that arise in domains such as chemical informatics and proteomics. GiS is designed for queries with high selectivity, and therefore, the processing cost of these queries is not dominated by the cost of verification [43].

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 6
Datasets and their characteristics.

Dataset name	Type of data graphs	Total # of graphs	# of vertices		# of edges		# of unique vertex labels (domain size)
			Max	Avg	Max	Avg	
Syn1	Small	80,000	30	12	33	13	235
Syn2	Small	80,000	36	17	42	21	180
Chem	Small	40,000	183	23	189	25	38
ASTRAL	Medium	34,810	904	174	3,677	686	106
PPI	Large	70	4,697	1,235	16,849	3,342	35,203

8.1. Implementation details

We implemented GiS in C++ and used Oracle Berkeley DB for storing the Graph Signature index. All experiments were conducted on an Intel processor machine with 2GB RAM and 250GB SATA drive, running Linux.

8.2. Datasets

We used three real datasets, namely, Chem, ASTRAL and PPI, and two synthetic datasets, namely, Syn1 and Syn2 for our evaluation. Table 6 lists these datasets and their characteristics. The synthetic datasets were generated using the PAFI software [27]. Chem is a real dataset containing graphs representing chemical compounds from NCI/NIH AIDS Antiviral Screen dataset [2]. The domain size for vertex labels in Chem was smaller than the other datasets. ASTRAL contains protein contact map graphs generated from a real protein dataset [1]. We selected a typical threshold distance of 7 Å between c_α atoms of the protein residues [23]. ASTRAL contains larger and denser graphs than the other three datasets. Contact map graphs have been used before by the authors of TALE [41]. PPI is based on protein interaction networks from the Interlogous Interaction Database [4]. PPI contains protein interactions of fly, mouse, human, yeast, rat, worm, etc. from different datasets such as MINT_Fly, BIND_Rat, and so on. Because we wanted to study the performance of GiS on large graphs, we collected the connected components and discarded small graphs.

8.3. Query sets

We employed two ways of generating high selectivity queries. For datasets containing small graphs (*e.g.*, Chem, Syn1 and Syn2), we generated graph queries using the Frequent Subgraph Mining (FSM) software in PAFI [27]. From these, we chose those that had high selectivity. For each dataset, we selected query graphs with increasing number of edges. Each query set typically had 20 randomly selected queries. We use the notation e_i to denote a set of queries with i edges each. For larger graphs, it was very time consuming to perform frequent subgraph mining. Therefore, we randomly selected a set of data graphs by fixing the minimum and maximum number of edges they should contain. All query graphs that we tested were connected graphs. When possible, we report the minimum and maximum number of true matches per query set in subsequent sections.

8.4. Techniques considered for comparison

We compared GiS with recent techniques for exact subgraph matching namely, FG-index, C-tree, and SING. Recall that FG-index uses mining for extracting frequent subgraphs for indexing and stores part of the index in main-memory and part on disk. C-tree does not rely on mining but the implementation is main-memory based. SING is also main-memory based, does not use mining, and extracts paths in graphs as features for indexing. For fairness of comparison, we measured SING's performance for finding the first occurrence of an exact subgraph match when comparing with GiS. SING has shown to be superior to recent techniques such as GCoding [57] and Tree $+\ \Delta$ [56], and hence we do not consider them in our study. SING has been shown to be superior to C-tree on large graphs. Note that techniques such as FG-index and C-tree were designed for small graphs.

For approximate (full) graph matching, we compared the performance of GiS with range queries supported by C-tree. We also compared the read cost of APPFULL with the filtering time of GiS. APPFULL scans every graph in the database during query processing. While GiS is not designed for approximate subgraph matching, we compared it with TALE, a disk-based technique for approximate subgraph matching of *large query graphs*, to gain insights into the scalability of GiS, which is also disk-based. We did not use PPI because we felt it was less appealing to pose very large queries using GiS.

Han et al. [19] developed a framework called iGraph by reimplementing previous techniques such as gIndex, FG-index, Tree $+\ \Delta$, C-tree, and GCoding to enable their comparison based on the I/O cost. We did not use the iGraph framework [19], because it was developed for Microsoft Windows. Instead, we used the original implementation of FG-index, C-tree, SING, and TALE from their respective authors.

In Table 7, we summarize the techniques considered for evaluation on small, medium-sized, and large graph datasets in our experimental study.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 7
Techniques considered for performance evaluation.

Type	Small data graphs		Medium-sized data graphs		Large data graphs	
	Datasets	Competitors	Datasets	Competitors	Datasets	Competitors
Exact subgraph matching	Syn1, Syn2, Chem	GiS, C-tree, FG-index, SING	ASTRAL	GiS, SING	PPI	GiS, SING
Approx. (full) graph matching	Syn1, Syn2, Chem	GiS, C-tree, APPFULL (read cost)	ASTRAL	GiS, TALE, APPFULL (read cost)	-na-	-na-

Table 8
Average signature size and construction time.

	Syn1	Syn2	Chem			ASTRAL	PPI
	$L(G)$	$L(G)$	G	$L(G)$	$L(L(G))$	$L(G)$	$L(G)$
# of graphs	80,000	80,000	40,000	40,000	40,000	34,810	70
Avg. size	73.3 bytes	129.9 bytes	90.3 bytes	128.7 bytes	274.5 bytes	20.6 KB	333.7 KB
Avg. time	53.8 µs	92.6 µs	40.0 µs	57.5 µs	134.2 µs	19.6 ms	341 ms

8.5. Evaluation metrics

For the indexing performance, we compared GiS with other approaches by measuring (a) the total wall clock time to build the index and (b) the total size of the index. We also measured the cost of constructing graph signatures using line graphs for GiS.

For query processing, we measured (a) the average wall clock time per query for the filtering phase and (b) the average precision achieved per query (*i.e.*, the pruning power). Suppose *cand* and *true* denote the candidate set and the set of true matches, respectively. We compute precision $p = \frac{|true|}{|cand|}$ because recall is always 1. When possible, we also measured the time for the verification phase.

Because the code provided by the authors of FG-index did not output the number of candidates or the filtering time, we compared GiS and FG-index during query processing by measuring the average wall clock time to process a query, which included both the filtering and verification costs.

We started with a cold file system buffer cache before beginning both the indexing and query processing tasks.

8.6. Constructing graph signatures and line graphs

Table 8 shows the average size of a graph signature and the average time to construct a signature for Syn1, Syn2, Chem, ASTRAL, and PPI. (The time included the cost of constructing line graphs.) One line graph computation was applied on Syn1, Syn2, ASTRAL, and PPI. On Chem, two line graph computations were employed. Although, this increased the average signature size by three times, we show later in Section 8.8 that the precision of exact subgraph matching improved significantly. As ASTRAL contained medium-sized, dense graphs, the construction time and signature size were larger than that for Syn1 and Syn2. PPI contained large graphs and therefore, the average construction time and signature size were higher than the rest.

Next, we report the cost of successive line graph construction for different graph sizes using synthetic graph datasets generated by PAFI [27]. In Fig. 11(a), we show the average time taken to construct $L^n(G)$ from $L^{n-1}(G)$, where $n = 1, 2, 3,$ and 4. Also we show the construction time on different graph datasets, with average number of vertices equals 10, 102, 814, and 7,800. Table 9 shows the average number of vertices and edges in $L^n(G)$. Line graph construction cost increased quickly when more than a few successive line graph computations were applied on large graphs.

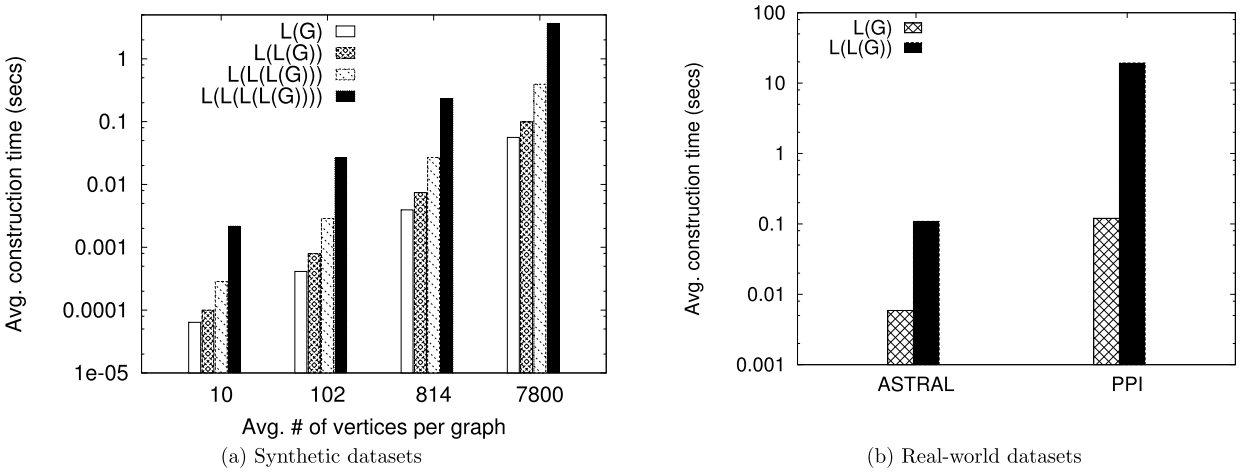
We also measured the average construction time to construct $L(G)$ and $L(L(G))$ on ASTRAL, which contained medium-sized, dense graphs. We also tested with PPI, which contained graphs even larger than those in ASTRAL. The results are shown in Fig. 11(b).

From the above results, we draw the following conclusion: Graph signatures can be constructed efficiently when a few line graph computations are required. The cost of line graph construction can grow quickly for large graphs when more than a few line graph computations are necessary. (This is consistent with our analysis in Section 6.5.) However, for small and medium-size real-world datasets, the signature generation method of GiS is very efficient. Note that GiS cannot handle very large graphs with millions of vertices and edges. Techniques like NeMa [26] are well-suited for such graphs.

8.7. Indexing performance

We first present the performance results for indexing on the datasets containing small data graphs. Then, we present the results on the datasets containing medium-sized and large data graphs.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002



(a) Synthetic datasets (b) Real-world datasets

Fig. 11. Cost of line graph construction.

Table 9
Size of line graphs for the synthetic datasets.

Size	$L(G)$		$L^2(G)$		$L^3(G)$		$L^4(G)$	
	Avg. $\|E\|$	Avg. $\|V\|$	Avg. $\|E\|$	Avg. $\|V\|$	Avg. $\|E\|$	Avg. $\|V\|$	Avg. $\|E\|$	Avg. $\|V\|$
10	19.1	12.3	46.8	19.1	220.4	46.8	2234.1	221.3
102	174.0	102.1	469.0	174.0	2353.0	469.0	24150.2	2353.0
814	1613.8	1002.3	4086.0	1613.8	19321.6	4086.0	185344	19321.6
7800	17271	10012	47754.5	17271	248847	47754.5	2.67e+06	248847

8.7.1. Evaluation on small data graphs

We compared GiS with its competitors on `Syn1`, `Syn2`, and `Chem`. For FG-index, as selected by the authors, we set the value of tolerance factor $\sigma = 0.01$ and $\sigma = 0.1$ for synthetic datasets and `Chem`, respectively. For fair comparison between GiS and C-tree, we built indexes with a fanout of 500. For SING, we used the default value (provided in the source code) for feature length (or path length), which yielded high precision of exact subgraph matching.

Fig. 12(a) shows the total time taken to index the entire datasets. For GiS, the total time included the time for signature construction and bulk-loading the index. Because `Chem` had a small number of distinct vertex labels, GiS used two line graph computations for this dataset. Clearly, GiS outperformed its competitors significantly in most cases. FG-index incurred higher indexing cost than GiS due to the application of frequent subgraph mining. For example, GiS was 3.3 times faster than FG-index for `Syn2`. On `Chem`, FG-index was two orders of magnitude slower than GiS. C-tree incurred a higher cost than GiS on the synthetic datasets due to the hierarchical clustering operation. For example, GiS was 3 times faster than C-tree for `Syn2`. On `Chem`, C-tree was slightly slower than GiS. It also required more memory than GiS as it is a main-memory based approach. As expected, on `Chem`, the time taken by GiS increased with increasing number of line graph computations. Finally, GiS was faster than SING for all the three datasets. Overall, GiS's holistic processing approach resulted in efficient index construction.

Fig. 12(b) shows the total size of the index constructed by GiS and its competitors. On `Chem`, GiS had the smallest index size even with two applications of line graphs. As expected, the index size of GiS increased with the computation of line graphs. On the synthetic datasets, the index sizes were the largest for SING due to its path-based indexing approach. Once again, GiS's holistic processing approach yielded small-sized indexes. Although FG-index had smaller index sizes than GiS on the synthetic datasets, we later show that GiS outperformed FG-index on exact subgraph matching queries.

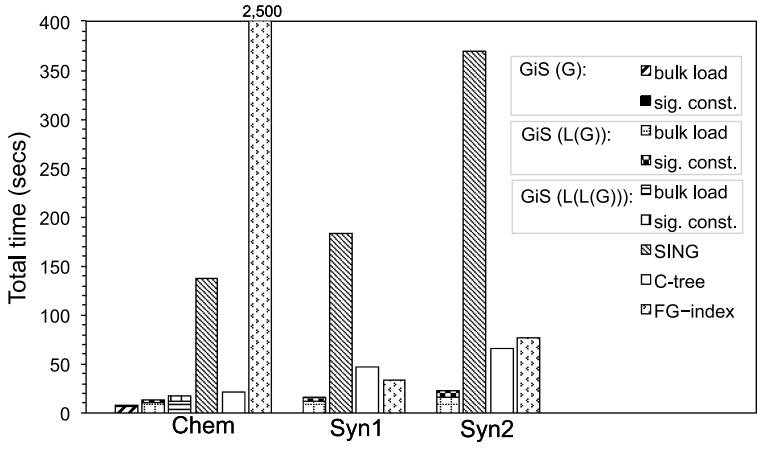
Next, we present results to show that GiS shows good scalability when the number of graphs to index are increased in the database. Fig. 13(a) shows the index construction time. Fig. 13(b) shows the index size. Both the index construction time and index size increased almost linearly with increasing number of graphs.

8.7.2. Evaluation on medium-sized and large data graphs

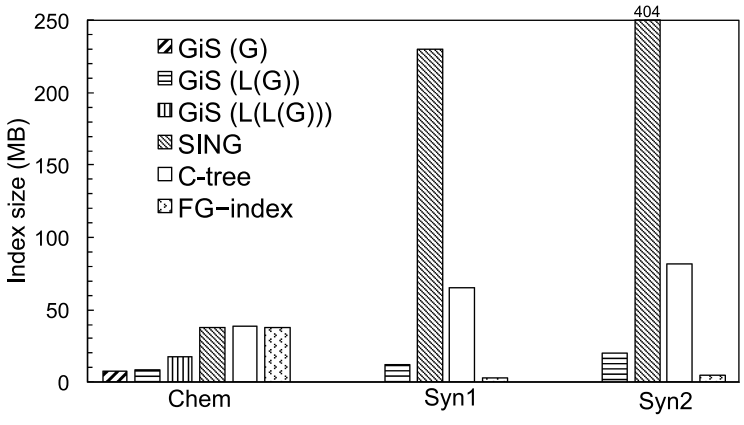
`ASTRAL` contained medium-sized, dense graphs. GiS indexed this dataset using one line graph computation in 1,192 seconds. (This time included the time for signature construction and bulk-loading the index.) The index size was 847 MB. We set the buffer cache size in Berkeley DB to 164 MB.

Both FG-index and C-tree failed to index `ASTRAL`. FG-index crashed as it ran out of virtual memory due to very large number of frequent patterns (more than a million) for a modest setting of $\sigma = 0.1$ and $\sigma = 0.15$. (These values are similar to values chosen by the authors of FG-index in their paper.) Moreover, finding frequent graph patterns required more than 3

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

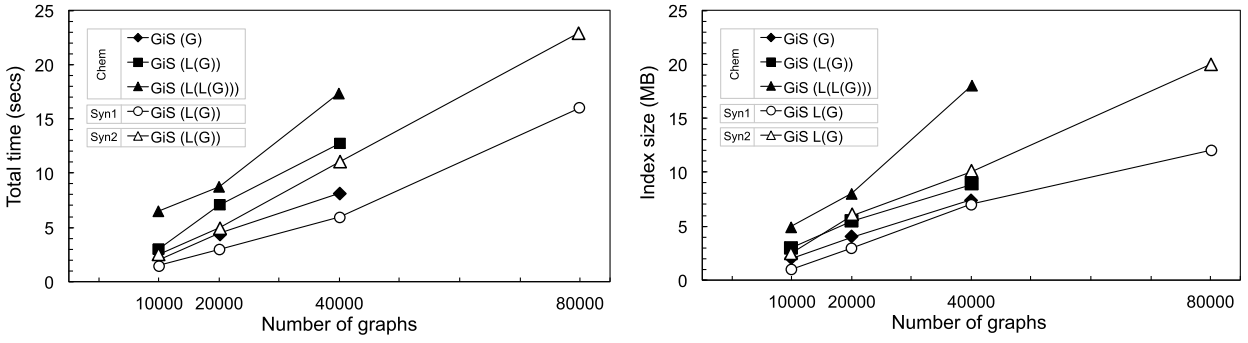


(a) Index construction time



(b) Index size

Fig. 12. Indexing performance (small graphs).



(a) Index construction time

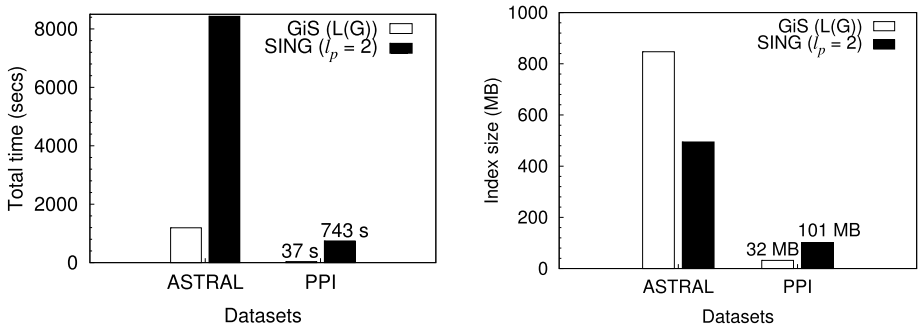
(b) Index size

Fig. 13. Scalability of index construction in GiS (small graphs).

days. This clearly demonstrates the limitation of a mining-based approach on moderate-sized, dense graphs. C-tree crashed while indexing ASTRAL even when maximum amount of memory was provided for heap space.

SING, a main-memory based technique, was unable to index the entire ASTRAL dataset using the default value of path length (or feature length) on a machine with 8 GB RAM. By design, SING stored not only the paths up to a particular length (along with frequency information) but also the position information of features for improved indexing and verification. This shows that main-memory based techniques will fail when graphs become denser and underpins the need for disk-based indexing techniques. We reduced the path length setting in SING to 2 to successfully index ASTRAL so that a comparison can be made with GiS. The index construction time and index size for ASTRAL are shown in Figs. 14(a) and 14(b), respectively.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002



(a) Index construction time (b) Index size

Fig. 14. Indexing performance (medium-sized and large graphs).

GiS was much faster than SING, although the index size of GiS was larger. Despite the larger index size, we show later in this section that GiS can outperform SING for exact subgraph matching. (The scalability of GiS on medium-sized graphs is presented later in Section 8.9.)

PPI contained the largest graphs compared to other datasets. We compared the indexing performance of GiS and SING. (C-tree and FG-index are designed for small graphs.) As the implementation of SING is main-memory based, it failed to index the entire dataset on a machine with 8 GB RAM using the default value of path length. So we reduced the path length to 2 for a fair comparison with GiS, which indexed this dataset using $L(G)$. (We set the Berkeley DB buffer cache size in GiS to 164 MB.) The index construction time and index size for PPI are shown in Figs. 14(a) and 14(b), respectively. GiS was faster and built a smaller index than SING.

8.8. Exact subgraph matching queries

We present the performance results of GiS and its competitors on small, medium-sized, and large data graphs. GiS can outperform its competitors on small and medium-sized data graphs, which is the main focus of our work. However, on large data graphs, GiS is unable to perform better than its competitor on certain types of queries.

8.8.1. Evaluation on small data graphs

Fig. 15 shows the evaluation results for Chem, a real-world dataset, which had only 38 unique/distinct vertex labels. (Among the tested datasets, Chem had the smallest number of distinct vertex labels.) Because some of the approaches did not support graphs with edge labels, for fair comparison, we first ignored edge labels in Chem. As shown in Fig. 15(a), GiS outperformed C-tree and SING for exact subgraph matching using $L(L(G))$. For example, the average response time of GiS was an order of magnitude faster than the average filtering time of C-tree for e_3. It was also two times faster than SING for the same query set. For e_5, GiS was an order of magnitude faster than SING. Because of the small number of distinct vertex labels in Chem, a path feature in SING was more likely to be found in many graphs. Therefore, its filtering cost was higher than that of GiS. GiS, using two successive line graph computations (*i.e.*, $L(L(G))$), performed holistic matching and outperformed its competitors on all the query sets.

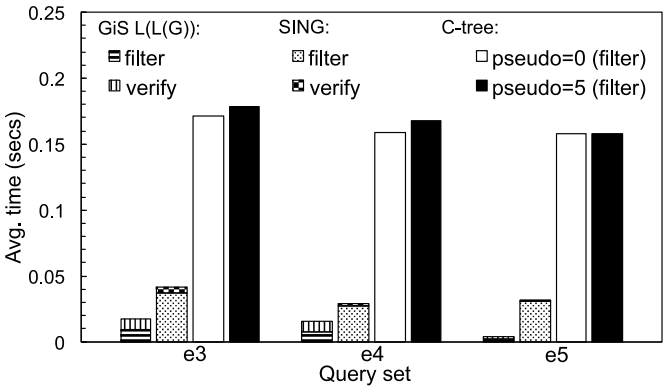
The average precision achieved by each approach is shown in Fig. 15(b). Line graphs significantly improved the precision achieved by GiS. For example, the average precision improved from 0.582 to 0.997 for e_3 when $L(L(G))$ was used instead of G. Similar improvement was observed for e_4. Overall, GiS (with $L(L(G))$) achieved comparable precision with C-tree (pseudo = 5) and SING. These results validate our claim that *multiple line graph computations can provide significant benefit* on datasets where the vertex labels are not discriminative enough. Interestingly, GiS (without line graphs *i.e.*, G) had similar pruning power to C-tree without pseudo-isomorphism tests (*i.e.*, pseudo = 0).

We compared GiS and FG-index on Chem by allowing edge labels. (Only the response time is output by the FG-index code.) Fig. 16 shows the average time taken by FG-index and GiS. Clearly, GiS's holistic processing approach outperformed FG-index's non-holistic approach. Recall that FG-index uses frequent subgraphs as the features for indexing. During query processing, it finds partial matches for the features in a query followed by computing the intersection of the partial matches. As a result, it was slower than GiS.

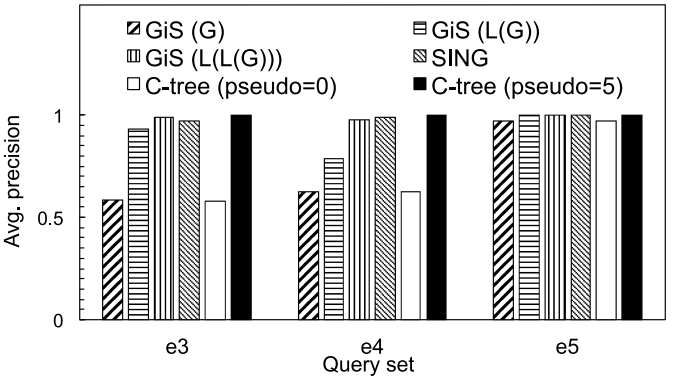
Next, we report the evaluation results of GiS, C-tree, FG-index, and SING on the synthetic datasets. For C-tree, the pseudo-isomorphism level of 1 was sufficient to achieve high precision. For GiS, we applied one line graph computation on the synthetic datasets. Figs. 17(a) and 17(b) show the average response time of the different approaches. GiS outperformed C-tree and FG-index for all the query sets. The larger index sizes of C-tree resulted in slower response time compared to GiS. Although, the index size of FG-index (Fig. 12) was smaller than that of GiS, its non-holistic approach of query processing increased the response time. For example, GiS was 5.5 faster than FG-index for Syn2 (e_{12}).

Let us now discuss the performance comparison between SING and GiS. We observed an interesting trend: GiS was slower than SING for smaller sized queries (e_3 and e_6), but was faster than SING as the size of the queries increased

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002



(a) Average time taken



(b) Average precision

Fig. 15. Performance results on `Chem` (no edge labels).

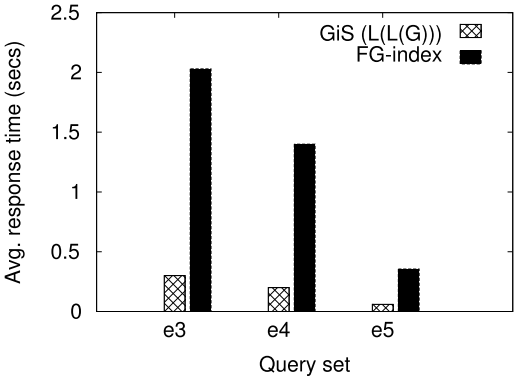


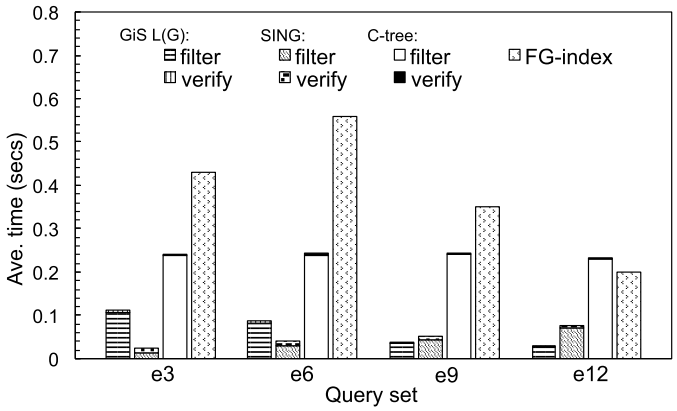
Fig. 16. Performance results on `Chem` (with edge labels).

(e_9 and e_{12}). In SING, the cost of path-based matching during filtering increased with the increase in the query size. This trend is consistent with our motivation (Section 3) for designing a holistic approach to query processing. While GiS outperformed SING on `Chem` even for small-sized queries, on the synthetic datasets, the higher number of distinct vertex labels was more advantageous to SING for small-sized queries (e_3 and e_6).

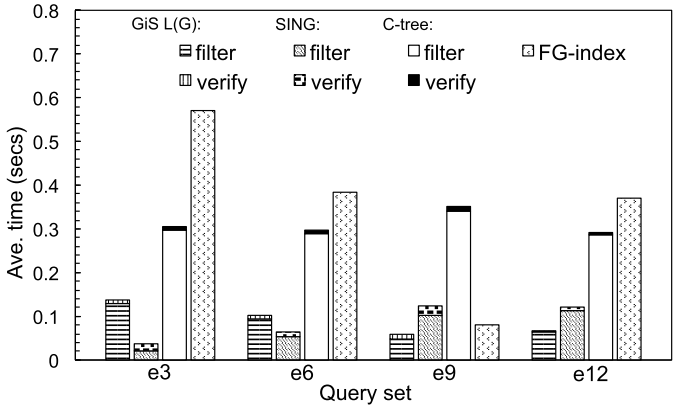
All the approaches yielded high precision of exact subgraph matching. The results are shown in Fig. 17(c). The minimum and maximum number of true matches per query set for each dataset are shown in Table 10.

8.8.2. Evaluation on medium-sized data graphs

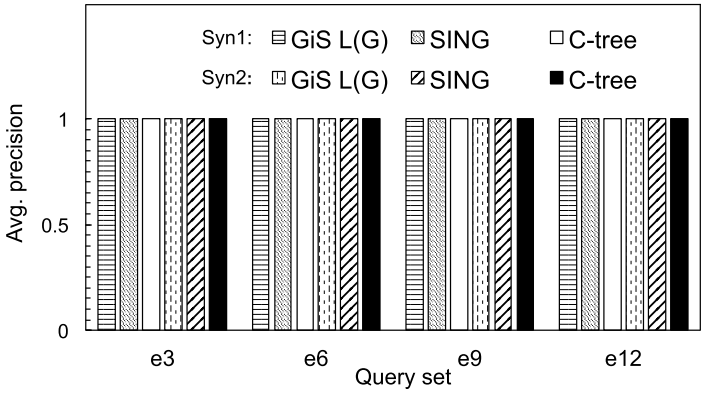
As discussed in Section 8.7.2, C-tree and FG-index failed to index `ASTRAL`. To compare GiS and SING, we conducted two sets of experiments: one by indexing the entire `ASTRAL` dataset (34,810 graphs) and the other by indexing a subset of `ASTRAL` (containing 2,000 graphs), which we will refer to as $\texttt{ASTRAL}_s$. SING indexed `ASTRAL` using a path length value



(a) Average response time (Syn1)



(b) Average response time (Syn2)



(c) Average precision

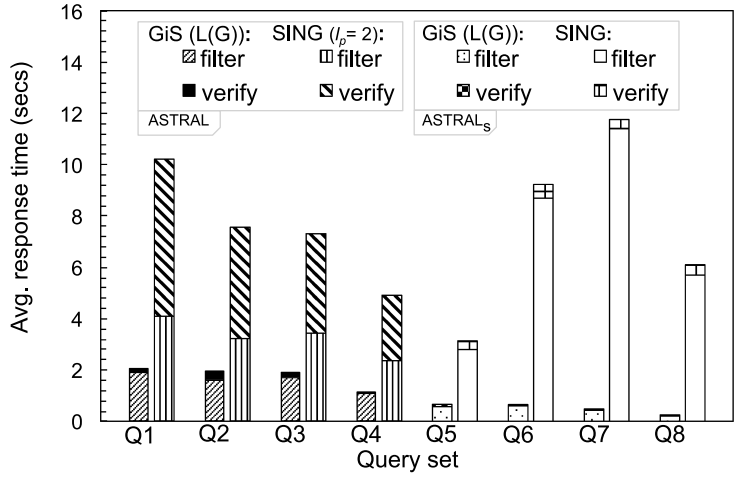
Fig. 17. Performance results on Syn1 and Syn2.

Table 10
True matches for the tested query sets.

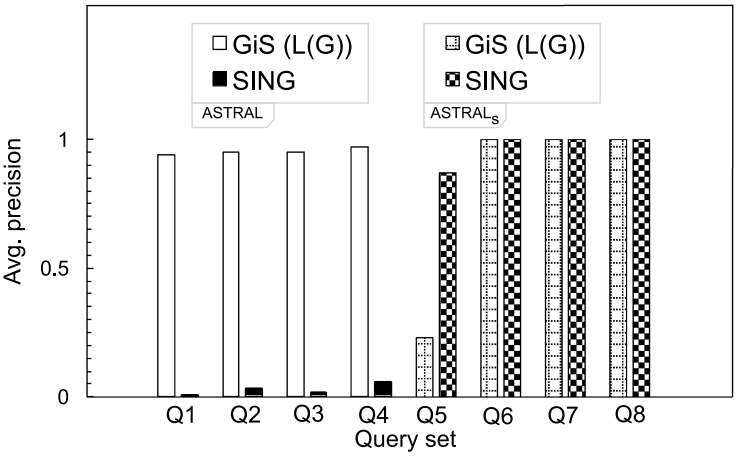
# of true matches	Syn1				Syn2				Chem		
	e_3	e_6	e_9	e_{12}	e_3	e_6	e_9	e_{12}	e_3	e_4	e_5
Minimum	181	236	82	82	141	82	634	80	53	28	4
Maximum	727	727	597	205	757	757	634	634	80	40	8

Table 11
Query sets used for evaluation on ASTRAL.

Dataset	Query set	Avg. # of vertices	Avg. # of edges	# of true matches (min, max)
ASTRAL	Q_1	23	48	(1, 11)
ASTRAL	Q_2	26	77	(1, 118)
ASTRAL	Q_3	34	80	(1, 71)
ASTRAL	Q_4	36	110	(1, 24)
ASTRAL_s	Q_5	9	14	(1, 22)
ASTRAL_s	Q_6	23	50	(1, 2)
ASTRAL_s	Q_7	27	79	(1, 11)
ASTRAL_s	Q_8	28	85	(1, 11)



(a) Avg. time taken



(b) Avg. precision

Fig. 18. Query processing performance on ASTRAL.

of 2. Because SING's implementation is main-memory based, it failed to index ASTRAL using higher values of path length. SING indexed ASTRAL_s using the default path length of 3.

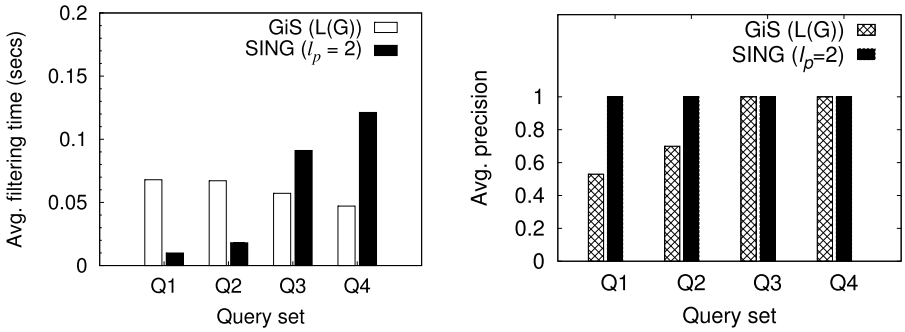
Table 11 shows the query sets used to compare GiS and SING, the average number of vertices and edges in each query set, along with the minimum and maximum number of true matches per query set. (The queries were randomly selected from the dataset.) Fig. 18(a) shows the average response time per query in each query set. (The time included the filtering time and verification time.) Fig. 18(b) shows the average precision per query in each query set.

Let us consider the results for ASTRAL. Clearly, GiS was several times faster than SING. SING yielded poor precision of matching using path length of 2, but GiS (using $L(G)$) achieved high precision of matching. The average verification time for

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 12
Query sets used for evaluation on PPI.

Query set $L(G)$	Avg. # of vertices	Avg. # of edges	# of queries
Q_1	25	27	70
Q_2	43	49	5
Q_3	499	871	8
Q_4	560	1232	12



(a) Avg. filtering time (b) Avg. precision

Fig. 19. Query processing performance on PPI.

SING was much higher because of poor precision of matching. One would expect that a higher path length during indexing would yield better precision for SING. Unfortunately, SING failed to index the dataset for such a setting.

Next, let us consider the results for $\texttt{ASTRAL}_S$. SING used the default value of path length during indexing. While SING indexed these graphs in 1,680 s with an index size of 2.9 GB, GiS indexed them in 35 s with an index size of 47 MB. As shown in Fig. 18(a), GiS was several times faster than SING for all the query sets. On Q_7, GiS was more than an order of magnitude faster than SING. While SING achieved high precision using the default path length, the size of the index was much larger than that of GiS and resulted in slower query processing. This demonstrates that the holistic processing approach of GiS is indeed superior than SING on medium-sized graphs. Although GiS had lower precision than SING on Q_5 (0.23 vs 0.87), yet the average response time of GiS was much faster than SING because the verification cost was small for the query sets, which contained high selectivity queries.

8.8.3. Evaluation on large data graphs

We report the query processing performance of GiS and SING on PPI. We used four query sets, namely, Q_1, Q_2, Q_3, and Q_4 as shown in Table 12. The queries were selected randomly from the original dataset based on their sizes. The average number of vertices and edges per query set, and the number of queries in each set are also shown. GiS used one line graph computation on the data graphs (*i.e.*, $L(G)$). Fig. 19(a) shows the average filtering time of GiS and SING. It is interesting to note that on smaller-sized queries, SING outperformed GiS. However, on larger-sized queries with hundreds of vertices, GiS outperformed SING. We observed that the filtering cost of SING increased when the size of the queries increased. This is because more features (or paths) in a query were examined during query processing—a limitation of a non-holistic approach.[7]

It is interesting to note that verification cost became the bottleneck for GiS. Similar to techniques like C-tree and FG-index, GiS relies on available exact subgraph matching algorithms [43] for the verification step on candidate graphs. SING, however, uses the information stored with features from the filtering step to speed up verification. We observed that using VFLib, a popular exact subgraph matching library,[8] the verification of an exact subgraph match did not finish even after an hour on certain candidate data graphs. On the other hand, SING was able to complete the verification phase; the average time taken for verification per query was at most 60 ms. Despite this limitation during verification, GiS did achieve high precision like SING for larger-sized queries. The average precision obtained by GiS and SING are shown in Fig. 19(b). (We used the true matches output by SING to compute the precision of GiS.)

One may wonder if simply indexing the edges of large graphs with many distinct vertex labels can lead to high precision for exact subgraph matching. So we conducted an experiment on PPI using SING by setting the path length to 1. We used the query sets Q_1, Q_2, Q_3, and Q_4. Overall, SING yield very poor precision, and the average precision for query sets Q_1, Q_2, Q_3, and Q_4 was 0.001, 0.0004, 0.0546, and 0.0562, respectively. This shows that by solely indexing edges on large graphs with many distinct vertex labels is not sufficient for exact subgraph matching.

[7] While the filtering cost of SING increased when path length of 3 was used, its performance trend in comparison with GiS was similar.
[8] http://amalfi.dis.unina.it/graph/.

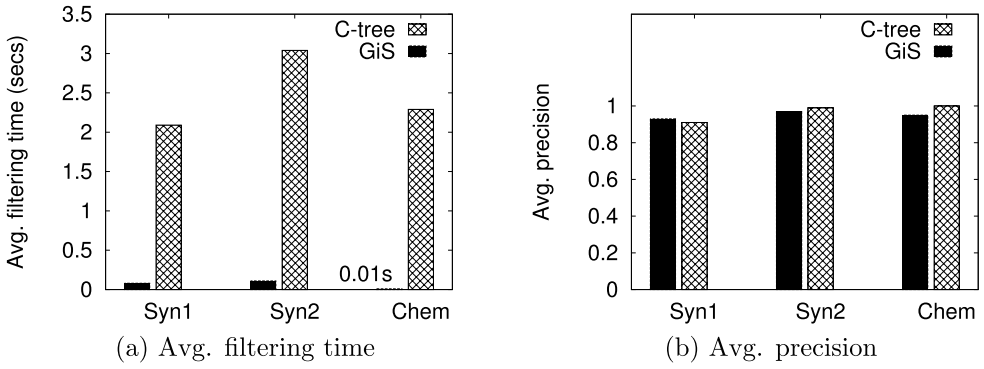


Fig. 20. Approximate graph matching (small data graphs).

8.8.4. Summary of evaluation

We summarize our key observations for exact subgraph matching based on the performance evaluation results of GiS, C-tree, SING, and FG-index on small, medium-sized, and large data graphs. Let us first consider the results on datasets with small data graphs. GiS was faster than C-tree and FG-index in all cases. It's holistic approach of query processing yielded superior performance than FG-index's frequent subgraph based indexing approach. GiS was faster than SING on a real-world dataset with small number of distinct vertex labels (*i.e.*, 38). However, on synthetic datasets with larger number of distinct vertex labels (*i.e.*, 180 and 235), SING was faster than GiS on small-sized queries. But as the size of the queries increased, SING was slower than GiS due to its path-based matching approach as more paths in the queries had to be matched.

On medium-sized (dense) data graphs, GiS was clearly superior to SING on all the tested query sets. The size of the indexes built by SING were significantly larger than those built by GiS. This was because SING used paths as features for indexing. This also increased the time taken by SING to process exact subgraph matching queries. On the contrary, reducing the path length in SING yielded poor precision of matching, which in turn increased the cost of the verification phase. On large data graphs with many distinct vertex labels, SING outperformed GiS in most cases. Only when a query had hundreds of vertices, GiS could do better than SING. However, such a query may not be useful in practice.

GiS required one or two line graph computations to achieve high precision on the tested datasets. Overall, line graph computation was efficient on small and medium-sized data graphs. However, on large graphs, the performance of line graph computation could become less efficient. This would be especially true when the large data graphs have small number of distinct vertex/edge labels as more than a few line graph computations would be required to make the signatures discriminative.

8.9. Approximate (full) graph matching queries

GiS is designed to support approximate (full) graph matching. It outputs those data graphs that have an edit distance of at most d (a user specified edit distance) with the query graph. Further, each edit operation is assigned a unit cost. Note that GiS does not use line graphs for approximate graph matching. We compared our scheme with similarity queries (*i.e.*, range queries) supported by C-tree. We also compared the read cost of APPFULL with the filtering time of GiS. APPFULL scans every graph in the input database and does not use an index. Finally, we compared GiS with TALE on medium-size, dense graphs. We did not compare GiS with SAGA due to significant difference in the distance model between the two approaches.

8.9.1. Evaluation on small data graphs

First, we compared GiS and C-tree. The average number of edges per query tested on each dataset was 14, 24, and 39, for `Syn1`, `Syn2`, and `Chem`, respectively. The tested queries had maximum edit distance $d = 1$ and $d = 3$. While running C-tree, we assigned the appropriate range value according to C-tree's similarity model. Figs. 20(a) and 20(b) show the average filtering time and the average precision for processing approximate graph matching queries. GiS achieved high precision like C-tree. However, the approximate graph matching method of GiS was significantly faster than C-tree's range query processing. The average filtering time of GiS was between 14 to 234 times faster than that of C-tree.

Next, we decided to gain insights on the performance of APPFULL. APPFULL does not build an index and therefore, during query processing, it sequentially reads the stars of every graph in the database to compute lower and upper bounds on the edit distances. Because the code of APPFULL was not readily available from the authors, we conducted a simple experiment to measure the cost of reading the stars structures of the input graphs in APPFULL. (We did not measure the cost of computing the bounds in APPFULL.) We compared this read cost with the average filtering time of GiS. The results are shown in Table 13. *For all the datasets, we observed that in about the time taken by APPFULL to just read the input graphs, GiS was able to complete its filtering phase.* We conclude that an index is necessary for efficient approximate graph matching.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

Table 13
Filtering time of GiS vs APPFULL (read cost).

Dataset	GiS (Avg. filtering time)	APPFULL (read cost)
Syn1	0.08 s	0.11 s
Syn2	0.11 s	0.16 s
Chem	0.01 s	0.1 s

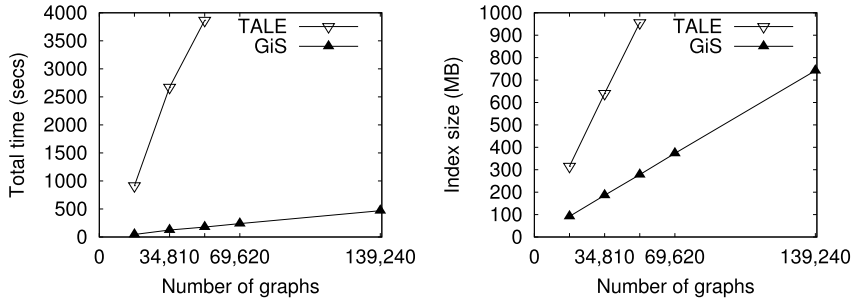
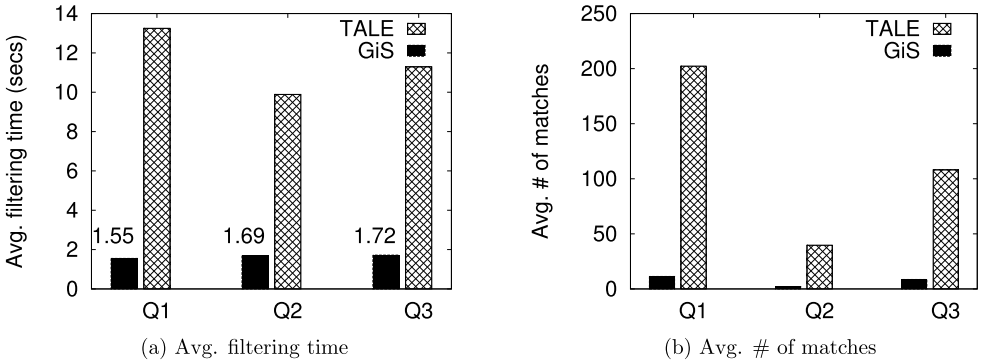


Fig. 21. Indexing performance (ASTRAL).



(a) Avg. filtering time (b) Avg. # of matches

Fig. 22. Performance evaluation on ASTRAL.

Furthermore, sequentially scanning the input graphs, like in APPFULL, will lead to poor performance on a large database of graphs.

8.9.2. Evaluation on medium-sized data graphs

As discussed in Section 8.8.2, C-tree failed to index ASTRAL. We compared GiS with TALE, which is a disk-based technique for approximate subgraph matching of *large query graphs*, on ASTRAL. (TALE is reported to be faster than another recently proposed technique called SAPPER [55].) We did this to gain insights into the scalability of GiS w.r.t. a disk-based approach. TALE is, however, designed for approximate subgraph matching, which is a more general problem than approximate (full) graph matching supported by GiS. TALE indexes the neighborhood information of each node in a graph. Fig. 21 shows the indexing performance of TALE and GiS by increasing number of graphs in the database, starting from 17,405 graphs. For fairness, we only report the cost of constructing the NH-index in TALE. (We did not run TALE beyond 55,215 graphs due to swap space constraints.) We set the buffer pool size to 512 MB for TALE [41] and 164 MB for GiS. We observed that GiS was several times faster than TALE, built smaller indices, and showed good scalability.

Next, we report the performance of GiS and TALE on approximate graph matching queries. We used three query sets with 100 random queries each. The query sets Q_1, Q_2, and Q_3 had an average of 58, 82, and 106 vertices, respectively. Also, Q_1, Q_2, and Q_3 had an average of 205, 304, and 404 edges, respectively. We set the parameters for TALE as suggested by the authors ($\rho = 25\%$, $P_{imp} = 25\%$). For fairness, we report only the filtering time of TALE (*i.e.*, time taken for index probing) and compare it with the filtering time of GiS. The average filtering time per query is reported in Fig. 22(a). While GiS uses a holistic matching approach, TALE starts by processing individual nodes in a query.

The final result set output by TALE (after applying its match algorithm) was a superset of the results output by GiS. (The recall of GiS is always one.) Because TALE is designed for a more general problem, *i.e.*, approximate subgraph matching, it will require an additional post-processing step to output the right matches for approximate (full) graph queries as TALE first finds approximate subgraph matches. Fig. 22(b) shows that average number of matches per query output by GiS and TALE.

We also compared the read cost of APPFULL with GiS for ASTRAL. APPFULL required 1.51 s to just read the input data graphs. On the other hand, GiS was able complete its filtering phase in almost the same time (Fig. 22(a)).

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

8.10. Extensions to GiS

We propose several extensions to GiS to support new class of applications. The first extension is to support directed graphs that are small and medium-sized arising in applications such as biological pathway analysis, supply chain management (SCM), and business process management (BPM). Biological pathways (*e.g.*, signal transduction pathways) are being used for drug discovery and disease treatment [10]. These pathways can be analyzed using graph-based searching to identify species and biological processes with similar interactions between molecules in a cell. In SCM, directed graphs are used to denote the route of orders/packages in a facility tracked by RFIDs et al. [6]. Graph query processing can be used for analyzing the efficiency of the supply chains and detecting problems and bottlenecks et al. [6]. Similarly, in BPM, directed graphs are used to model business processes. Given the large number of such processes within a company, there is need for efficient management of business process models. This requires efficient indexing and querying of these graphs [28,25].

It is straightforward to extend GiS to support a large database of directed graphs that arise in the aforementioned applications. This can be achieved by constructing the ordered-label pair of each edge during signature construction by employing the direction of the edge instead of using $lex(\cdot)$. That is, if there is an edge from v_i to v_j, then the ordered-label pair of the edge is $l(v_i), l(v_j)$. Similarly, during line graph construction, the label of a vertex in a line graph can be computed as above. The rest of the steps in GiS do not need any modifications.

GiS can be extended to support update operations on the index as discussed in Section 6.4. From our experiments, we observed that inserting signature into Berkeley DB required a few milliseconds. To compute the union on a group of graph signatures in a leaf node and then insert the union into Berkeley DB required a few hundred milliseconds. We believe updates can be supported efficiently using Berkeley DB as the underlying storage.

GiS did not perform well on large graphs with thousands of vertices and edges for exact subgraph matching. This was because of the long signatures that are generated for the graphs and processed during filtering. In addition, the cost of the verification using existing tools like VFLib for subgraph isomorphism became a bottleneck. One way to solve this issue is to partition the data graphs and construct signatures on the partitions. Partitioned edges would need to be kept track of to avoid missing matches. The verification phase will also become faster as we would be dealing with smaller graphs.

Given that modern processors contain several processing cores, we can speed up query processing in GiS by exploiting pipeline parallelism during the filtering and verification phases. Note that approaches such as NeMa can naturally leverage parallelism.

9. Conclusions

We propose a new way of representing a graph holistically via its signature. Thus, graphs can be indexed and queried holistically. Using a bulk-loading strategy, a disk-based index is built over graph signatures. The index speeds up the processing of exact subgraph matching and approximate graph matching queries. We propose the novel use of line graphs for improving the precision of exact subgraph matching. We also develop techniques for approximate graph matching by leveraging the properties of graph signatures without the use of line graphs. We demonstrate through performance evaluation on real and synthetic datasets that GiS's holistic query processing approach provides a scalable and efficient disk-based solution for indexing and querying a large database of small and medium-sized data graphs.

Note that GiS does not support mismatches on the vertex or edge labels. Furthermore, the holistic query processing approach of GiS is not suited for very large graphs with millions of vertices and edges. For such cases, an approach like NeMa, which decomposes a large graph into sub-units for indexing and query processing is a better alternative.

Acknowledgments

We thank the anonymous reviewers for their excellent comments and suggestions. We are grateful to the authors of C-tree, FG-index, SING, and TALE for their code and assistance. This work was supported in part by the National Science Foundation under Grant No. 1115871.

References

[1] ASTRAL, http://astral.berkeley.edu/pdbstyle-1.71.html.
[2] NCI/NIH AIDS Antiviral Screen Dataset, http://dtp.nci.nih.gov.
[3] Oracle Berkeley DB, available from http://www.oracle.com/technology.
[4] The Interlogous Interaction Database, http://ophid.utoronto.ca/ophidv2.201.
[5] M.A. Bayir, T.D. Guney, T. Can, Integration of topological measures for eliminating non-specific interactions in protein interaction networks, Discrete Appl. Math. 157 (2009) 2416–2424.
[6] D. Bleco, Y. Kotidis, Graph analytics on massive collections of small graphs, in: Proceedings of the 17th International Conference on Extending Database Technology, Athens, Greece, 2014, pp. 523–534.
[7] A. Broder, On the resemblance and containment of documents, in: Proc. of the Compression and Complexity of Sequences 1997, 1997.
[8] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proc. of the 2002 SIGMOD Conference, Wisconsin Madison, WI, 2002.
[9] C. Calude, G. Paun, G. Rozenberg, A. Salomaa, Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View, Springer, 2001.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002

[10] E.G. Cerami, B.E. Gross, E. Demir, I. Rodchenkov, O. Babur, N. Anwar, N. Schultz, G.D. Bader, C. Sander, Pathway commons, a web resource for biological pathway data, Nucleic Acids Res. 39 (Database Issue) (Jan 2011) D685–D690.
[11] J. Cheng, Y. Ke, A. Fu, J. Yu, Fast graph query processing with a low-cost index, VLDB J. (2010) 1–19.
[12] J. Cheng, Y. Ke, W. Ng, A. Lu, FG-index: towards verification-free query processing on graph databases, in: Proc. of the 2007 SIGMOD Conference, Beijing, China, 2007, pp. 857–872.
[13] J. Cheng, Y. Ye, W. Ng, Efficient query processing on graph databases, ACM Trans. Database Syst. 34 (1) (2009) 1–48.
[14] N.V. Dokholyan, L. Li, F. Ding, E.I. Shakhnovich, Topological determinants of protein folding, Proc. Natl. Acad. Sci. 99 (13) (2002) 8637–8641.
[15] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1979.
[16] R. Giugno, D. Shasha, GraphGrep: a fast and universal method for querying graphs, in: Intl. Conf. on Pattern Recognition, 2002.
[17] A. Golovin, K. Henrick, Chemical substructure search in SQL, J. Chem. Inf. Model. 49 (1) (2009) 22–27.
[18] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proc. of the 1984 SIGMOD Conference, June 1984, pp. 47–57.
[19] W.-S. Han, J. Lee, M.-D. Pham, J.X. Yu, iGraph: a framework for comparisons of disk-based graph indexing techniques, Proc. VLDB Endow. 3 (September 2010) 449–459.
[20] R. Haralick, G. Elliott, Increasing tree search efficiency for constraint satisfaction problems, Artif. Intell. 14 (3) (Oct 1980) 263–313.
[21] H. He, A.K. Singh, Closure-tree: an index structure for graph queries, in: Proc. of the 22th ICDE Conference, 2006, pp. 38–49.
[22] J. Hellerstein, A. Pfeffer, The RD-tree: an index structure for sets, Technical Report UW-CS-TR-1252, Univ. of Wisconsin Madison, 1994.
[23] J. Hu, X. Shen, Y. Shao, C. Bystroff, M.J. Zaki, Mining protein contact maps, in: BIOKDD02: Workshop on Data Mining in Bioinformatics, 2002, pp. 3–10.
[24] H. Jiang, H. Wang, P.S. Yu, S. Zhou, GString: a novel approach for efficient search in graph databases, in: Proc. of the 23th ICDE Conference, 2007, pp. 566–575.
[25] T. Jin, J. Wang, N. Wu, M. Rosa, A.H.M. Hofstede, Efficient and accurate retrieval of business process models through indexing, in: Proc. of On the Move to Meaningful Internet Systems: OTM 2010, Berlin, Heidelberg, 2010, pp. 402–409.
[26] A. Khan, Y. Wu, C.C. Aggarwal, X. Yan, NeMa: fast graph search with label similarity, in: Proceedings of the 39th International Conference on Very Large Data Bases, PVLDB'13, Trento, Italy, 2013, pp. 181–192.
[27] M. Kuramochi, G. Karypis, An efficient algorithm for discovering frequent subgraphs, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1038–1051.
[28] T. Madhusudan, J. Zhao, B. Marshall, A case-based reasoning framework for workflow model management, in: Advances in Business Process Management, Data Knowl. Eng. 50 (1) (2004) 87–115.
[29] B.T. Messmer, H. Bunke, Efficient subgraph isomorphism detection: a decomposition approach, IEEE Trans. Knowl. Data Eng. 12 (2) (2000) 307–323.
[30] J. Nacher, T. Yamada, S. Goto, M. Kanehisa, T. Akutsu, Two complementary representations of a scale-free network, Physica A 349 (2005) 349–363.
[31] R.D. Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, D. Shasha, SING: subgraph search in non-homogeneous graphs, BMC Bioinform. 11 (1) (Feb 2010) 96.
[32] M. Neuhaus, H. Bunke, Bridging the Gap Between Graph Edit Distance and Kernel Machines, World Scientific Publishing, 2007.
[33] D. Pal, P.R. Rao, A tool for fast indexing and querying of graphs, in: Proceedings of the 20th International Conference on World Wide Web, Hyderabad, India, 2011, pp. 241–244.
[34] J.B. Pereira-Leal, A.J. Enright, C.A. Ouzounis, Detection of functional modules from protein interaction networks, Proteins, Struct. Funct. Bioinform. 54 (1) (2004) 49–57.
[35] M.O. Rabin, Fingerprinting by random polynomials, Technical Report TR 15-81, Harvard University, Cambridge, MA 02138, 1981.
[36] P. Rao, B. Moon, Locating XML documents in a peer-to-peer network using distributed hash tables, IEEE Trans. Knowl. Data Eng. 21 (12) (December 2009) 1737–1752.
[37] K. Riesen, H. Bunke, Approximate graph edit distance computation by means of bipartite graph matching, J. Image Vis. Comput. 27 (7) (June 2009) 950–959.
[38] H. Shang, X. Lin, Y. Zhang, J.X. Yu, W. Wang, Connected substructure similarity search, in: Proc. of the 2010 SIGMOD Conference, Indianapolis, 2010, pp. 903–914.
[39] H. Shang, Y. Zhang, X. Lin, J.X. Yu, Taming verification hardness: an efficient algorithm for testing subgraph isomorphism, in: Proc. of the 34st VLDB Conference, 2008, pp. 364–375.
[40] Y. Tian, R.C. McEachin, C. Santos, D.J. States, J.M. Patel, SAGA: a subgraph matching tool for biological graphs, Bioinform. J. 23 (2) (2007) 232–239.
[41] Y. Tian, J.M. Patel, TALE: a tool for approximate large graph matching, in: Proc. of the 24th ICDE Conference, Cancun, 2008, pp. 963–972.
[42] D. Ucar, S. Parthasarathy, S. Asur, C. Wang, Effective pre-processing strategies for functional clustering of a protein–protein interactions network, in: 5th IEEE Symposium on Bioinformatics and Bioengineering, 2005, pp. 129–136.
[43] J.R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[44] A. van Rooij, H.S. Wilf, The interchange graph of a finite graph, Acta Math. Hung. 16 (3–4) (1965) 263–269.
[45] J.T. Wang, K. Zhang, G.-W. Chirn, Algorithms for approximate graph matching, Inf. Sci. 82 (1–2) (1995) 45–74.
[46] P. Willett, J. Barnard, G. Downs, Chemical similarity searching, J. Chem. Inf. Comput. Sci. 38 (6) (1998) 983–996.
[47] D.W. Williams, J. Huan, W. Wang, Graph database indexing using structured graph decomposition, in: Proc. of the 23th ICDE Conference, Istanbul, 2007, pp. 976–985.
[48] R.J. Wilson, Introduction to Graph Theory, Longman, 1999.
[49] X. Yan, P. Yu, J. Han, Graph indexing: a frequent structure based approach, in: Proc. of the 2004 SIGMOD Conference, 2004.
[50] X. Yan, P.S. Yu, J. Han, Substructure similarity search in graph databases, in: Proc. of the 2005 SIGMOD Conference, Baltimore, 2005, pp. 766–777.
[51] X. Yan, F. Zhu, J. Han, P.S. Yu, Searching substructures with superimposed distance, in: Proc. of the 22th ICDE Conference, Atlanta, 2006, pp. 88–99.
[52] Z. Zeng, A.K. Tung, J. Wang, J. Feng, L. Zhou, Comparing stars: on approximating graph edit distance, in: Proc. of the 35st VLDB Conference, Lyon, France, 2009.
[53] S. Zhang, M. Hu, J. Yang, TreePi: a novel graph indexing method, in: Proc. of the 23th ICDE Conference, Istanbul, 2007, pp. 966–975.
[54] S. Zhang, S. Li, J. 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, Saint Petersburg, Russia, 2009, pp. 192–203.
[55] S. Zhang, J. Yang, W. Jin, SAPPER: subgraph indexing and approximate matching in large graphs, in: Proc. of the 36th VLDB Conference, Singapore, 2010, pp. 903–914.
[56] P. Zhao, J.X. Yu, P.S. Yu, Graph indexing: tree + delta >= graph, in: Proc. of the 33rd VLDB Conference, 2007, pp. 938–949.
[57] L. Zou, L. Chen, J.X. Yu, Y. Lu, A novel spectral coding in a large graph database, in: Proc. of the 11th Intl. Conference on Extending Database Technology, 2008.

Please cite this article in press as: D. Pal et al., Fast processing of graph queries on a large database of small and medium-sized data graphs, J. Comput. Syst. Sci. (2016), http://dx.doi.org/10.1016/j.jcss.2016.04.002