Information Sciences 261 (2014) 116–131

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

SQBC: An efficient subgraph matching method over large and dense graphs

Weiguo Zheng [a,b], Lei Zou [b,*], Xiang Lian [c], Huaming Zhang [d], Wei Wang [e], Dongyan Zhao [b]

[a] State Key Laboratory of Software Engineering, Wuhan University, LuoJiaShan, Wuhan, PR China
[b] Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, PR China
[c] Department of Computer Science, University of Texas – Pan American, 1201 West University Drive, Edinburg, TX 78539, USA
[d] Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899, USA
[e] Department of Electronic Technology, Engineering University of CAPF, Xi'an, PR China

ARTICLE INFO

Article history:
Received 22 November 2012
Received in revised form 9 July 2013
Accepted 4 October 2013
Available online 21 October 2013

Keywords:
Algorithm
Large network
Database
Graph theory
Subgraph isomorphism
Index strategy

ABSTRACT

Recent progress in biology and computer science have generated many complicated networks, most of which can be modeled as large and dense graphs. Developing effective and efficient subgraph match methods over these graphs is urgent, meaningful and necessary. Although some excellent exploratory approaches have been proposed these years, they show poor performances when the graphs are large and dense. This paper presents a novel Subgraph Query technique Based on Clique feature, called *SQBC*, which integrates the carefully designed clique encoding with the existing vertex encoding [40] as the basic index unit to reduce the search space. Furthermore, *SQBC* optimizes the subgraph isomorphism test based on clique features. Extensive experiments over biological networks, RDF dataset and synthetic graphs have shown that *SQBC* outperforms the most popular competitors both in effectiveness and efficiency especially when the data graphs are large and dense.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

Graph is fascinating and superior to other data structures to model complex relationships among objects. For example, a chemical compound can be represented as a graph, where each vertex is an atom and each edge is a chemical bond. The complex interactions among components in cells (e.g. proteins, genes, metabolites) can also be modeled as graphs, such as pathways and protein–protein interaction networks.

Recently, the growing popularity of biological networks has generated many interesting graph data management problems, such as motif detection, pathway finding and network alignment, among which, a basic operation is to search a query graph over a large graph database, called *subgraph search*, which can assist biologists to find some interesting complexes and motifs in biological networks. However, the key challenge of subgraph search lies in its inherent hardness (subgraph isomorphism is a NP-complete problem [32]) and large graph sizes. PathGuide (http://www.pathguide.org/) contains 325 biological pathway databases, most of which are large and growing rapidly.

There are two scenarios of subgraph queries. One is searching a query graph over a large number of small graphs, and the other is searching a query graph over a single large graph. For example, searching for chemical compounds containing a specific substructure associated with biological activity from a compound database belongs to the former one, where each

* Corresponding author. Address: Institute of Computer Science and Technology, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing 100871, PR China. Tel.: +86 10 82529643.
E-mail address: zoulei@pku.edu.cn (L. Zou).

0020-0255/$ - see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2013.10.003

compound corresponds to a small data graph. Given a structural motif, locating its occurrences over a large biological network belongs to the latter scenario, which is the focus of this work.

Since the volume (amount of data), velocity (speed of data in and out) and variety (range of data types and sources) of scientific and Web data are increasing sharply [14], studying subgraph queries over large and dense graphs is meaningful and has attracted extensive attention. For example, the RDF (Resource Description Framework) data model is a W3C standard to describe resources on the Web and enables data exchange and richer integration [13]. In RDF, data items are represented in the form of triples (subject, predicate, object). As a standard language for querying RDF data, SPARQL is considered as one of the key technologies of the Semantic Web. Fig. 1(a) gives a query example, which retrieves the given name and family name of a scientist, who was born in Germany, and whose spouse was born in Hungary. In this SPARQL query, the "where" clause consists of triple patterns that contain either variables or literals. Actually, each SPARQL query can be represented by a graph [16]. The graph corresponding to this SPARQL query is presented in Fig. 1(b). As a result, any SPARQL query can be equivalently transformed a subgraph query problem, which locates the subgraphs in RDF data graph matching with the query graph. Besides the application in Semantic Web mentioned above, studying the problem of subgraph query over large graphs is useful and promising in social networks [33], bioinformatics [20], and road networks [2] as well.

Subgraph isomorphism has been well studied for almost 40 years, and many classical algorithms have been proposed to solve this problem, such as ULMMAN [32] and VF2 [4]. However, these methods do not work well in subgraph search problem since they need to compute everything on the fly. Therefore, the subgraph search problem has attracted tremendous interests in database community. In order to speed up query processing, existing approaches adopt filtering-and-refinement paradigm and the whole framework has two steps: offline and online. Given a graph database, we first build structural indexes in offline processing. When a query graph is issued, structural indexes are utilized to prune false positives in graph databases. Specifically, in the first scenario, data graphs that cannot contain the query graph are eliminated. In the second scenario, some subgraphs of the target large graph that cannot contain query graph are filtered out to reduce the search space for subgraph isomorphism.

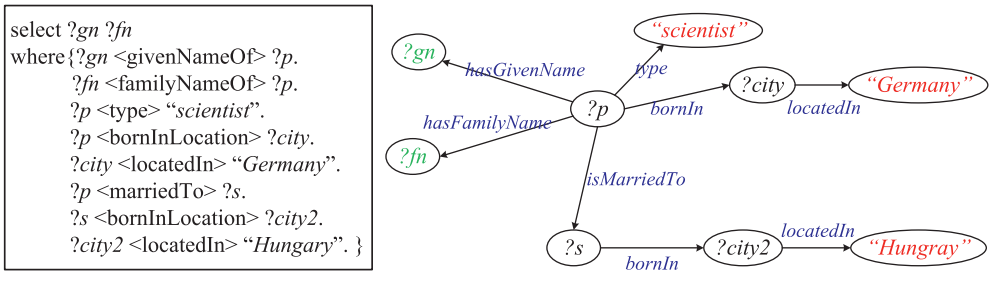
In this paper, we propose a novel *S*ubgraph *Q*uery technique *B*ased on *C*lique (see Definition 3) features (denoted as *SQBC*), which also follows the filtering-and-refinement paradigm. It employs a hybrid index by considering both neighborhood information and structural features. To summary, we make the following contributions in this paper:

- We propose a new subgraph query algorithm over large and dense graphs.
- We exploit a hybrid index by considering both the neighborhood information and structural features, and carefully design the vertex code and clique code to strengthen the pruning power.
- We optimize the subgraph isomorphism test based on clique features.
- Extensive experiments over synthetic and real datasets have shown the effectiveness and efficiency of the proposed algorithm.

The remaining of this paper is organized as follows: Section 2 reviews the related work on subgraph search. The preliminaries and problem definition are introduced in Section 3. Section 4 presents the hybrid index, based on which an effective and efficient subgraph search algorithm is described in Section 5. In Section 6, the experimental results are reported over synthetic and real datasets in terms of efficiency. The analyses of the proposed method are discussed in Sections 7 and 8 concludes this paper.

2. Related work

So far, many subgraph search algorithms have been proposed and their differences lie in pruning strategies [5,9,7,6,17,23,24,27,35–37,39,34,28,20,3,43,38,42,40]. A popular way to perform filtering is to select structural features (e.g., paths, subtrees, subgraphs) as features, and occurrences of these features in graph databases are recorded, like an inverted index. When a query graph Q is given, all features in Q are enumerated and their occurrences in G are utilized to find



(a) A SPARQL query (b) The graph corresponding to the SPARQL query

Fig. 1. A SPARQL query and its corresponding graph.

matches of Q. In GraphGrep [7], all paths up to *maxL* are enumerated as features. SING [20] follows the same method, but proposes a compression technique to reduce the index size. gIndex [35] and FG-index [3] propose to use frequent subgraphs as features instead of paths, since subgraphs can provide larger pruning power. In order to reduce the search space, QuickSI [23] uses the frequencies of tree features that appear in the graph database, and determines an effective search order. However, these feature-based solutions do not work well in a single large graph G, i.e., the specific problem addressed by this paper. For example, it is costly to enumerate all paths in a large dense graph. Mining frequent subgraphs from a large single graph is still an open problem in data mining community [10,41]. Therefore, existing features are not suitable for large dense biological networks. Furthermore, most features are only designed for pruning. They do not consider how to utilize features to speed up subgraph isomorphism test except for GraphGrep.

Another index strategy is based on neighborhood structure around each vertex. The rationale of these methods is as follows: According to the subgraph isomorphism definition, each vertex u in query graph Q must have a mapping vertex v in data graph G if G contains Q. Furthermore, the neighborhood structure around u should also be preserved around v. Based on the intuition, some vertices in G can be eliminated to reduce the search space for subgraph isomorphism algorithms. For example, GCoding [43] computes spectral code for each vertex by considering two-hop subgraph around the vertex. SPath [38] utilizes n-step shortest paths around vertices as basic index units. NOVA [42] proposes a novel vector index (nIndex) based on the neighborhood label distribution of vertices. However, these methods do not show satisfactory performance in large and dense networks, because the n-step local structures around some high-degree vertices are very large, which will lead to large offline computing overhead. SMS [40] designs a vertex code for each vertex according to their neighborhoods, which is lightweight. However, its pruning ability is limited due to the little local information it captures.

In this paper, we propose to employ a hybrid index by considering both the neighborhood information and structural features. First, we propose a simple yet effective strategy to encode neighborhood structure around each vertex, which is similar to that in SMS. Specifically, given a vertex v, we only consider its neighbor labels and the degrees of the neighbors. Different from that in SMS, we incorporate the clique (see Definition 3) information into the signature to improve its pruning ability. The lightweight encoding strategy facilitates both offline index building and online pruning process in dense graphs. Compared with existing approaches, experimental study shows that the pruning power of the lightweight encoding strategy does not degrade significantly.

Furthermore, we find all maximal cliques (a subgraph where each pair of vertices has an edge, formally defined in Definition 3) in data graph G as features. Obviously, if a query graph Q contains a clique, its matches also contain cliques. Therefore, we only need to probe cliques and their neighborhood structures in data graph G. Cliques in biological networks are of interest to biologists because many different problems from bioinformatics have been modeled using cliques [25,22,18]. For example, Samudrala and Moult [22] model protein structure prediction as a problem of finding cliques in a graph whose vertices represent positions of subunits of the protein. Moreover, "clique" features can be used to optimize subgraph isomorphism check. It is easy to check if a query graph Q (or a subgraph of Q) is subgraph isomorphic to a clique by a cheap set operation, based on which, we optimize verification process, i.e., subgraph isomorphism test.

3. Preliminaries and problem definition

In this section, we give the general definition of subgraph isomorphism before presenting the problem definition in this paper. Table 1 lists the frequently-used notations throughout the paper.

A vertex-labeled graph is defined as a 4-tuple $G = \{V, E, L_V, F\}$, where (1) V is a set of vertices; (2) $E \subseteq V \times V$ is a set of edges; (3) L_V is a label set; and (4) F is a function $V \rightarrow L_V$ that assigns a label to each vertex. A biological network can be modeled as a graph, in which each vertex represents a molecule or complex, and each edge denotes the relationship between two vertices. Fig. 1 shows two graphs Q and G, where the letter inside vertices are vertex labels, and $v_1, \ldots, v_6$ and $u_1, \ldots, u_7$ beside vertices are vertex IDs that we introduce to simplify description of the graph.

Table 1
Frequently-used abbreviations.

Notations	Definition and Description
$NB(v)$	Neighbors of vertex v
NDL	Neighbor degree information in terms of labels
$SumLS(c)$	The sum of vertex labels in clique c
IS	Id set
LS	Label set
CMC	The current maximal clique
$SP(v)$	The supervertices of vertex v
$CV(v)$	The candidates of vertex v
SNP	Sequence number of position
SMQ	The match sequence of Q
SMG	The match sequence of G
CCR	Clique coverage rate

Definition 1 (*Graph Isomorphism [40]*). A graph $G = \{V, E, L_V, F\}$ is isomorphic to another graph $G' = \{V', E', L'_V, F'\}$, denoted by $G \approx G'$, if and only if there exists a bijection function g, such that (1) $\forall v \in V$, $F(v) = F'(g(v))$; and (2) $\forall v_1, v_2 \in V, \overrightarrow{v_1 v_2} \in E \Longleftrightarrow \overrightarrow{g(v_1)g(v_2)} \in E'$.

Definition 2 (*Subgraph Isomorphism [40]*). Given two graphs Q and G, Q is subgraph isomorphic to G if and only if Q is isomorphic to at least one subgraph G' of G, and G' is called a *match* of Q in G.

For example, Fig. 2 describes a running example of the query graph Q and the target graph. The letters inside vertices are vertex labels, and $v_1, \ldots, v_6$ and $u_1, \ldots, u_7$ beside vertices are vertex IDs. Q is subgraph isomorphic to G, where the match of Q is denoted by dotted lines.

Since we adopt maximal cliques features in this paper, we give its formal definition here.

Definition 3 (*Clique*). A *clique* is a (sub)graph where each two vertices have an edge, where a *maximal clique* is a clique that is not included in another larger clique.

For example, G has two maximal cliques. One is c_1 that consists of vertices u_1, u_2, u_3 and u_7. The other one consists of vertices u_1, u_6 and u_7.

Problem Statement 1. Given a query graph Q and a single large data graph G, the problem to be addressed in this paper is to find all matches of Q over G, where $|Q| \ll |G|$ [40].

In Problem Statement 1, $|Q|$ and $|G|$ denote the number of vertices in Q and G, respectively. $|Q| \ll |G|$ means that $|G|$ is far larger than $|Q|$, which emphasizes that the problem addressed in this paper is answering subgraph queries over a single large graph.

4. Hybrid index

The main framework for our algorithm is in accordance with the state-of-art paradigm, i.e., construct the index first (offline phase) and then conduct the query processing (online phase). The intrinsic difference of many subgraph search algorithms lies in indexes and pruning strategies. In this section, we first introduce the features to be indexed, and then organize the features in a hybrid index structure.

4.1. Index unit

In this paper, we propose to utilize a hybrid index by considering both vertex neighborhood information and structural features. First, we design a lightweight encoding technique and then propose to mine the maximal cliques as structural features.

4.1.1. Vertex feature

Since subgraph isomorphism is NP-complete, an efficient subgraph query algorithm should reduce the search space as much as possible. In a single large graph, it is very expensive to mine lots of complex structures. Therefore, we design a lightweight index, i.e., vertex feature, which is based on the neighborhood structure of each vertex.

Definition 4 (*Vertex Code*). Given a vertex u in graph G, its vertex code is defined as $C(u) = [L(u), S(u), NDL(u) = \{l_i, (d_{i1}, \ldots, d_{im})\}^+]$, where $L(u)$ is u's label, $S(u)$ is the size of the largest maximal clique containing u, $NDL(u)$ denotes the degree information of u' neighbors in terms of labels, l_i is a vertex label $(d_{i1}, \ldots, d_{im})$ is the degree list of u's neighbor vertices with label l_i and $d_{i1} \geqslant d_{i2} \geqslant \cdots \geqslant dim$. If u is not included in any clique, $S(u) = 0$.

Example 1. The vertex codes of v_4, u_1 and u_6 in Fig. 2 are as follows: $C(v_4) = [a, 0, \{a, (2); b, (2,2)\}]$, $C(u_1) = [a, 4, \{a, (3); b, (4); c, (4); d, (3)\}]$, and $C(u_6) = [a, 3, \{a, (4); b, (4,2)\}]$.

Note that the vertex code here is an extension of the one in [40] by considering the maximal clique features, $S(u)$. Different from existing neighborhood pruning methods, our vertex code is very cheap to generate in dense graphs.

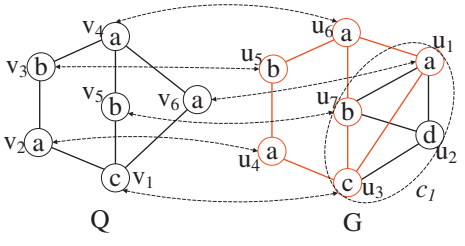


Fig. 2. A running example of the query graph Q and the target graph G.

For the purpose of presentation, we define *presequence* in Definition 5, which is used to generate the candidates of a vertex in query Q.

Definition 5 (*Presequence [40]*). Given two sequences of integers in descending order, $S = (s_1, s_2, \ldots, s_m)$ and $T = (t_1, t_2, \ldots, t_n)$, S is called a *presequence* of T if and only if (1) $m \leqslant n$ and (2) $\forall s_i \in S$, $|\{t_j \,|t_j \geqslant s_i \wedge t_j \in T|\}| \geqslant i$, where $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$.

Example 2 presents three sequences to illustrate the presequence relationship.

Example 2. Given three sequences $S_1 = \{11,9,8,5,3\}$, $S_2 = \{14,13,10,8,5\}$, and $S_3 = \{15,13,9,7,6,4,2\}$. For each element s in S_1, there are enough elements in S_3 that are larger or equal to s. Thus, S_1 is a presequence of S_3. There are only two elements in S_3 that are no smaller than the element "10" in S_2. However, element "10" is the third largest element in S_2. Hence, S_2 is not a presequence of S_3.

It is intuitive that if a vertex u in Q have matches in G, the neighborhood structure around u should be preserved around the mapping vertices. Formally, To test whether a vertex in Q matches a vertex in G or not, we give the definition of *subvertex* based on *presequence*.

Definition 6 (*Subvertex, Supervertex*). Given two vertices v and u in query graph Q and data graph G, respectively, v is a subvertex of u if and only if: (1) $L(v) = L(u)$; (2) $S(v) \leqslant S(u)$; (3) $\forall l_i \in NDL(v)$, $\exists l_j \in NDL(u)$ and $l_i = l_j$ and the degree list $\{d_{i1}, \ldots, d_{im}\}$ of $NDL(v)$ is a presequence of the degree list $\{d_{j1}, \ldots, d_{jn}\}$ of $NDL(u)$. (If v is a *subvertex* of u, we say that u is v's *supervertex*.)

It is straightforward to know, if a vertex v (in query graph Q) is not a *subvertex* of u (in data graph G), u cannot match v. Thus, we can easily find that u_1 cannot match with v_4.

To accelerate the process of finding the candidates of a vertex, an inverted index is constructed, which consists of labels as the keys and the ID lists of the vertices with the corresponding label as the values. Furthermore, we build an inverted index for each vertex v, in which the key is the label of v's neighbors and the value is the ID list of v's neighbors with the corresponding label.

4.1.2. Maximal clique feature

In this paper, we propose to mine maximal cliques as features, since it can support both filtering and verification. (The discussions regarding to the verification will be presented in Section 5.)

After obtaining all *maximal cliques*, we can regard them as specific nodes, and encode them so as to test whether one subgraph (may be a clique) of query Q is subgraph isomorphic to a clique or not.

Definition 7 (*Clique Code*). Given a clique c in graph G, its code is defined as $C(c) = [Num(c), SumLS(c), IS(c) = \{u_1, \ldots, u_m\}, LS(c) = \{l_i, (d_{i1}, d_{i2}, \ldots, d_{ik})\}^+]$, where $Num(c)$ is the number of vertices in clique c, $SumLS(c)$ is the sum of vertex labels in the clique (each distinct label can be hashed to an integer value), $IS(c)$ is the corresponding vertex id set in the clique, l_i is a vertex label and $(d_{i1}, d_{i2}, \ldots, d_{ik})$ is the degree list of vertices with label l_i in clique c.

For example, the clique code of clique c_1 consisting of vertices u_1, u_2, u_3 and u_7 in Fig. 1 is C (c_1) = [4, 394, {u_1, u_2, u_3, u_7}, {a, (4); b, (4); c, (4); d, (3)}] (assume that each vertex label is mapped to its ASCII value).

Definition 8 (*Subclique*). Given two cliques c_q and c_g in a query graph Q and a data graph G, respectively, c_q is a subclique of c_g if and only if (1) $Num(c_q) \leqslant Num(c_g)$; (2) $SumLS(c_q) \leqslant SumLS(c_g)$; and (3) $\forall l_i \in LS(c_q)$, $\exists l_j \in LS(c_g)$ and $l_i = l_j$ and the corresponding degree list $\{d_{i1}, d_{i2}, \ldots, d_{ik}\}$ of $LS(c_q)$ is a presequence of the corresponding degree list $\{d_{j1}, d_{j2}, \ldots, d_{jt}\}$ of $LS(c_g)$.

If c_q is a *subclique* of c_g, c_q is subgraph isomorphic to c_g, and we can also say c_g is a *superclique* of c_q, which is denoted as $c_q \sqsubset c_g$.

Lemma 1. *If Q is subgraph isomorphic to G, each subgraph of Q must be subgraph isomorphic to the corresponding subgraph of G.*

This is similar to the Apriori property in the frequent pattern mining [1]. On the basis of Lemma 1, if Q is subgraph isomorphic to G and there is a clique c_q in Q, the data graph G must contain one superclique of c_q at least; otherwise Q is not subgraph isomorphic to G. So we can locate the possible matches according to the cliques when they exist in Q.

4.2. Index construction

There are two factors needed to be considered in the index construction. On one hand, an important step is to find the candidates efficiently for a clique in the query graph, we should devise an effective index to facilitate the query processing. On the other hand, the local structural information around each clique, which is required in the verification processing, should be preserved. Hence, we devise an inverted index for cliques, which consists of two parts.

According to Definition 8, given a clique c_q in Q and a candidate clique c_g in G, c_g must contain all the labels that c_q contains. A straightforward method is to index the cliques in G with all the labels of a clique being the key entry. However, the method is expensive in terms of space cost especially when the data graph G is dense and contains a large number of cliques.

Thus, we propose an efficient inverted index in terms of both time and space cost as shown Fig. 3(a). Specifically, each element of the first part index is a key-value pair, where vertex label l is the key and the cliques containing the corresponding label are the value, denoted as $\mathtt{I}(l)$. Using the inverted index, we can easily obtain the candidate cliques through Eq. (1), where $cand(c_q)$ represents the candidate cliques of c_q.

$$cand(c_q) = \bigcap_{l \in c_q} \mathtt{I}(l) \tag{1}$$

In order to preserve the local structural information, we record the entrance and neighborhood vertices of each maximal clique as depicted in Fig. 3(b). The entrance vertices of a maximal clique c are the vertices that have at least one neighbor vertex out of the clique, denoted as $EN(c)$. The neighborhood vertices of a maximal clique c are the neighbors of c's entrance vertices, which are not included in clique c. For example, the entrance vertices of clique c_1 in Fig. 1 are $EN(c_1) = \{u_1, u_3, u_7\}$, and its neighborhood vertices are $NB(c_1) = \{u_4, u_6\}$. Furthermore, we also record the neighborhood maximal cliques of each vertex if any.

Considering that the issue of how to enumerate all the maximal cliques in a graph is not our contribution, so we implement a method proposed in the existing study [31]. To make our paper self-contained, we introduce its main idea here. The search process maintains two sets, *CMC* (the current maximal clique) and *SUBG* (candidate vertices to be expanded). Initially, *CMC* is equal to Ø, and *SUBG* is equal to *V*. In the expanding process, choose a vertex $v \in SUBG$ each time, then $CMC = CMC \cup v$, and $SUBG = SUBG \cap NB(v)$, where $NB(v)$ is the neighbor set of vertex v. *CMC* is expanded iteratively until *SUBG* is equal to Ø. In addition, there are some pruning strategies to reduce the search space [21,11,8,30]. After mining all maximal cliques, there are no completely dense regions if each maximal clique is viewed as a vertex, and we call the graph condensed graph.

4.2.1. Time complexity

Since SQBC needs to invoke the operation of mining all the maximal cliques in a graph, its time complexity of offline processing depends on the complexity of enumerating maximal cliques. In the worst case, listing all maximal cliques may require exponential time as there exist graphs with exponentially many maximal cliques [31]. Actually, it is not uncomputable in real graphs. Provided that the average degree is d, listing the maximal cliques containing a vertex v requires $O(d!)$ running time according to the algorithm aforementioned. Thus, the average time complexity of mining all the maximal cliques is $O(n \cdot d!)$. SING [20] employs paths of bounded length as features. In addition, it also preserves the position information of these features. The index of NOVA [42] is a structured as a multi-dimensional vector signature consisting of the label distribution of neighborhood vertices. During the index construction phase, QuickSI [23] requires to mine tree features and organize them as a prefix tree. It is obvious that building indices of SING, NVOA and QuickSI are time and space consuming especially in large and dense graphs. SMS [40] does not employ any structure features, such as subgraphs, trees or paths. Hence, it is time efficient to encode each vertex in its offline phase.

5. Query processing

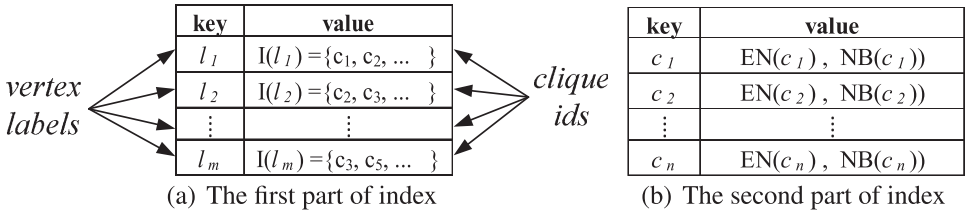
In this section, we discuss the query process based on the index constructed as aforementioned. Given a query graph Q, we generate vertex codes and clique codes, according to the method in the previous section.

A pair of matching sequences *SMQ* and *SMG* are maintained to store the partially matched vertices during query processing. For example, Fig. 4 shows a match of Q by matching sequences *SMQ* and *SMG*. In the traditional subgraph isomorphism algorithms, such as VF2 algorithm [4], one vertex is matched at a time. There may exist many cliques in dense data graphs. Based on these cliques, we can optimize subgraph isomorphism algorithm by matching several vertices at a time. For the ease of presentation, we consider two scenarios of Q, i.e., Q contains cliques or not.

5.1. Process of queries without cliques

At the very beginning of the query process, if there are no cliques in Q, the vertex with the largest degree is considered preferentially, which is pushed into the matching sequence *SMQ* together with its neighbors.

Initially, for each vertex v in a query graph, we need to find candidate vertices matching v. Before presenting the strategy to compute candidates, we introduce a definition *prior vertex*.



(a) The first part of index (b) The second part of index

Fig. 3. Inverted index.

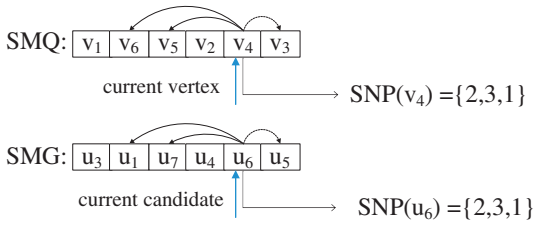


Fig. 4. SNP of current vertices in Q and G.

Definition 9 (*Prior Vertex*). In the match sequence, the prior vertex of v is the vertex that is firstly pushed into *SMQ* (or *SMG*) among the neighbor vertices of v.

According to Definition 9, each vertex has a prior vertex except for the first vertex in the matching sequence. The neighbor vertices and all *Supervertices* (Definition 6) of v are denoted as $NB(v)$ and $SP(v)$, respectively. During the query processing, assume that the current vertex to be matched in a query graph Q is v_i, and the priori vertex of v_i is v_j, which has been matched with vertex u_j in G, the candidate vertices that can match v_i (denoted as $CV(v_i)$) is computed by the following equation.

$$CV(v_i) = NB(u_j) \bigcap SP(v_i) \tag{2}$$

where $NB(u_j)$ denotes neighbor vertices of vertex u_j, and $SP(v_i)$ denotes supervertices of v_i in data graph G.

Let us consider query Q and graph G in Fig. 1. Assume that vertex v_1 (in Q) has been matched to u_3, as shown in SMQ and SMG in Fig. 4. Now, the candidates of v_6 that is a neighbor of v_1 are $CV(v_6) = NB(u_3) \bigcap SP(v_6) = \{u_1\}$.

Obtaining the candidate vertices of v, the process proceeds in depth-first manner iteratively. Each time, we choose one vertex $u \in CV(v)$, and try to push u into the match sequence *SMG*. Before pushing the partial match into match sequences, we need to perform verification as follows.

As we know, if a vertex u in G matches with v in Q, the structural information around v should be preserved around u. In this paper, we propose to use one-step neighborhood information to verify v's candidates. On one hand, we check the neighborhood matched information of u. On the other hand, we also consider the unmatched information so as to filter out as many false alarms as possible. Formally, we give a definition based on which the verification processing can be performed efficiently.

Definition 10 (*Sequence Number of Position*). Given a graph Q and its match sequence *SMQ*, Sequence Number of Position of the current vertex v to be matched is a sequence of number, denoted as $SNP(v) = \{s_1, s_2, \ldots, s_n, m\}$, where $s_1, s_2, \ldots, s_n$ are the positions of v's neighbors that have been matched in *SMQ*, and $s_1 < s_2 < \cdots < s_n$, and m is the number of v's neighbors that have not been matched.

It is easy to build $SNP(v)$ for vertex v. We only need to record the position of a vertex when it is pushed into the matching sequence. Hence, it is straightforward to obtain $SNP(v)$ by traversing the neighbors of vertex v. Similarly, we can also define and build the sequence number of position for vertices in G. Based on $SNP(v)$, we acquire the following lemma to filter out false alarms.

Lemma 2. *Given two vertices v and u in Q and G respectively,* $SNP(v) = \{s_1, s_2, \ldots, s_n, m_1\}$, $SNP(u) = \{t_1, t_2, \ldots, t_k, m_2\}$, *v matches u if and only if (1)* $\{s_1, s_2, \ldots, s_n\}$ *is a subsequence of* $\{t_1, t_2, \ldots, t_k\}$ *and (2)* $m_1 \leqslant m_2$.

Proof. (1) If $\{s_1, s_2, \ldots, s_n\}$ is not a subsequence of $\{t_1, t_2, \ldots, t_k\}$, the former sequence must contain at least one element s_i that is not contained in the latter subsequence. It means that some neighbors of u cannot match with s_i. Therefore, u cannot match with v. (2) If v's unmatched neighbors are more than u's unmatched neighbors, vertex v must contain at least one neighbor v_j that cannot be matched with any neighbor of u. Therefore, much unnecessary work can be avoided, and the query efficiency improves a lot. □

Example 3. If the current vertex to be matched is v_4, *SMQ* and *SMG* for the graphs Q and G (in Fig. 1) are shown as Fig. 2, where u_6 is a candidate vertex to match v_4. $SNP(v_4) = \{2, 3, 1\}$, $SNP(u_6) = \{2, 3, 1\}$. According to Lemma 2, we conclude that v_4 can match u_6.

After matching the current vertex in *SMQ*, all its neighbors that are not in *SMQ* are pushed into the sequence. When all vertices of Q have been matched, a match between Q and G is found.

Lemma 3. *When the number of the matched vertices in SMQ is equal to* $|Q|$, *the pair of match sequence SMQ and SMG is a match mapping between Q and G.*

Proof. When $|SMQ| = 1$, it is obvious that the vertex in *SMG* matches the vertex in *SMQ*. Supposing that when $|SMQ| = k$ and the subgraph consisting of the vertices in *SMQ* matches the corresponding subgraph consisting of the vertices in *SMG*. When $|SMQ| = k + 1$ and the next vertex to be matched is v_{k+1}, according to Lemma 2 all the information of the candidates is checked after matching v_{k+1}. So the subgraph consisting of the vertices in *SMQ* matches the subgraph consisting of the vertices in *SMG* when $|SMQ| = k + 1$. According to the principle of mathematical induction, Lemma 3 is proved. □

As shown in this section, our algorithm combines filtering and verification together so that false positive candidates can be avoided as earlier as possible.

Like traditional subgraph isomorphism algorithm, in the above match processing, one vertex is matched at a time. In order to improve query performance, in our algorithm, several vertices in *Q* can be matched at the same time by utilizing maximal cliques in *G*. Algorithm 1 shows the details.

Algorithm 1. PQwithoutMC(*G*, *Q*, *SMQ*, *SMG*, *i*)

Require: Target Graph *G*, query graph *Q*, *SMQ*, the position i of current vertex v_0 to be matched in *SMQ*, candidate vertex u_0 of v_0.
Ensure: *SMG*.
1: $v_0 \leftarrow$ the vertex at $SMQ_{[i]}$
2: Obtain all the candidates $CV(v_0)$ of v_0 by Eq. (2)
3: **for** each $u_0 \in CV(v_0)$ **do**
4: **if** u_0 does not exist in any clique **then**
5: push u_0 into *SMG*
6: PQwithoutMC(*G*, *Q*, *SMQ*, $i + 1$)
7: **else**
8: $c \leftarrow$ the largest maximal clique that contain u_0
9: $\kappa \leftarrow$ the size of c
10: $n \leftarrow 2$
11: **if** $i + n > |SMQ|$ **then**
12: $n \leftarrow |SMQ| - i$
13: **end if**
14: $S \leftarrow SMQ_{[i]} \ldots SMQ_{[i+n-2]}$
15: push the possible match of S within c into *SMG*
16: PQwithoutMC(*G*, *Q*, *SMQ*, $i + n - 1$) and the search space of $SMQ_{[i+n-1]}$ does not contian the vertices in c
17: $S \leftarrow SMQ_{[i]} \ldots SMQ_{[i+n-1]}$
18: **if** the label set of $S \subseteq$ the label set of c **then**
19: push the possible match of S within c into *SMG*
20: PQwithoutMC(*G*, *Q*, *SMQ*, $i + n$)
21: $n \leftarrow n + 1$
22: **if** $n \leqslant \kappa$ **then**
23: goto step 11
24: **end if**
25: **end if**
26: **end if**
27: **end for**

When trying to match with a vertex v in *Q*, we examine whether its candidate u is included in some maximal clique. If u is included in some maximal clique, we will match several vertices together gradually (Steps 17–24 in Algorithm 1). Meanwhile, to ensure that the results are complete, we need to consider the matches in which a substructure consists of some vertices within a clique c and others outside c (Steps 14–16). Specifically, assume that v (in *Q*) matches with u (in *G*) and u is included in a maximal clique c. v's successors in *SMQ* are denoted as $v'_1, \ldots, v'_n$. We divided the whole search space into several parts:

1. v matches with u and v'_1 does not match with any vertex in c;
2. v matches with u and $\{v'_1\}$ matches with some vertex in c and none of $\{v'_2, \ldots, v'_n\}$ matches with any vertex in c;
3. v matches with u and $\{v'_1, v'_2\}$ match with some vertices in c and none of $\{v'_3, \ldots, v'_n\}$ matches with any vertex in c;
4. ……

Note that, it is easy to determine whether a subgraph of *Q* is subgraph isomorphic to a clique c in *G*, since we only need to perform an inclusion query. For example, given a query graph *Q* and a data graph *G* in Fig. 1. The current vertex to be matched is v_1, and its candidate is u_3. v_1's neighbors are also pushed into match sequence *SMQ*, and v_1's successors in *SMQ* are $\{v_6, v_5, v_2, v_4\}$. Noted that u_1 is included in maximal clique c_1, several branches need to be dealt with as follows.

1. Match v_1 with u_3 and v_6 cannot match any vertex in clique c_1. In this case, there is no candidate that can match v_6, since there exists no vertex (out of c_1) that is adjacent to u_3 is a *supervertex* (see Definition 6) of v_6. The search branch is terminated.
2. We try to match $\{v_1, v_6\}$ with some vertices in clique c_1 and other v_1's successors match vertices outside of c_1. The label set $S_{q=}\{c, a\}$ of $\{v_1, v_6\}$ and the label set $S_c = \{a, b, c, d\}$ of c_1. Since $S_q \subseteqq S_c$, thus, we push a possible match (u_3, u_1) of (v_1, v_6) into *SMG*. Then, the search space for v_5 does not include clique c_1. In this case, there is no candidate can match v_5. The search branch is terminated.
3. We try to match $\{v_1, v_6, v_5\}$ to some vertices in clique c_1 and other v_1's successors match vertices outside of c_1. The label set $S_q = \{c, a, b\}$ of $\{v_1, v_6, v_5\}$, the label set $S_c = \{a, b, c, d\}$ of c_1. $S_q \subseteqq S_c$. So we push a possible match (u_3, u_1, u_7) of (v_1, v_6, v_5) into *SMG*. At this time, the search space for v_2 does not include c_1. In this case, we can find one candidate vertex u_4 that can match v_2. Thus, this search branch will be continued.
4. We try to match $\{v_1, v_6, v_5, v_2\}$ to some vertices in clique c_1. The label set $S_q = \{c, a, b, a\}$ of $[v_1, v_6, v_5, v_2]$, the label set $S_c = \{a, b, c, d\}$ of c_1. $S_q \nsubseteqq S_c$. So there is no any match of $[v_1, v_6, v_5, v_2]$ in c_1. Thus, this search branch is terminated.

The optimization technique will speed up finding candidate partial matching sequences, since it can consider several vertices in a query graph Q at a time, when there exist cliques. When we find the partial sequences, we also need to do the verification according to Lemma 2.

5.1.1. Time complexity

Since the problem of subgraph isomorphism have proven to be NP-complete [32]. In worst case, all of these methods, such as SQBC, SMS, SING, and NOVA, have the exponential running time complexity. However, SING adopts the VF2 [4] algorithm in the verification phase. SMS and NOVA also employ state expanding methods that is trying to speed up the isomorphic enumeration. Nevertheless, SING, SMS and NOVA try to match only a vertex at each partial state. In comparison, utilizing clique features SQBC may match several vertices together at each partial state, which improves the searching efficiency.

5.2. Process of queries with cliques

Generally speaking, we have the analog approach when query graphs contain cliques. The main differences lies in the method of computing candidates and the vertex sequence pushed into *SMQ*.

Given a query graph Q having a clique, we can locate matches of Q in G based on the maximal cliques. Different from the former case, if there are some cliques in the query graph Q, at the very beginning, we select the clique c with largest size, because the clique with largest size may have less candidates. All the vertices of the selected clique are pushed into the match sequence *SMQ*. At this time, all their neighbors those do not exist in *SMQ* yet are pushed into *SMQ*.

When the current vertex v_i to be matched in *SMQ* is in a maximal clique c, its priori vertex and *supervertexes* are v_j and $SP(v_i)$. Assume that the current vertex matching to v_j is u_j and the set of vertices in c's *supercliques* is $SPC(c)$. The candidate set $CV(v_i)$ of v_i can be computed by Eq. (3). Beyond that, nothing else information about vertex v_i need to be checked.

$$CV(v_i) = NB(u_j) \bigcap SP(v_i) \bigcap SPC(c) \tag{3}$$

The rest processes are same with that of queries without maximal cliques. It is clear that we can reduce the search space if the query graph contains cliques. Moreover, we do not need to verify the candidates for vertices (in Q) contained in cliques, because we have checked all the structural and label information around these vertices when we compute their candidates as Eq. (3). Undoubtedly, this will speed up the verification processing.

6. Experimental results

In the section, we evaluate our algorithm (denoted as SQBC) over both synthetic and real datasets. As presented in latest study [15], GADDI [36], SPath [38] and QuickSI [23] are state-of-the-art methods dealing with exact subgraph query over a single large graph. Actually, we have compared SMS with GADDI, SPath and NOVA in [40], and confirm that SMS outperforms these methods significantly. Thus, we compare SQBC with QuickSI [23] (also denoted as QSI), SMS [40] and SING [20] in this paper. SING employs path structures as its index, which does not incorporate encoding of vertices. SMS is our previous work, in which, we do not consider clique features. Our methods have been implemented using standard C++. The executable code of SING is provided by its authors. We also implement the algorithm QuickSI in our best effort. All experiments are conducted on a P4 3.0 GHz machine with 2Gbytes RAM running Linux and 250 GB SATA disk.

6.1. Datasets

To confirm the effectiveness and efficiency of our method, we adopt two synthetic and two real datasets in our experiments. These datasets have different characteristics as described in following sections.

6.1.1. Synthetic datasets

(a) Erdos Renyi (ER) Model. ER model is a classical random graph model, which defines a random graph as N vertices connected by M edges, chosen randomly from $N(N-1)/2$ possible edges. In experiments, we generate a set of small graphs and a large graph. The size of small graphs varies from 1K to 10K. The default average degree is set to be 4. The large graph is composite of 500K vertices and 2.4 million edges.
(b) Scale-Free (SF) Model. A scale-free network is a network whose degree distribution follows a power law distribution. It means that the fraction $P(k)$ of vertices having k neighbors in the network. We use the graph generator gengraphwin (http://www.cs.sunysb.edu/~algorith/implement/viger/distrib/) to generate a set of small graphs and a large graph. This dataset is denoted as SF. The numbers of vertices in small graphs are varied from 1K to 10K, and the default average degree is set to be 9. The large graph is composed of 500K vertices and 2,289,867 edges.

6.1.2. Real datasets

(a) HPRD is a human protein interaction network consisting of 9460 vertices, 34,998 edges and 307 vertex labels, in which the labels are the GO term descriptions.
(b) BioGRID is a freely accessible online interaction repository with data compiled from major model organism species by searching publications, and it is available at http://thebiogrid.org/. We download the release Build Statistics (3.1.83) December 2011, and use all the organisms. It consists of 36,078 proteins and 298,819 raw protein and genetic interactions.
(c) Yago is an RDF dataset. The RDF (Resource Description Framework) is a standard data model to describe resources on the Web and enables data exchange and richer integration. In RDF, data items are represented in the form of triples (subject, predicate, object), where vertices correspond to subjects and objects, and edges correspond to predicates [13]. Yago [26] is a RDF knowledge base which is automatically extracted from Wikipedia[1] and WordNet.[2] We keep the entities and the relationships between entities in the original Yago dataset, and randomly select one *rdf:type* from all the *rdf:types* incident to an entity as the label of this entity. There are 368,587 vertices, 527,934 edges and 45,450 vertex labels in Yago graph. The edge labels and direction are ignored in our experiments.

6.2. Experimental results

We evaluate the proposed method in this paper over the datasets as presented above, and report the offline performance, online query response time, and the scalability in terms of query graph size (the number of vertices in a graph).

6.2.1. Experiments on synthetic datasets

To prove the efficiency of SQBC, we conduct the experiments over the set of small graphs of both ER and SF compared with SING, QuickSI and SMS. In this experiment, we fix the size of query graphs, and vary the size of database graphs from 1K to 10K. Firstly, we generate a graph consisting of 10,000 vertices, and then randomly extract a connected graph consisting 9000 vertices from this 10K dataset. Iteratively, we generate xK dataset from $(x+1)$K dataset, where $1 \leqslant x \leqslant 9$.

First, we generate a set of uniform query graphs, which indicates that the query graphs for each dataset is the same set of queries. This set of queries is generated from 1K dataset in the following way. Randomly choose one of its vertices v. Starting from v, proceed in a iterative way: (I) Every time, randomly choose one vertex v_i, which is one of the selected vertices. (II) And then randomly choosing x neighbors of v_i where $1 \leqslant x \leqslant |NB(v_i)|$($|NB(v_i)|$ is the size of $NB(v_i)$). The process stops until a fixed total number vertices is reached. This yields groups of 100 queries having 20 vertices for ER and SF datasets. We perform these queries for each method and average their response time.

The consumed time by these methods is summarized in Fig. 5. The horizontal axes represent the size of the data (ER and SF) graphs, which vary from 1K to 10K Fig. 5(a) is the query response time (in milliseconds) over ER dataset. Note that, in Fig. 5(a), SING fails to return the answers since it runs out memory when dealing with 10K dataset. It shows that the query response time of SQBC over ER datasets is less than 1 ms. Undoubtedly, SQBC outperforms SING, QuickSI and SMS greatly. What is more,the performance lines of SMS and SQBC are almost horizontal demonstrating that our algorithms have a good scalability as $|V(G)|$ increases. In contrast, the query response time of SING and QuickSI both increase significantly with $|V(G)|$ increasing.

Fig. 5(b) is the query results over SF graphs. There are two vertical axes in Fig. 5(b), where the left one indicates the query response time (in milliseconds) and the right one indicates the number of matches. As expected, both the query response time and the number of matches increase as the size of dataset increases. Moreover, the performance gap between QuickSI and our methods (SMS and SQBC) also increases. Noted that there is no query response time by SING for the dataset of SF because it runs out of memory during the index construction phase.

[1] http://en.wikipedia.org/wiki/Main_Page.
[2] http://wordnet.princeton.edu/.

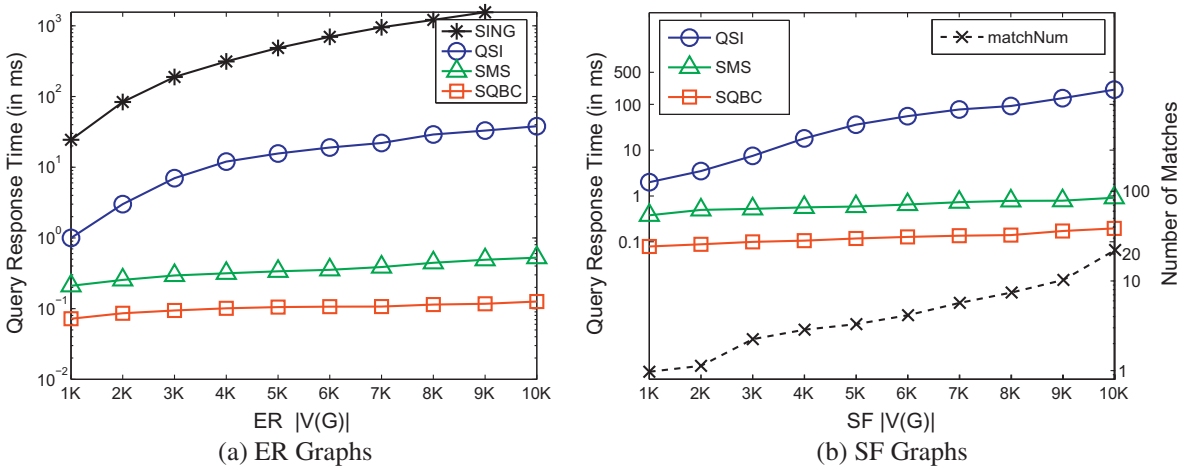


Fig. 5. Query results over ER and SF graphs.

We also perform some experiments to study the factors impacting the query performance. In this experiment, we use the same datasets as the former experiment, but adopt different query graphs. For each dataset (ER1K ∼ ER10K, SF1K ∼ SF10K), we randomly extract 100 subgraphs having 20 vertices from the corresponding dataset as its query graphs.

Fig. 6 presents the results over SF datasets, where Fig. 6(a) reports the query response time and number of matches. Generally, the larger the size of database graph is, the more time will be consumed. However, as shown in Fig. 6(a), the query response time maybe less even if the database graph is larger. For example, the query response time over SF10K is less than that of SF9K, one of the possible reasons is that the number of matches of the corresponding query over SF10K is less. But the query performance may not decrease even though the number of matches increases. As an example, the number of matches over SF9K is more than that over SF8K, but the query response time over SF9K is less than that over SF8K. To study the underlying reason, we compute the ratio of common vertices and edges for these matches, denoted as *cs*, which is defined as Eq. (4). In Eq. (4), $|V(q)|$ and $|E(q)|$ are the number of vertices and edges in the query graph Q respectively, m is the number of matches, $|V_d|$ and $|E_d|$ are the number of distinct vertices and edges in the matches respectively. Obviously, the more common structures (vertices and edges) the matches share, the larger is *cs*. Fig. 6(b) reports the ratio of common structures for the corresponding queries over each dataset. As depicted in Fig. 6(b), the matches over SF9K share many substructures which can save much time, but the matches over SF8K share less substructures (see Fig. 7).

$$cs = 1 - \frac{|V_d| + |E_d|}{(|V(Q)| + |E(Q)|) \cdot m} \tag{4}$$

Hence, there are three possible factors affecting the query response time. First, the size of data graphs. With a fixed query graph, the response time consumed over a larger data graph is biased to be more than that of a small data graph, since it is

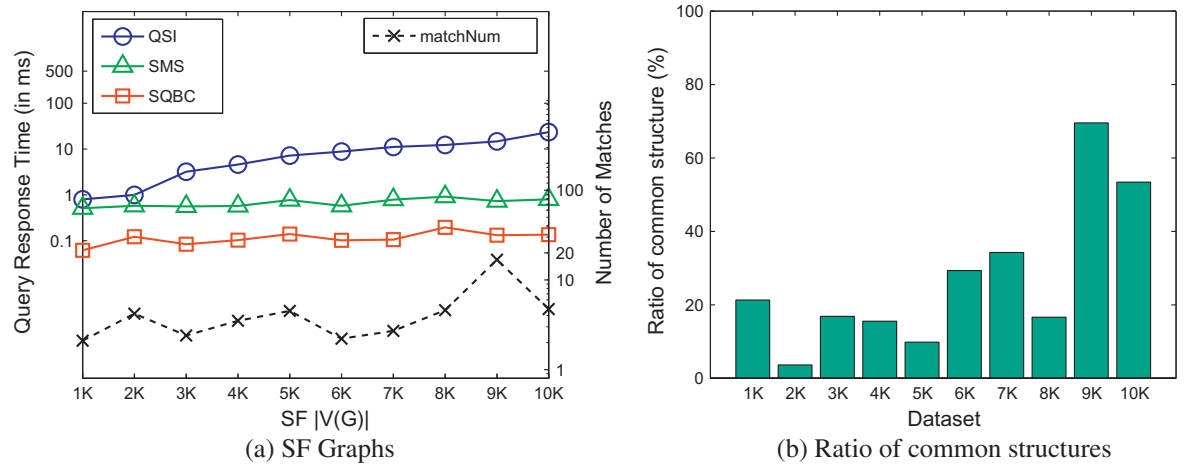


Fig. 6. Query results over ER and SF graphs.

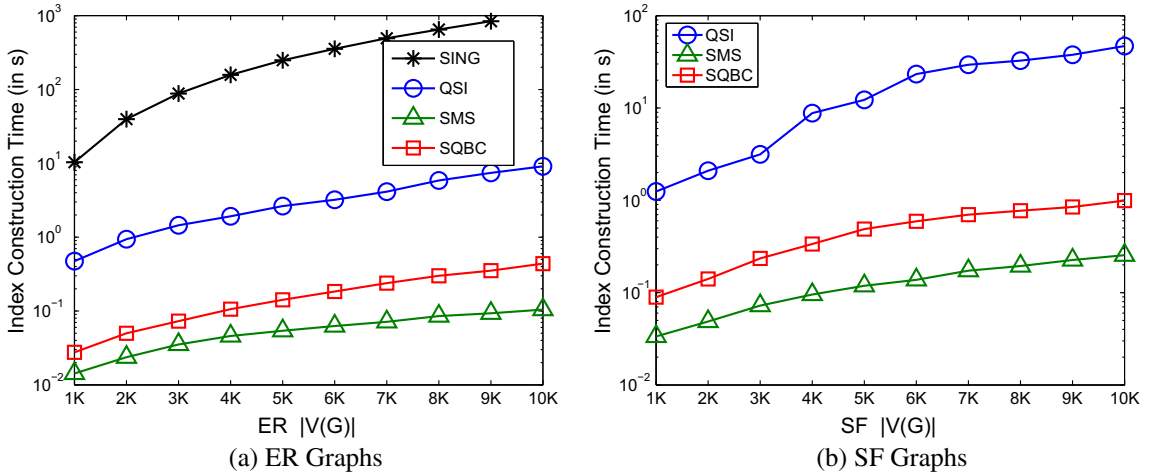


Fig. 7. Index building time (in seconds) over ER and SF graphs.

probable that the number of candidates in the large data graphs is more than that in the small one. Second, the number of matches. In general, the more matches exist, the more time will be consumed. Third, the common substructures of the matches. Straightforwardly, if there are more matches, the consumed time is more. However, this is not always the case. If many matches share the same substructures, the search space is limited, which may lead to reduce the query time for delivering all the matches.

Furthermore, the index construction time of SQBC is less than that of SING and QuickSI, but more than that of SMS, which is shown in Fig. 4. The horizontal axes represent the size of the data (ER and SF) graphs, which varies from 1K to 10K. The vertical axes represent the index building time in terms of seconds. The index construction time of SING is extremely high because it enumerates all paths starting from a vertex. QuickSI suffers from the similar problem, for it builds the index based on frequent tree features of graph database. As presented in the previous section, SQBC needs to compute all the maximal cliques, which degrades the index building performance compared to SMS.

6.2.2. Experiments on real datasets

In the experiment, we use HPRD, BioGRID and Yago as real database graphs, which are larger and more complex. The query graphs are generated in the same way as described in the previous section. However, we fix the database graphs, and vary the size of query graphs from 10 to 100. For the query graphs of each size, we generate 100 graphs, and average the response time for each set of queries. The corresponding results are shown in Fig. 8.

As all these database graphs are larger, SING fails to return the answers. There are two *y*-axes on both left and right side in Fig. 8(a) and (c). The dotted line which represents the number of matches is in the right axis. As depicted in Fig. 8(c), the query performance lines of SQBC and SMS rise and fall as the number of matches fluctuates, since these two algorithms both combine the filter and verification operations together, and their search process proceed in depth-first manner, which makes it possible that the search time is less even with more matches if they share the same substructures. Compared to SMS performance line, the fluctuation of SQBC performance line is more obvious due to the clique feature employed in SQBC. When the query graphs or the matches contain some cliques, the search space and the computational cost of verification operation can be reduced further. Besides the possible factors (the size of data graphs, the number of matches, the common substructures of the matches) that impact the query performance discussed in the former section, the size of query graph also impacts the query performance because it need much more information to be verified for query graphs of large size.

Table 2 shows the index building time of SQB, SMS and QuickSI over the datasets of HPRD, BioGRID, Yago, ER500K and SF500K. The first column are the algorithms SQBC, SMS and QuickSI, and the first row are the datasets. "–" in the last row indicates that QuickSI cannot run over the corresponding datasets in our experiments for their heavy indices.

It is obvious that the query performance of SQBC is more efficient than that of SMS and QuickSI. However, the index construction time of SQBC is less than that of QuickSI, but more than that of SMS.

To analyze the possible reasons that SQBC outperforms other algorithms, we compute the clique coverage rate (*ccr*) of HPRD, BioGRID and Yago, as shown in Table 3. The clique coverage rate is defined as the proportion of the vertices existing in at least one clique that consists of at least three vertices. The *ccr*s of HPRD and BioGRID are 44% and 37% respectively, which implies that there are many cliques in biological datasets. The index based on the mining of cliques is significant. Note that, the *ccr* of Yago is relatively low, because each entity in the dataset used in our experiments keeps only one *rdf:type*. In addtion, we compute the proportion of the vertices that exist in at least one clique to all vertices in matches, denoted as *pvcm*. The average *pvcm*s of HPRD and BioGRID are 83.3% and 65.6% respectively, which means that most matching vertices exist in at least one clique. Thus, the clique-based optimization technique can speed up query processing.

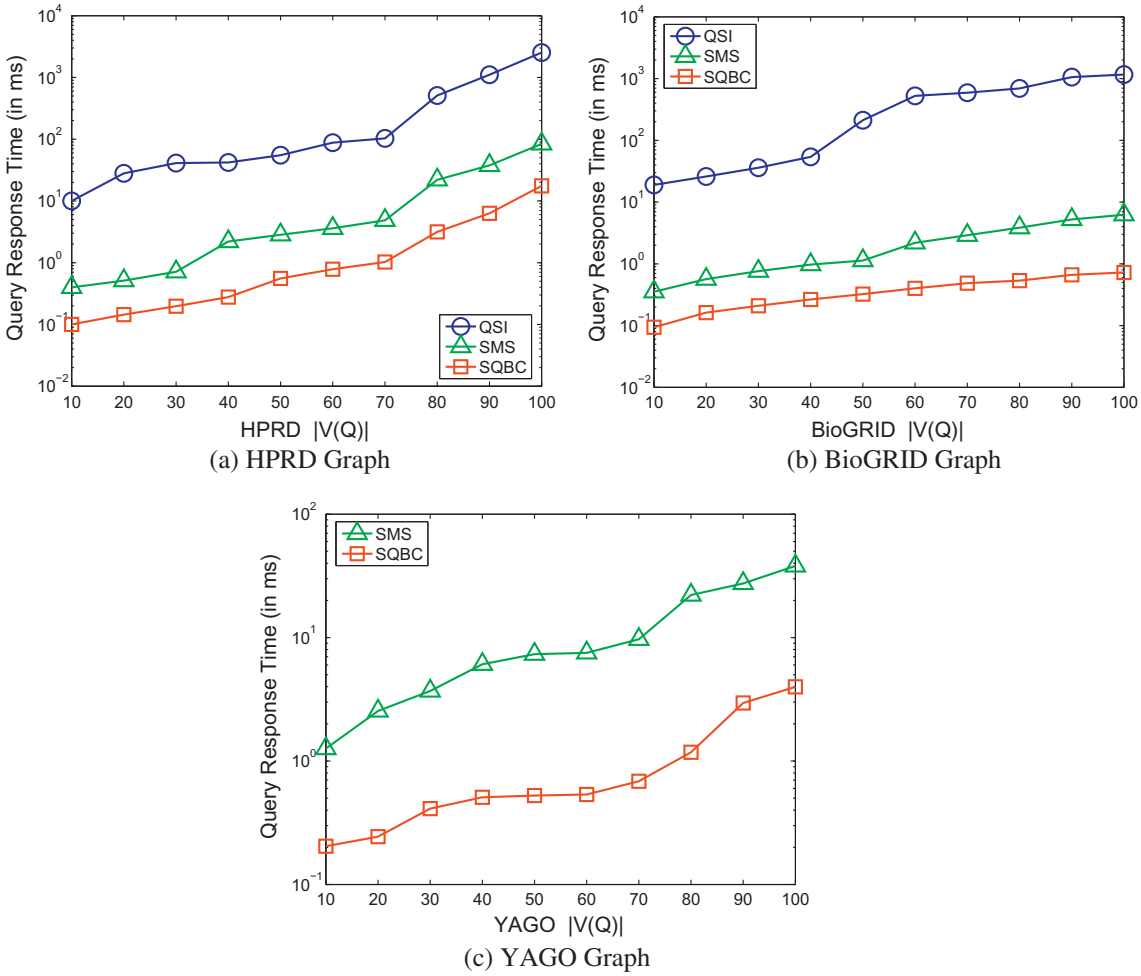


Fig. 8. Query response time (in milliseconds) over HPRD, BioGRID and YAGO graphs.

Table 2
Index building time.

Algorithm	HPRD	BioGRID	Yago	ER500K	SF500K
SQBC	0 s 628,666 μs	16 s 572,097 μs	194 s 262,701 μs	681 s 677,540 μs	803 s 232,577 μs
SMS	0 s 143,063 μs	0 s 950,985 μs	2 s 878,424 μs	15 s 294,909 μs	14 s 140,946 μs
QSI	2.3 s 56,200 μs	19 s 521,450 μs	–	–	–

6.2.3. Experiments on large datasets

To further study the efficiency and scalability of our algorithm, we fix the database to be ER500K and SF500K, and vary the size of query graph from 10 to 100. The query graphs are randomly generated from the two datasets in the same way as adopted in the former section. The algorithms SING and QuickSI cannot run over these two datasets in our experiments for their heavy indices.

Fig. 9 shows the query performance over these two datasets. There are two *y*-axes in each Fig. 9(a) and (b), where the dotted lines represent the number of matches for the corresponding queries. It indicates that the query response time shows a growing trend as the size of query graphs increases. Notice that, SQBC outperforms SMS by one order of magnitude as to some queries, and the fluctuation of SQBC performance line is more obvious than that of SMS performance line. As discussed in the former section, one main reason is that the query graphs or the matches contain some cliques, which reduces the search space and the computational cost of verification operation during the search process of SQBC. However, SMS is insensitive to the cliques in the query graphs or the matches, because it only utilize the neighborhood information of vertices. As displayed in Fig. 9, SQBC has shown high performance that the query response time is only near 1 ms even for the queries of size 100, which means it can work efficiently over both ER and SF graphs.

Table 3
The clique coverage rate (CCR) of real datasets.

Datasets	HPRD	BioGRID	Yago
CCR	44%	37%	17.4%

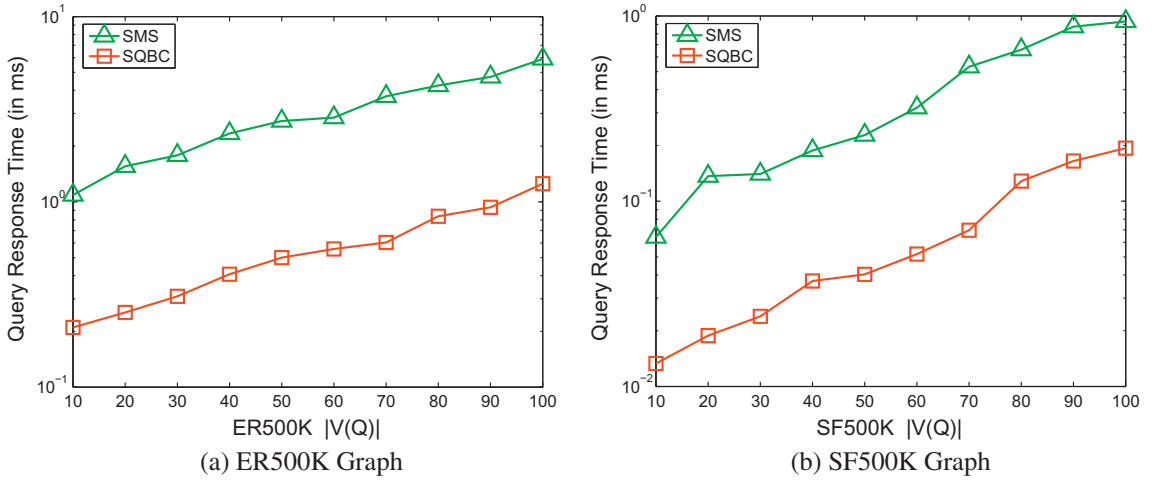


Fig. 9. Query response time (in milliseconds) over ER500K and SF500K graphs.

The index building time over these two datasets are shown in Table 1. Similar to that over other datasets, SMS is more efficient than SQBC in the index construction phase.

7. Discussion

As shown in the previous section, SING fails to answer the query in our experiments when the database graph is large and dense. Since SING considers all paths starting from a vertex producing an explosion of features in terms of the number, the space requirement of SING is costly. Moreover, the process of index construction is computationally expensive especially when it comes to deal with large and dense graphs. The heavy index also keeps the query performance of SING limited in the filtering phase.

QuickSI defines the Swift-Index which is a prefix tree. It needs to pre-compute the frequent trees as structural features, and determines the search order for the query graph q. So QuickSI does not work well enough over large and dense graphs, which has been proved in our experiments. SMS does not employ any feature structure, so its preprocessing time is least.

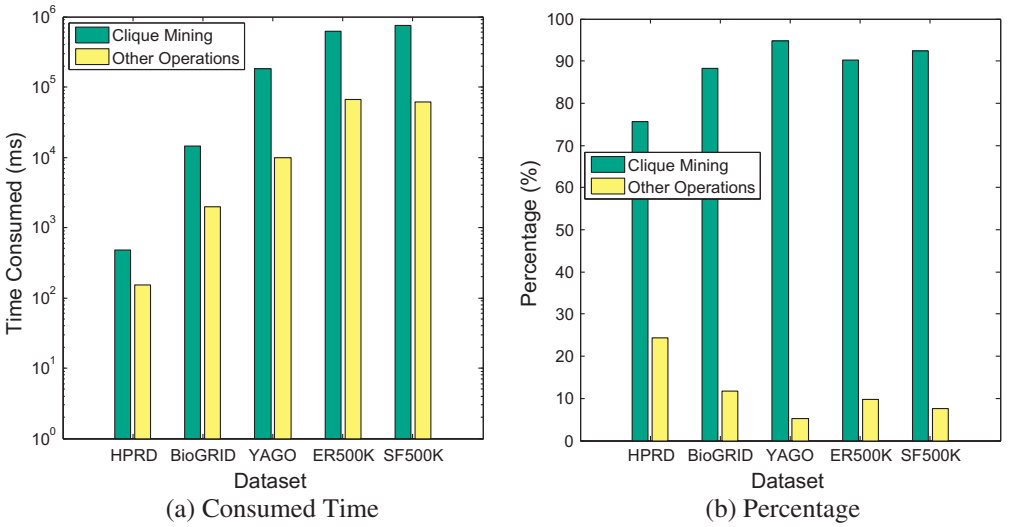


Fig. 10. Time consumed by maximal clique mining and the other operations.

SQBC outperforms SMS and SING in terms of the query response performance, due to the feature selection of maximal clique. The maximal clique is used to reduce the search space. What is more, the isomorphism test over two cliques is easily verifiable as discussed in the section of maximal clique feature. Consequently, it facilitates the filtering operation. However, the index building performance of SQBC is less efficient than that of SMS, because the process of mining all the maximal cliques during the preprocessing phase is costly in terms of time. So we divide the index building phase into two parts: maximal clique mining and all the rest operations except maximal clique mining. Fig. 10(a) shows the time consumed by the maximal clique mining and the other operations in the index building phase. Fig. 10(b) displays the percentage occupied by these two parts respectively. Obviously, The operation of maximal clique mining accounts for most of index construction time, especially when the data graphs are large and dense. So if we want to improve the preprocessing efficiency, a more efficient algorithm of mining maximal cliques needs to be designed. There are many existing literatures focus on this issue [21,11,8,19,12,30,31,29].

In the query processing, we introduce a new strategy, which combines the filtering and verification operation together so that many false positives can be avoided. During the iteration, the search space decreases fast. As expectation in Section 4.2, the experiment results show that the query performance of SQBC outperforms that of SMS with structure feature employment at the expense of increasing the preprocessing time. As discussed in the previous section, the main factors impacting the query response performance are: the size of data graphs, the size of query graphs, the amount of matches, the common substructures shared by matches, and the maximal cliques in the query and data graphs.

To summarize, SQBC is the best choice when the query graph or the data graph is dense, or the size of query graph set is large. Because we can only build the index once, based on which the set of query graphs are performed efficiently. Otherwise, SMS is more appropriate.

8. Conclusions

In this paper, we have proposed a new subgraph match algorithm over large and dense graphs called SQBC. It employs the structure of maximal cliques to reduce the search space. Clique code and vertex code are carefully designed based on the locality information. What is more, the new verification strategy is integrated into the framework. Though the index building time of *SQBC* is a little more than the time consumed by *SMS*, extensive experiments over real and synthetic datasets have shown that *SQBC* outperforms the most popular competitors including *SMS*, *QuickSI*, and *SING* greatly in terms of online query efficiency.

Besides the problem presented in this paper, the study of approximate subgraph match, subgraph query over probabilistic graphs and subgraph query over graphs updating frequently are meaningful, interesting and challenging. We will focus on these problems in the future work.

Acknowledgements

This work was supported by NSFC under Grants 61370055, 61272344 and China 863 Project under Grant No. 2012AA011101. Lei Zou's work was also supported by CCF-Tencent Open Research Fund.

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: VLDB, 1994, pp. 487–499.
[2] J.E. Beasley, N. Christofides, Theory and methodology: vehicle routing with a sparse feasibility graph, Eur. J. Oper. Res. 98 (3) (1997).
[3] J. Cheng, Y. Ke, W. Ng, A. Lu, Fg-index: towards verification-free query processing on graph databases, in: SIGMOD Conference, 2007, pp. 857–872.
[4] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, A (sub)graph isomorphism algorithm for matching large graphs, IEEE Trans. Pattern Anal. Mach. Intell. 26 (10) (2004) 1367–1372.
[5] B. Dost, T. Shlomi, N. Gupta, E. Ruppin, V. Bafna, R. Sharan, Qnet: a tool for querying protein interaction networks, J. Comput. Biol. 15 (7) (2008) 913–925.
[6] B. Gedik, K.-L. Wu, P.S. Yu, L. Liu, A load shedding framework and optimizations for m-way windowed stream joins, in: ICDE, 2007, pp. 536–545.
[7] R. Giugno, D. Shasha, Graphgrep: a fast and universal method for querying graphs, in: ICPR (2), 2002, pp. 112–115.
[8] A. Grosso, M. Locatelli, W.J. Pullan, Simple ingredients leading to very efficient heuristics for the maximum clique problem, J. Heuristics 14 (6) (2008) 587–612.
[9] H. He, A.K. Singh, Closure-tree: an index structure for graph queries, in: ICDE, 2006, p. 38.
[10] T. Horváth, J. Ramon, S. Wrobel, Frequent subgraph mining in outerplanar graphs, Data Min. Knowl. Discovery 21 (3) (2010) 472–508.
[11] K. Katayama, A. Hamamoto, H. Narihisa, An effective local search for the maximum clique problem, Inform. Process. Lett. 95 (5) (2005) 503–511.
[12] S. Khuller, B. Saha, On finding dense subgraphs, in: ICALP (1), 2009, pp. 597–608.
[13] G. Klyne, J.J. Carroll, Resource description framework (rdf): concepts and abstract syntax, in: W3C Recommendation, 2004.
[14] D. Laney, 3d data management: controlling data volume, velocity and variety, Gartner, 2001.
[15] J. Lee, W.-S. Han, R. Kasperovics, J.H. Lee, An in-depth comparison of subgraph isomorphism algorithms in graph databases, in: PVLDB, 2013.
[16] X. Lian, L. Chen, Efficient query answering in probabilistic rdf graphs, in: SIGMOD Conference, 2011, pp. 157–168.
[17] F. Mandreoli, R. Martoglia, G. Villani, W. Penzo, Flexible query answering on graph-modeled data, in: EDBT, 2009, pp. 216–227.
[18] T. Matsunaga, C. Yonemori, E. Tomita, M. Muramatsu, Clique-based data mining for related genes in a biomedical database, BMC Bioinform. (2009) 10.
[19] N. Modani, K. Dey, Large maximal cliques enumeration in sparse graphs, in: CIKM, 2008, pp. 1377–1378.
[20] R.D. Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, D. Shasha, Sing: subgraph search in non-homogeneous graphs, BMC Bioinform. 11 (2010) 96–110.
[21] P.R.J. Östergård, A fast algorithm for the maximum clique problem, Discr. Appl. Math. 120 (1–3) (2002) 197–207.
[22] R. Samudrala, J. Moult, A graph-theoretic algorithm for comparative modeling of protein structure, J. Mol. Biol. 279 (1) (1998) 287–302.
[23] H. Shang, Y. Zhang, X. Lin, J.X. Yu, Taming verification hardness: an efficient algorithm for testing subgraph isomorphism, PVLDB 1 (1) (2008) 364–375.

[24] D. Shasha, J.T.-L. Wang, R. Giugno, Algorithmics and applications of tree and graph searching, in: PODS, 2002, pp. 39–52.
[25] V. Spirin, L.A. Mirny, Protein complexes and functional modules in molecular networks, in: PNAS, 2003, pp. 12123–12128.
[26] F.M. Suchanek, G. Kasneci, G. Weikum, Yago: a large ontology from wikipedia and wordnet, J. Web Sem. 6 (3) (2008) 203–217.
[27] Y. Tian, R.C. McEachin, C. Santos, D.J. States, J.M. Patel, Saga: a subgraph matching tool for biological graphs, Bioinformatics 23 (2) (2007) 232–239.
[28] Y. Tian, J.M. Patel, Tale: a tool for approximate large graph matching, in: ICDE, 2008, pp. 963–972.
[29] E. Tomita, T. Kameda, An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments, J. Global Optim. 37 (1) (2007) 95–111.
[30] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, M. Wakatsuki, A simple and faster branch-and-bound algorithm for finding a maximum clique, in: WALCOM, 2010, pp. 191–203.
[31] E. Tomita, A. Tanaka, H. Takahashi, The worst-case time complexity for generating all maximal cliques and computational experiments, Theor. Comput. Sci. 363 (1) (2006) 28–42.
[32] J.R. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[33] D.J. Watts, P.S. Dodds, M.E.J. Newman, Identity and search in social networks, Science 29 (5571) (2002).
[34] D.W. Williams, J. Huan, W. Wang, Graph database indexing using structured graph decomposition, in: ICDE, 2007, pp. 976–985.
[35] S. Zhang, M. Hu, J. Yang, Treepi: A novel graph indexing method, in: ICDE, 2007, pp. 966–975.
[36] S. Zhang, S. Li, J. Yang, Gaddi: distance index based subgraph matching in biological networks, in: EDBT, 2009, pp. 192–203.
[37] S. Zhang, J. Yang, W. Jin, Sapper: subgraph indexing and approximate matching in large graphs, PVLDB 3 (1) (2010) 1185–1194.
[38] P. Zhao, J. Han, On graph query optimization in large networks, PVLDB 3 (1) (2010) 340–351.
[39] P. Zhao, J.X. Yu, P.S. Yu, Graph indexing: Tree + delta >= graph, in: VLDB, 2007, pp. 938–949.
[40] W. Zheng, L. Zou, D. Zhao, Answering subgraph queries over large graphs, in: WAIM, 2011, pp. 390–402.
[41] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, P.S. Yu, Mining top-k large structural patterns in a massive network, PVLDB 4 (11) (2011) 807–818.
[42] K. Zhu, Y. Zhang, X. Lin, G. Zhu, W. Wang, Nova: a novel and efficient framework for finding subgraph isomorphism mappings in large graphs, in: DASFAA (1), 2010, pp. 140–154.
[43] L. Zou, L. Chen, J.X. Yu, Y. Lu, A novel spectral coding in a large graph database, in: EDBT, 2008, pp. 181–192.