SING: Subgraph search In Non-homogeneous Graphs

Raffaele Di Natale [1], Alfredo Ferro [1], Rosalba Giugno [1], Misael Mongiovì [1], Alfredo Pulvirenti [1], and Dennis Shasha [2]

[1] Dipartimento di Matematica ed Informatica
Università di Catania
95125 Catania, Italy
{dinatale,ferro,giugno,mongiovi,pulvirenti}@dmi.unict.it

[2] Courant Institute of Mathematical Sciences
New York University
New York 10012, USA
shasha@cs.nyu.edu

ABSTRACT

Finding the subgraphs of a graph database that are isomorphic to a given query graph has practical applications in several fields, from cheminformatics to image understanding. Since subgraph isomorphism is a computationally hard problem, indexing techniques have been intensively exploited to speed up the process. Such systems filter out those graphs which cannot contain the query, and apply a subgraph isomorphism algorithm to each residual candidate graph. Traditionally, however, the applicability of such process is limited to databases of small graphs, because their filtering power degrades on large graphs. In this paper, SING (Subgraph search In Non-homogeneous Graphs), a novel indexing system able to cope with large graphs, is presented. The method uses the notion of *feature*, which can be a small subgraph, subtree or path. Each graph in the database is annotated with the set of all its features. The key point is to make use of feature locality information. This idea is used to both improve the filtering performance and speed up the subgraph isomorphism task. Extensive tests on chemical compounds, biological networks and synthetic graphs show that the proposed system outperforms the most popular systems in query time over databases of medium and large graphs. Other specific tests show that the proposed system is effective for single large graphs.

General Terms

graph search, indexing, subgraph isomorphism, large graphs

1. INTRODUCTION

Graphs naturally model a multitude of complex objects in the real world. A chemical compound can be represented by a graph where atoms are vertices and bonds are edges. Biological networks model the complex of interactions among components in cells, (e.g. proteins, genes, metabolites). Social networks, the web, the water system and the power grid are all represented by graphs. A basic operation is the search of a query graph in a target graph or, more generally, in a database of graphs. Searching a molecular structure in a database of molecular compounds is useful to detect molecules that preserve chemical properties associated with a well known molecular structure. This can be used in screening and drug design. Searching subnetworks in biological networks helps to identify conserved complexes, pathways and motifs among species, and assist in the functional annotation of proteins and other cell components.

The problem of searching for a query graph in a target graph is called *subgraph isomorphism* and is known to be NP-complete. Since the subgraph isomorphism test is expensive, screening all graphs of a large database can be unfeasible. Recently, indexing techniques for databases of graphs have been developed with the purpose of reducing the number of subgraph isomorphism tests involved in the query process. In a *preprocessing* phase the database of graphs is analyzed and an index is built. A query is processed in two phases. In the *filtering* step the index is used to discard the graphs of the database which cannot contain the query, producing a small set of *candidate* graphs. The set of candidates is then verified (*verification* step) by a subgraph isomorphism algorithm and all the resulting matches are reported.

Most graph indexing tools are based on the concept of *feature*. Depending on the particular system, a feature can be either a small graph [7, 1, 6], a tree [4] or a path [3, 2]. The filtering property is based on checking whether the features of the query are contained in each target graph. In the preprocessing phase the database of graphs is scanned, the features are extracted from each graph and stored in the index data structure. During the filtering phase, the features are extracted from the query and the index is probed in order to discard all graphs which do not contain some feature of the query.

Existing indexing techniques are effective on databases of small graphs but they become unfeasible when applied to huge graphs. The reason is that features that may be rare in small graphs are likely to be found in enormous graphs just by chance. This implies that filtering systems based only on the presence or number of features are not effective for large graphs. Moreover the subgraph isomorphism test over a large graph is extremely expensive. Unfortunately, alternative indexing systems which do not make use of features [4, 10] show similar problems on large graphs.

To make the verification phase faster, GraphGrep [3] stores all the feature occurrences of each graph, and discards the part of the graph which does not contain features of the query thus restricting the search to small portions of the target graph. However, this produces a larger index which is more difficult to manage and can lead to a reduction in filtering performance. Furthermore, the features of the query often occur in many parts of the graphs, reducing the filtering power.

In this paper, a novel approach to cope with large graphs is proposed. The present approach makes use of paths as features. In contrast to systems that use more complex features such as subgraphs or subtrees, our index includes all paths of bounded length. The position of a feature within the graph is considered. This additional information is used to both improve the filtering power and guide the verification phase allowing an effective pruning of the search tree. In contrast to GraphGrep, only the starting point of a feature is stored and bit arrays are used to reduce the index size. Furthermore this information is used to optimize the verification phase. Notice that this approach cannot be used for graph features since graphs have no starting points. Although a similar approach could be used for tree features (using the roots as starting points), the resulting preprocessing time would be higher since enumerating subtrees is much more expensive than enumerating paths. Despite using path features, our system is effective in capturing the topology of graphs and it is shown to perform better than existing systems in terms of query processing time, while keeping the size of the index comparable. An exhaustive experimental analysis on real and synthetic data shows that the proposed system is efficient and effective on both databases of small graphs and single large graphs.

The paper is organized as follows. In Section 2 the basic concepts are introduced. In Section 3 both feature-based and non-feature-based graph indexing systems are reviewed. Feature based graph indexing systems are considered in A unified framework and the differences among them are discussed. Section 4 presents the novel graph indexing system. In Section 5 the results of the experimental analysis are reported. Comparisons with the most popular systems over databases of chemical compounds show that our system is faster in processing queries of size greater than 4 on a database including large molecules. Additional results on gene regulatory networks and synthetic data are reported. For these data the proposed system outperforms all the other tools in terms of query time. Section 6 concludes the paper and addresses future directions.

2. PRELIMINARIES

This paper considers undirected node-labeled graphs. However, the concepts introduced in what follows can be easily extended to edge-labeled and directed graphs. An undirected labeled graph (in what follows simply a graph) is a 4-tuple $g = (V, E, \Sigma, l)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges (a symmetric binary relation on V), Σ is the alphabet of labels and $l : V \to \Sigma$ is a function which maps each vertex onto a label. If $e = (v_1, v_2)$ is an edge, then v_1 and v_2 are called its *endpoints*. We set $size(g) = |E|$ and indicate with $\mathcal{G}$ the set of all possible graphs. A graph $g_1 = (V_1, E_1, \Sigma, l_1)$ is said to be a subgraph of another graph $g_2 = (V_2, E_2, \Sigma, l_2)$ iff $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

Given two graphs $g_1 = (V_1, E_1, \Sigma, l_1)$, $g_2 = (V_2, E_2, \Sigma, l_2)$ an *isomorphism* between g_1 and g_2 is a bijection $\phi : V_1 \to V_2$ so that:

- $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$
- $l_1(u) = l_2(f(u)) \forall u \in V_1$

A *subgraph isomorphism* between g_1 and g_2 is an isomorphism between g_1 and a subgraph of g_2. A graph g_1 is said to be isomorphic to another graph g_2 if there exist an isomorphism between g_1 and g_2. For the sake of simplicity we say also that g_1 is equivalent to g_2 and write $g_1 \approx g_2$. Notice that $\approx$ is an equivalence relation on $\mathcal{G}$. A graph g_1 is said to be subgraph isomorphic to another graph g_2 if there exist a subgraph isomorphism between g_1 and g_2. In this case we say that g_1 is *contained* in g_2 and write $g_1 \precsim g_2$.

In this paper, the following two problems will be discussed:

First_query_occurrence problem: Given a database of n graphs $D = \{g_1, g_2, ..., g_n\}$ and a query graph q, executing the query q on D is equivalent to find all graphs g of D such that q is subgraph isomorphic to g. In the following we assume, without loss in generality, that all graphs of D and the query graph, share the same alphabet Σ.

All_query_occurrences problem: Given a database of n graphs $D = \{g_1, g_2, ..., g_n\}$ and a query graph q, executing the query q on D is equivalent to find all subgraph isomorphisms between q and elements of D.

We will make extensive use of the notion of feature. Features are formally introduced by the following definition.

Definition 1. Let $\mathcal{G}$ be the set of all possible graphs in a given alphabet of labels. A set $\mathcal{F}$ is a *set of features on* $\mathcal{G}$ iff there exists a binary relation $is_a_feature \subseteq \mathcal{F} \times \mathcal{G}$ such that the following property holds (*graph upward monotonicity*):

$$\forall f \in \mathcal{F}, q, g \in \mathcal{G}, \quad is_a_feature(f, q) \wedge q \precsim g \to is_a_feature(f, g)$$

In what follows $is_a_feature(f, g)$ is expressed by saying that g *contains* f.

Every set of features defines a *pruning rule* for the subgraph isomorphism problem:

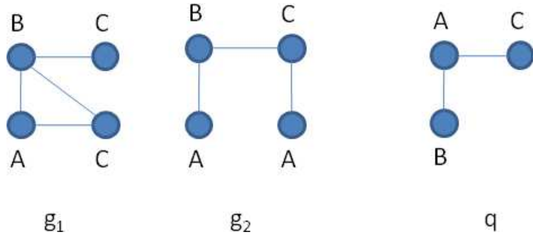

Figure 1: A database of two graphs g_1, g_2 and a query q. $q \precsim g_1$ but $q \not\precsim g_2$.

Pruning rule 1. If $is_a_feature(f, q)$ and $\neg\, is_a_feature(f, g)$ then q cannot be subgraph isomorphic to g.

Examples of set of features are:

- The set $Paths_{\leq k}$ of all labeled paths of length $\leq k$. Here a labeled path is the sequence of labels.
- The set $Subtrees_{\leq k}$ of all labeled subtrees of depth $\leq k$.
- The set $Subgraphs_{\leq k}$ of all labeled subgraphs of size $\leq k$.

This paper considers the set of features $Paths_occ_{\leq k}$ of pairs (p, n), where p is a labeled path of length $\leq k$ and n is a lower bound on the number of occurrences of p in the given graph. The corresponding pruning property asserts that if the query graph q contains at least n occurrences of a given labeled path p and g does not contain at least n occurrences of p, then q cannot be subgraph isomorphic to g and g can be pruned.

Notice that in all above examples if a feature f is a subfeature of a given feature f' of g then f' is also a feature of g. The following definition formalizes this notion.

A *downward monotonic* set of features is a partially ordered set of features $(\mathcal{F}, \preceq)$ such that:

$$\forall f, f' \in \mathcal{F}, g \in \mathcal{G},\quad f \preceq f' \wedge is_a_feature(f', g) \rightarrow is_a_feature(f, g)$$

For instance $Paths_{\leq k}$ is a downward monotonic set of features with respect to the subsequence relation between labeled paths. $Paths_occ_{\leq k}$ is downward monotonic with respect to the number of occurrences. However it is not downward monotonic with respect to the subsequence relation. Indeed in Figure 1, (ABC,2) is a feature of g_1 but (AB,2) is not a feature of g_1.

A downward monotonic set of features allows an additional optimization in the pruning process: the pruning rule can be restricted only to maximal features f in the query. This means that no other feature f' in the query can be strictly greater than f in the partial order of features.

3. RELATED WORKS

Table 1: Review of the main graph indexing systems

System	Features	Data mining	All matches
GraphGrep	Path	No	Yes
gIndex	Graphs	Yes	No
FGIndex [1]	Graphs	Yes	No
GDIndex [6]	Graphs	No	Yes
TreePi [8]	Tree	Yes	No
Tree+δ [9]	Tree+graphs	Yes	No
CTree	-	No	No
GCoding	-	No	Yes
SING	Path	No	Yes

Graph indexing systems are based on a filter-and-verification scheme which includes two main phases: (1) preprocessing: an index is built by scanning the database; (2) query processing: the index is probed to efficiently answer the query. The query processing is divided in two sub-steps: filtering and matching. The filtering step prunes all graphs of the database which cannot contain the query graph, generating a set of candidate graphs. The matching step executes a subgraph isomorphism algorithm on all candidate graphs. All graph indexing systems except CTree [4] and GCoding [10] use the concept of feature. Table 1 synthesizes the characteristics of the main graph indexing tools. In the next subsections we briefly survey feature-based and non-feature-based systems respectively.

3.1 Feature-based graph indexing systems

All feature-based graph indexing systems are characterized by choosing a set of features $\mathcal{F}$ and apply Pruning rule 1 to features of $\mathcal{F}$. To prune as many graphs as possible, the graph indexing systems consider a set of features $F_q \subseteq \mathcal{F}$ such that each feature $f \in F_q$ is contained in q, and prune all graphs $g \in D$ which do not contain some feature in F_q. The filter-and-verification scheme is performed in the following way:

- **Preprocessing:** each graph of the database is examined off-line in order to extract all features of $\mathcal{F}$ which are contained in the graph. An inverted index is generated, which maps each feature $f \in \mathcal{F}$ into the set $graph_set(f)$ of all graphs containing f.
- **Query processing:**
  - **Filtering:** The given query q is examined in order to extract a suitable set $F_q \subseteq \mathcal{F}$ of features contained in q. A set of candidate graphs C is then computed by $C = \bigcap_{f \in F_q} graph_set(f)$.
  - **Matching:** Each candidate graph is examined in order to verify that the given query is subgraph isomorphic to it. If the All_query_occurrences problem must be solved, then an exhaustive enumeration of all distinct subgraph matches is executed.

The differences among the various graph indexing systems lie mainly on the choice of the sets $\mathcal{F}$ and F_q. $\mathcal{F}$ can be a set of bounded-size graphs, trees or paths. Since the number of features can be very high, some graph indexing systems

select a restricted feature set from the database. For example gIndex selects frequent subgraphs of bounded size. This operation requires to perform a heavy graph data mining step during the preprocessing phase. A possible choice for F_q is $F_{all} = \{f \in \mathcal{F} | is_a_feature(f, q)\}$. If $\mathcal{F}$ is an ordered feature set, F_q can be chosen, without loss in pruning power, to be the set F_{max} of all maximal features in F_{all}. It is also possible to choose any set $F_q : F_{max} \subseteq F_q \subseteq F_{all}$. This is the choice made in SING.

Some indexing systems consider also more effective pruning rules based on the number of feature occurrences and the distances between features. Some systems define compact representations of the index. A description of the various indexing systems follows, with a discussion about positive and negative aspects of the various choices.

3.1.1 Graph features

Let $\mathcal{G}_{/\approx}$ be the partition of $\mathcal{G}$ induced by graph isomorphism. Given two classes $G_1, G_2 \in \mathcal{G}_{/\approx}$, we say that $G_1 \precsim G_2$ if $g_1 \precsim g_2$ for some $g_1 \in G_1$ and $g_2 \in G_2$. This is equivalent to saying that $g_1 \precsim g_2$ for every $g_1 \in G_1$ and $g_2 \in G_2$. Notice that $\mathcal{G}_{/\approx}$ and each subset of it are partially ordered by the relation $\precsim$.

Some systems such as gIndex [7], GDIndex [6] and FGIndex [1] use graphs as features. They consider a set of features $\mathcal{F} = \mathcal{G}_{/\approx}$. All isomorphic graphs are considered as a unique feature represented by their equivalence class. The main advantage of using graph features is that they are more suitable to capture the topological structure of graphs. Consequently they tend to produce a fewer number of candidates. On the other hand, some problems need to be faced.

First, a simple representation of the equivalence classes of $\mathcal{G}$ must be found. To efficiently manage such classes, the graph-features-based indexing systems code each graph into a string, called *canonical label*, which is invariant with respect to isomorphism. Hence, each class in $\mathcal{G}$ is represented by its unique canonical label. Similarly to graph isomorphism, the canonical labeling problem is not known whether it is NP-hard or not. However some proposed practical implementations show good performance [5].

Another problem is that the number of graph features grows exponentially with the graph size, leading to a large index which degrades the performance of the preprocessing and filtering phases. To solve this problem gIndex and FGIndex choose as features a set of frequent subgraphs. gIndex considers also the concept of *discriminative subgraphs* to further reduce the number of features. All these approaches require to perform a heavy data mining step in the preprocessing phase, leading to a loss of efficiency. Moreover, when it comes to coping with large graphs the mining step may become unfeasible. FGIndex uses a small index resident in main memory, and stores the remaining index in secondary storage. The authors of FGIndex use a novel concept of *δ tolerance closed frequent subgraph* to distinguish from main-memory-resident features and secondary-memory-resident ones. When the query cannot be performed using only the main-memory-resident index, it is used to identify the blocks of the secondary memory index to be loaded. To avoid expensive disk accesses, a small set of maximal features which cover the whole query graph is selected.

GDIndex enumerates all induced subgraphs contained in each graph of the database. It organizes all the features in a DAG representing the partial order relation $\subseteq$ among features. The size of the index is reduced by avoiding redundancy. Each feature is associated with the set of graphs containing it and not containing any ancestor-feature in the DAG. During the filtering phase, the set of graphs containing a feature can be deduced by the feature-DAG. Enumerating all subgraphs of a graph is very expensive, therefore this approach can be used only on databases of very small graphs.

3.1.2 Tree features

Tree features are easier to be managed since the tree-isomorphism problem can be solved in polynomial time. TreePi [8] is the first attempt to use trees as features. Authors describe a linear-time algorithm for computing the canonical labeling of a tree. They experimentally show that tree features capture the topological structure well enough. Therefore, using them may result in a good compromise between efficiency and effectiveness of filtering. Since trees, unlike graphs, have unique centers, the distance (shortest path) between pairs of features in a graph can be computed. TreePi uses an additional pruning rule based on distances between features to improve the quality of the match. More precisely, this pruning rule is based on the observation that for a query graph to be subgraph isomorphic to a target graph, the distance between each pair of query vertices cannot be lower than the distance between corresponding vertices in the target graph. Tree+δ [9] uses as features both trees and a restricted class small graphs to improve the filtering performance. As for graphs, enumerating all trees of bounded size still produces a large amount of features. Consequently, a restricted set of features needs to be selected by an expensive data mining step.

3.1.3 Path features

3.2 Non-feature based graph indexing systems

4. A NEW APPROACH BASED ON FEATURE LOCATION

5. EXPERIMENTAL RESULTS

6. CONCLUSIONS

7. ACKNOWLEDGMENT

8. REFERENCES

[1] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. *Proceedings of ACM SIGMOD international conference on Management of data*, pages 857 – 872, 2007.

[2] A. Ferro, R. Giugno, M. Mongiovi, A. Pulvirenti, D. Skripin, and D. Shasha. Graphfind: enhancing graph searching by low support data mining techniques. *BMC Bioinformatics*, (9), 2008.

[3] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs, 2002.

[4] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE '06: Proceedings*

of the 22nd International Conference on Data Engineering, page 38, Washington, DC, USA, 2006. IEEE Computer Society.

[5] M. Kuramochi and G. Karypis. Frequent subgraph discovery. *Proceedings of International Conference Data Mining*, pages 313–320, 2001.

[6] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 976–985, 2007.

[7] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Transactions on Database Systems*, 30(4):960–993, 2005.

[8] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. *Proceedings of IEEE 23rd International Conference on Data Engineering*, pages 181–192, 2007.

[9] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta $\leq$ graph. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 938–949. VLDB Endowment, 2007.

[10] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 181–192, New York, NY, USA, 2008. ACM.