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, Catania, Italy
[2]Courant Institute of Mathematical Sciences, New York University, New York, USA

Email: Raffaele Di Natale - dinatale@dmi.unict.it; Alfredo Ferro*- ferro@dmi.unict.it; Rosalba Giugno - giugno@dmi.unict.it; Misael Mongiovì - mongiovi@dmi.unict.it; Alfredo Pulvirenti - pulvirenti@dmi.unict.it; Dennis Shasha - shasha@cs.nyu.edu;

*Corresponding author

Abstract

Background: 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. The applicability of such systems is limited to databases of small graphs, because their filtering power degrades on large graphs.

Results: 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.

Conclusions: 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.

Background

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 [1–3], a tree [4] or a path [5, 6]. 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, 7] show similar problems on large graphs.

To make the verification phase faster, GraphGrep [5] 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 large 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 extensive 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.

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*):*
$\forall f \in \mathcal{F}, q, g \in \mathcal{G}$,

$$is\_a\_feature(f,q) \wedge q \precsim g \rightarrow is\_a\_feature(f,g)$$

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:

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:

$\forall f, f' \in \mathcal{F}, g \in \mathcal{G}$,

$$f \preceq f' \wedge is\_a\_feature(f', g) \rightarrow is\_a\_feature(f, g)$$

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.

Related work

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 [7] 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.

Feature-based graph indexing systems

All feature-based graph indexing systems are characterized by choosing a set of features $\mathcal{F}$ and apply a 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 in 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 the performance of an expensive 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 [5, 6] and the distances between features [8]. Some systems define compact representations of the index [2, 6]. A description of the various indexing systems follows, with a discussion of the positive and negative aspects of the various choices.

Graph features. Some systems such as gIndex [1], GDIndex [3] and FGIndex [2] use graphs as features. They consider a set of features $\mathcal{F} = \mathcal{G}_{/\approx}$, where $\mathcal{G}$ is the universe of graphs and $\mathcal{G}_{/\approx}$ is the partition of $\mathcal{G}$ induced by graph isomorphism. All isomorphic graphs are considered as a single 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 fewer candidates.

Unfortunately, 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 the performance of an expensive 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 impractical. 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, the main-memory index 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 $\preceq$ 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.

Tree features. Tree features are easier to manage since the tree-isomorphism problem can be solved in polynomial time. TreePi [8] is the first attempt to use trees as features. The 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. As shown by authors, a unique center can be defined for a tree. Consequently 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 less than the distance between corresponding vertices in the target graph. Tree+δ [9] uses as features both trees and a

restricted class of small graphs to improve the filtering performance. As for graphs, enumerating all trees of bounded size still produces a large number of features. Consequently, a restricted set of features needs to be selected by an expensive data mining step.

Path features. GraphGrep [5] and GraphFind [6] consider as features all paths of length up to l_p (usually 4). Formally a k-length *path* of a graph $G = (V, E, \Sigma, l)$ is an ordered sequence of vertices $(v_1, v_2, ..., v_k) \in V^k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. We say that a path is *simple* if all of its vertices are distinct. A *path feature* on Σ is an ordered sequence of labels $(a_1, a_2, ..., a_k)$ where $a_1, a_2, ..., a_k \in \Sigma$.

Given a graph $g = (V, E, \Sigma, l)$ and a path feature $f = (a_1, a_2, ..., a_k) \in \Sigma^k$, f is said to be *contained in* g, in symbols $is\_a\_feature(f, g)$, if there is a simple path $(v_1, v_2, ..., v_k) \in V^k$ such that $l(v_i) = a_i$ for $1 \leq i \leq k$. In this case $(v_1, v_2, ..., v_k)$ is called a *path occurrence* of f starting from v_1.

Fixed an integer $l_p > 0$, the set $\mathcal{P} = \bigcup_{1 \leq k \leq l_p} \Sigma^k$ is a partially ordered feature set with respect to the relation $\preceq$ defined by:

$(a_1, ...a_n) \preceq (b_1, ..., b_m)$ if $n \leq m$ and $a_i = b_i \forall i = 1 \ldots n$.

Therefore, a Pruning rule 1 can be used to select candidate graphs.

To improve the quality of filtering, the number of path occurrences of each path feature is stored in an inverted index. When a query q is processed, a set F_q of path features is extracted from it and the number of occurrences of each path feature in the query is compared to the corresponding number of occurrences of the same path feature in each graph of the database. A graph is a candidate if the number of occurrences of each path feature in it is greater than the corresponding number in the query.

GraphGrep also stores the location of each path occurrence in the graphs of the database. Moreover, for each candidate graph, it prunes all parts of the graph which do not contain any path feature of the query. This choice produces an improvement of the matching phase. However the resulting index size is quite large.

The choice of using path features in GraphGrep leads to a very efficient preprocessing phase. On the other hand, it limits the filtering power since paths cannot fully synthesize the topology of graphs.

Non-feature based graph indexing systems

Recently, two non-feature-based graph indexing systems have been proposed. They have been shown to outperform many feature-based indexing systems, probably because they are able to better capture the structure of graphs.

CTree [4] organizes the graphs of the database in a R-tree-like data structure. The leaves represent single graphs while internal nodes represents sets of graphs synthesized in graph structures called *closure graphs*. The closure graph of a set of graphs is obtained in the following way. All graphs in the set are aligned by a fast approximate algorithm, called *Neighbor Biased Mapping*. The vertices of the closure graph are labeled by the sets of labels of the corresponding aligned vertices. Similarly, the edges of the closure graphs are the union of aligned edges. When a query is given, an approximate matching algorithm with no-false-negatives is executed on the closure graphs of the tree in a top-down fashion. When the closure graph of a node has a negative response, all the subtrees rooted at that node are pruned and all its leaf graphs are discarded. The remaining graphs are the candidates, and they can be verified by an exact matching algorithm.

Despite the flexibility and filtering power of CTree, its filtering efficiency is limited since the execution of the approximate matching algorithm is expensive and needs to be applied to many closure graphs.

GCoding [7] uses the eigenvalue properties of the adjacency matrix for pruning graphs. In particular, it makes use of the *Interlacing theorem* which bounds the eigenvalues of the adjacency matrices of matching graphs. In the preprocessing phase, all the graphs of the database are scanned. For each vertex v of a given graph, a *vertex signature* is computed. This computation involves its label, its neighbor's labels together with the higher eigenvalues of the adjacency matrix of the tree rooted on v and representing all n-length paths starting from v. The vertex signatures of a graph are then merged to form the *graph signature*. Finally the graph signatures are organized in a B-tree-like structure for efficient search. When a query q is given, the vertex and the graph signatures of q are computed. The graph signature is used to identify in the B-tree a first set of candidate graphs. Than, a second set of candidate graphs is selected from the first one by discarding all graphs whose vertex signatures do not match the vertex signatures of the query. The correspondence between graph signatures and vertex signatures is defined by applying the Interlacing theorem.

Thanks to its coding strategy based on eigenvalues, GCoding allows a compact representation of the index.

However the computation of eigenvalues is expensive, leading to a slower preprocessing phase. Finally, the loss of information introduced by the chosen coding produces a less effective pruning compared to CTree.

Results and Discussion

Approach

The proposed approach is based on a new feature-locality-based pruning rule that reduces the set of candidates resulting from the application of Pruning rule 1. The new pruning rule captures the structure of the graphs much better, leading to a strong reduction of candidates. Locality information is also used to reduce the search space of the verification phase. Our concept has been inspired by Treepi [8], which uses the concept of distance between features, requiring the computation of all-pair-distances. The SING approach performs less computation, leading to an improvement in efficiency as shown below.
Like GraphGrep and GraphFind, the proposed approach uses path features. Consider the graphs in Figure 1. It is easily verifiable that q is subgraph isomorphic to g_1 but not to g_2. q contains the features (A, B) and (A, C) and they are also contained in both g_1 and g_2. Based on these features the graph g_2 cannot be pruned. Note that the occurrences of both features in q start from the same vertex. The same situation holds in g_1 but not in g_2. More precisely in g_2 there is no vertex from which occurrences of both features start. Consequently vertex labeled A of q cannot match with any vertex of g_2, which can be pruned. The following statements formalize this concept. They are immediate consequences of the definition of subgraph isomorphism. Let $start(f, g)$ be the set of vertices v such that an occurrence of f starts from v in g.

Statement 1 *Given two graphs $q = (V_q, E_q, \Sigma, l_q)$ and $g = (V_g, E_g, \Sigma, l_g)$, let $\phi : V_q \rightarrow V_g$ be a subgraph isomorphism between q and g. For each vertex $v \in V_q$ the following holds:*
$\{f \in \mathcal{F} | v \in start(f, q)\} \subseteq \{f \in \mathcal{F} | \phi(v) \in start(f, g)\}$.

Statement 2 *Given two graphs q, g. If $q \precsim g$ then for each vertex v of q must exist at least a vertex u in g so that*
$\{f \in \mathcal{F} | v \in start(f, q)\} \subseteq \{f \in \mathcal{F} | u \in start(f, g)\}$.

Statement 2 suggest a more effective way to prune the graph database. Given a candidate graph g, for each vertex v of the query graph q, there exists a vertex u of g such that each feature starting from v also starts from u. Consequently, if for some vertex of q there is no such corresponding vertex u, g can be pruned. Statement 1 gives a method to reduce the search space of the matching algorithm. That is, it introduces a

more restrictive condition on the matching pairs of vertices. A detailed description of each phase of the proposed graph indexing system is given in Section Methods.

Experimental Analysis

This section compares the proposed system to the most popular tools. Three different dataset classes are used. Tests on real data coming from a database of small molecules (DTP AIDS Antiviral Screen) and a database of biological networks labeled with gene expressions are performed. We evaluate SING on large graphs by generating a synthetic scale-free network of 2000 nodes and executing several queries of sizes ranging from 4 to 16.

The proposed system was implemented in C++ and compiled with the GNU compiler gcc 3.3.

Experimental analysis was performed on an Intel Xeon with 2 GB of memory using Linux OS.

The other tools used for the comparison are: CTree, GCoding and gIndex.

Molecular data

Experiments on molecular data were performed over the DTP AIDS Antiviral Screen dataset published by the National Cancer Institute [10]. The complete dataset contains about 42000 chemical compounds. The experiment took three subsets containing respectively 8000, 24000 and 40000 graphs. Each compound correponds naturally to a graph whose nodes are the atoms labeled with their atomic symbol. Each simple or multiple chemical bond between two atoms is represented by a single edge.

For each database, a set of queries is generated in the following way. Randomly choose a graph g of the database and one of its vertices v. Starting from v, proceed randomly in a breadth-first fashion until a fixed total number t of edges is reached. This yields groups of 100 queries, each having a number of edges equal to 4, 8, 16 and 32 respectively.

Tables 2 and 3 show the comparison results of SING against CTree and GCoding with respect to preprocessing time and index size, respectively. SING builds the index more rapidly than GCoding. Since CTree does not extract the features from the graphs, its preprocessing time is much lower than the other tools. Figure 2 reports the average number of candidates passing the filtering phase on each query group of a given size. Figures 3 and 4 compares SING in terms of average query processing time against CTree and GCoding, respectively. As above, tests are performed on query groups of a given size. CTree and GCoding consider First_query_occurrence and All_query_occurrences, respectively. Consequently, Figure 3 reports comparisons on the First_query_occurrence problem whereas Figure 4 refers to the All_query_occurrences

problem. SING outperforms all the other tools in all tests except 4-size queries. At size 4, CTree outperforms the other tools, but its filtering and matching steps scale less well as size increases.
The advantage of CTree in the preprocessing phase suggests its employment in applications having a small number query executions per graph. In high query environments, using SING may be more appropriate. Table 4 shows the best-performing tool depending on the number of queries and the query size. The table takes the total of preprocessing and query time into account.
Since the gIndex data mining step is expensive, especially over large graphs, this system was not able to run on these databases. For example, in the experiments reported in [1] all H atoms, together with their bonds, were deleted. In order to compare SING against gIndex, we generated a dataset of small molecular compounds as follows. From the whole AIDS compounds database, 8000 graphs having less than 250 nodes and less than 250 edges were selected. The parameters of gIndex were fixed in such a way to obtain an index size similar to the SING one. The maximum feature size was fixed to 10. gIndex uses a support threshold which grows with the feature size. The maximum support threshold was set to 1.0. gIndex generated an index of 8.5 MB in 391 seconds. SING produced an index of 7.5 MB in 135 seconds. The number of candidates and the filtering time are reported in Figure 5. Since gIndex does not perform the matching task, the total query time is not reported. The filtering time of gIndex tends to be constant with respect to the query size. Compared to gIndex, SING takes more filtering time on low-size queries. On the other hand SING takes less time when the query size increases. The filtering power of SING and gIndex are comparable over small queries. gIndex produces a smaller number of candidates on larger queries at the expense of a longer preprocessing and filtering time.

Biological networks

To evaluate the performance of SING on large networks, we generated a database of gene networks labeled with discretized gene expressions, based on a transcription regulation network of Escherichia Coli annotated with gene expressions. We extracted The largest connected component from the complete network, available with the supplementary material of [11]. Gene expression profiles of 22 samples of the experiment GDS2825 (freely available from NCBI [12]), concerning the analysis of E. Coli K12 strains adapted to grow in benzalkonium chloride (a commonly used disinfectant and preservative) were used. We discretized each gene expression value by mapping it into a set of 5 levels: very low, low, medium, high, very high. Those levels became the node labels of the regulatory networks.
Following Alon [11], we attempted to identify groups of nodes connected with a given topology and

annotated with a certain gene expression level profile. One can use this approach to understand a gene regulation mechanism, by verifying if a given pattern is present in a set of samples, where it occurs and which genes are involved.

We queried the networks database with a set of motifs labeled with gene expression levels using motifs found by U. Alon et. al. [11]. Each vertex was labeled with the gene expression level "very high". Preprocessing time and index size are reported in Table 5. Figure 6 reports the total processing time of each query, named as in [11]. SING outperforms the other methods in query processing time.

Synthetic data

To evaluate the performance of SING over a single large graph, we generated a scale free network of 2000 nodes having about 4000 edges was generated. We then evaluated the query time of SING against Vento [13] and CTree. We did not consider the other tools because the filtering phase is useless for a single graph and the verification phase is performed by Vento. SING generated a 1.2 MB index in 35 seconds, while CTree generated a 88 KB index in 0.5 seconds. The query time is reported in Figure 7.

Conclusions

We have introduced a new algorithm for graph search called SING and compared it with its most popular competitors. SING performs filtering to discard graphs and additional semantic filtering during the detailed graph search. Our experiments suggested that CTree should be used when few queries need to be performed on the graphs. In contrast, using SING is advantageous when a high number of queries need to be performed, especially on databases of large graphs. On databases of small graphs, gIndex may reduce the number of graphs to be verified at the expense of a longer elaboration time. Figure 8 depicts a decision tree which suggest the system to be used over different scenarios. When a lot of queries need to be executed on databases of medium and large graphs, our tests suggest that the employment of SIGMA is the most performing choice.

Methods

In this section we report a detailed description of the proposed system. Our indexing system is based on the classical Pruning rule 1 and the new pruning rule introduced in Section Results. The three steps of the filter-and-verification scheme are discussed separately.

Preprocessing

The preprocessing phase constructs two data structures (for details, see Figure 8). First, it constructs a global index that maps each path feature to the set of graphs containing it ($GI[f]$) and the number of occurrences of that path feature in each graph. The second index maps each feature to the set of starting vertices of all its occurrences. $LI[g][f][v] = 1$ if an occurrence of the feature f in the graph g starts from v. Otherwise $LI[g][f][v] = 0$.

Construction visits all graphs of the database and enumerates, using a depth-first strategy, all feature paths contained in each graph. That is, starting from each vertex it visits all paths of depth at most l_p, where l_p is a fixed threshold (usually $l_p \leq 10$, by default $l_p = 4$).

The structures GI and LI can be implemented using hash tables. Therefore the average complexity of the above algorithm in the database D is $O(d_m^{(l_p-1)} \cdot v_a \cdot |D|)$, where d_m is the maximum degree of the vertices and v_a is the average number of vertices in graphs of D. In our implementation the structures GI and LI are binary trees, so the complexity is $O(d_m^{(l_p-1)} \cdot v_a \cdot |D| \cdot log(|D|) \cdot log(|\Sigma^{l_p}|))$ in the worst case.

In the implementation, $LI[g][p]$ is represented as a bit array whose size is equal to the number of nodes of g. This choice both reduces the index space and produces faster filtering (Section Second step filtering below gives details).

Filtering

The filtering phase applies Pruning rule 1 and Statement 2 to prune the graphs of the database which cannot contain the query (see Figure 9 for details). Structure FQ is the set of features contained in the query, extracted by using a depth-first search strategy similar to the one used in preprocessing. The procedure looks only for maximal paths within the query, discarding all paths that are prefixes of a larger path. FVQ associates with each vertex the set of features starting from that vertex. This is used in the second filtering step.

The first step of the pruning procedure computes the set $C_1 = \bigcap_{f \in FQ} \{g \in GI[f] : GI[f][g] \geq FQ[f]\}$. This retains graphs only if they have all features in the query. In addition, if a graph does not have at least as many occurrences of each feature as the query does, it is discarded.

If FQ and FVQ are represented using hash tables, the average complexity of extracting query features is $O(d_m^{(l_p-1)} \cdot |V_q|)$, where d_m is the maximum degree of the vertices and V_q is the set of query vertices. The complexity of the first step filtering is $O(|\Sigma^{l_p}| \cdot |D|)$ using hash tables. In practice the complexity is lower since the set of graphs associated to a feature is lower than $|D|$ and not all possible path features occur in

the database.

Second step filtering

The first step of filtering takes into account only the occurrences of a feature in the database graphs. The second step filtering uses locality information to further prune the database. For each graph g which passes the first step filtering test, a mapping between query vertices and vertices of g is computed by the following procedure. Let v be a vertex of the query and $FVQ[v]$ be the previously computed set of features starting from v. The algorithm computes $M[g][v] = \bigcap_{f \in FVQ[v]} start(f, g)$ as the set of vertices of the graph g compatible with v. If for some vertices v we have $M[g][v] = \emptyset$ then the graph g is discarded. Statement 2 guarantees the correctness of this second filtering step.

Since LI is implemented using bit arrays, the set $M[g][v]$ can be efficiently computed by the logical AND operation. The complexity is $O(|C_1| \cdot |V_q| \cdot |\Sigma^{l_p}| \cdot v_a)$ if M is implemented by either vectors or hash tables. In practice, this is an extremely fast operation on modern hardware.

Matching

To each candidate graph, the VF2 [13] subgraph matching algorithm is applied. VF2 is a combinatorial search algorithm which spawns a search tree by branching states and makes use of a set of feasibility rules to prune the search. Every state of VF2 consists of a partial match between the query and the target graph. Starting from an initial state consisting of an empty match, VF2 produces a sequence of states by increasing the size of the partial match. Each step generates a new state by adding a pair of corresponding vertices to the partial match. When the partial match cannot be extended, the algorithm backtracks. To decide if two vertices can be matched, VF2 uses a set of topological feasibility rules and a semantic feasibility rule (largely based on label comparison).

SING replaces the semantic compatibility criterion between nodes with a more efficient test to reduce the breadth of the search tree. As described in Section Second step filtering, for each vertex v of the query, a set $M[g][v]$ of compatible vertices of the graph g is computed. $M[g][v]$ represents the set of vertices of g which can be matched to v. In fact, by definition if a vertex v' of the graph g does not belong to $M[g][v]$, there is at least one feature f such that $v \in start(f, q)$ and $v' \notin start(f, g)$. It follows by Statement 1 that no subgraph isomorphism ϕ between q and g can map v into v'. This means that the pair (v, v') cannot be involved in any match.

SING solves both the First_query_occurrence and the All_query_occurrences problems. In the

First_query_occurrence case, the matching algorithm stops when the first match of the graph is found.

Acknowledgements

We would like to thank the author of gIndex, CTree and GCoding for having kindly provided their tools for comparison purpose. Authors were in part supported by PROGETTO FIRB ITALY-ISRAEL grant n. RBIN04BYZ7: "Algorithms for Patterns Discovery and Retrieval in discrete structures with applications to Bioinformatics" and by the Sicily Region grants PROGETTO POR 3.14: "Ricerca e Sviluppo suite di programmi per l'analisi biologica, denominata: BIOWARE".

References

1. Yan X, Yu PS, Han J: **Graph Indexing Based on Discriminative Frequent Structure Analysis**. *ACM Transactions on Database Systems* 2005, **30**(4):960–993.
2. Cheng J, Ke Y, Ng W, Lu A: **Fg-index: towards verification-free query processing on graph databases**. *Proceedings of ACM SIGMOD international conference on Management of data* 2007, :857 – 872.
3. Williams DW, Huan J, Wang W: **Graph Database Indexing Using Structured Graph Decomposition**. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on* 2007:976–985, [http://dx.doi.org/10.1109/ICDE.2007.368956].
4. He H, Singh AK: **Closure-Tree: An Index Structure for Graph Queries**. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, Washington, DC, USA: IEEE Computer Society 2006:38.
5. Giugno R, Shasha D: **Graphgrep: A fast and universal method for querying graphs** 2002, [http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.2381].
6. Ferro A, Giugno R, Mongiovi M, Pulvirenti A, Skripin D, Shasha D: **GraphFind: enhancing graph searching by low support data mining techniques**. *BMC Bioinformatics* 2008, (9).
7. Zou L, Chen L, Yu JX, Lu Y: **A novel spectral coding in a large graph database**. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, New York, NY, USA: ACM 2008:181–192.
8. Zhang S, Hu M, Yang J: **TreePi: A Novel Graph Indexing Method**. *Proceedings of IEEE 23rd International Conference on Data Engineering* 2007, :181–192.
9. Zhao P, Yu JX, Yu PS: **Graph indexing: tree + delta $\leq$ graph**. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment 2007:938–949.
10. **NCI DTP Antiviral Screen data. http://dtp.nci.nih.gov/docs/aids/aids_data.html**.
11. Kashtan N, Itzkovitz S, Milo R, Alon U: **Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs**. *Bioinformatics* 2004, **20**(11):1746–1758.
12. **National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/**.
13. Cordella L, Foggia P, Sansone C, Vento M: **A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs**. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2004, **26**(10):1367–1372.

Figures

Figure 1 - Graph samples

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

Figure 2 - Candidates

Number of candidates over databases of molecular compounds

Figure 3 - Query time First_query_occurrence

Total query time over databases of molecular compounds. The tools solve the First_query_occurrence problem.

Figure 4 - Query time All_query_occurrences

Total query time over databases of molecular compounds. The tools solve the All_query_occurrences problem.

Figure 5 - Comparison with gIndex

Comparison with gIndex over a dataset of 8000 small molecular compounds.

Figure 6 - Results on biological networks

Query time over biological networks.

Figure 7 - Results on a single graph

Query time over a single large graph.

Figure 8 - Decision tree

A decision tree showing the best performing system on different scenarios. SING is the best choice on applications where a lot of queries need to be processed on databases of medium and large graphs.

Figure 9 - Preprocessing

Preprocessing algorithm.

Figure 10 - Filtering

Filtering algorithm.

Tables

Table 1 - Graph indexing systems

Review of the main graph indexing systems.

System	Features	Data mining	All matches
GraphGrep	Path	No	Yes
gIndex	Graphs	Yes	No
FGIndex [2]	Graphs	Yes	No
GDIndex [3]	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

Table 2 - Preprocessing time

Preprocessing time on AIDS molecular compounds. The times are expressed in seconds.

Database size(kb)	CTree	GCoding	SING
8000	8	642	149
24000	25	1948	452
40000	42	2960	755

Table 3 - Index size

Index Size on AIDS molecular compounds. The sizes are expressed in KB.

Database size	CTree	GCoding	SING
8000	13844	6687	8445
24000	41372	20088	25279
40000	70208	30651	42830

Table 4 - Winner table

The best-performing system depending on the number of queries of a given size. Experiments on AIDS database 40,000 molecular compunds. Comparison between SING and CTree and GCoding.

# of queries	size 4	size 8	size 16	size 32
≤ 70	CTree	CTree	CTree	CTree
71-147	CTree	CTree	CTree	**SING**
148-157	CTree	CTree	**SING**	**SING**
≥ 158	CTree	**SING**	**SING**	**SING**

Table 5 - Preprocessing over biological networks

Preprocessing time and index size over biological networks

Tool	Index size (KB)	Preprocessing time (sec)
GraphFind	1252	23
CTree	292	1.3
GCoding	85	101