Answering Subgraph Queries over Large Graphs

Weiguo Zheng, Lei Zou *, and Dongyan Zhao

Peking University, Beijing, China
{zhengweiguo,zoulei,zdy}@icst.pku.edu.cn

Abstract. Recently, subgraph queries over large graph-structured data have attracted lots of attentions. Most of the recent algorithms proposed to solve this problem apply the structural features of graphs to construct the index, such as path, tree and subgraph. However, there is no a solid theory foundation of which structure is the best one to construct the index. What is more, the cost of mining these structures is rather expensive. In this paper, we present a high performance graph query algorithm, SMS, based on the simple yet effective neighborhood structure. To further improve the query performance, a graph partition solution is proposed and the efficient codes of vertices and blocks are carefully designed. Extensive experimental studies demonstrate the effectiveness and scalability of our algorithm in the issue of subgraph queries on large graph-structured data.

1 Introduction

In the era of Web data explosion, it is very important to organize and manage massive data efficiently. Well-established relational databases work well in the areas they are designed for, such as Enterprise Resourse Planning (ERP) and Management information system (MIS), but they have a serious shortcoming, that is "schema before data". Furthermore, it is quite expensive for RDBMS-based systems to handle updates over schemas. However, in Web data management, such as social network analysis, the schema is always changing. In this case, graph is a good model to solve this problem, because graph is powerful in expression with its structural features. Furthermore, also due to its flexibility, graph provides a good solution for Web data mashup. Thus, graph data management has attracted lots of attentions in different areas, such as bioinformatics, social networks, and semantic data management. It is a key problem to develop efficient and effective techniques to manage, process, and analyze graph data. Among these techniques, subgraph query is an interesting, fundamental, and important task.

Recently, many techniques processing subgraph query over large graph databases have been proposed [16,17,2,21]. There are two scenarios of subgraph queries, one of which is searching a query graph over a large number of small graphs, and the other one is searching a query graph over a large graph. Note that, this work focuses on the latter scenario.

Many subgraph isomorphism algorithms have been proposed, such as Ullmann [15] and VF2 algorithm [3]. However, these algorithms cannot work well in large graph databases. As we know, subgraph isomorphism is a classical NP-complete problem. In

* Corresponding author.

H. Wang et al. (Eds.): WAIM 2011, LNCS 6897, pp. 390–402, 2011.

order to improve the query performance, most of existing works adopt the "filter-and-refine" framework. Specifically, some indexes are built in offline processing. At run time, based on these indexes, an efficient filtering algorithm is used to reduce the search space. Finally, subgraph isomorphism algorithm is used to find final answers.

Most index-based solutions propose to utilize some frequent structural features, such as paths, trees or subgraphs, to construct the indexes [14,16,17,2,6,5,21]. However, this kind of methods lose their advantages in large graphs, since frequent structural pattern mining over a large graph is still an open question in data mining community. In order to address this issue, some neighborhood-based solutions have been proposed. For example, Zhao and Han propose SPath algorithm [20], which utilizes shortest paths around the vertex as basic index units. A key problem of SPath lies in the inefficiency of offline processing. Furthermore, it is quite expensive to perform updates over these indexes.

In this paper, for each vertex, we propose a simple yet effective strategy to encode the neighborhood structure around the vertex. Specifically, we consider the labels and the degrees of the neighbors. Based on the codes, we combine the filtering and verification together to speed up query processing. Furthermore, in order to address the scalability of our method, we propose a graph partition solution. We partition a large graph G into n blocks G_i, $i = 1, ..., n$. For each block G_i, we also design a code, based on which, many blocks can be filtered out safely. In order to find matches across multi-blocks, we build some carefully-designed indexes.

To summarize, in this work, we make the following contributions:

1. Based on the neighborhood of one vertex, we propose a simple yet effective code for each vertex.
2. We propose an efficient subgraph search algorithm, which combines the filtering and refining process together.
3. In order to answer a subgraph query over a very large graph, we propose a graph partition-based solution.
4. Extensive experiments confirm the superiority over existing algorithms.

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 defines the problem definition. Section 4 introduces the subgraph query algorithm SMS. The improved algorithm SMSP based on graph partition is proposed in Section 5. Section 6 reports the experimental results on real and synthetic datasets. Finally Section 7 gives the conclusion.

2 Related Work

There are mainly two embranchments of subgraph query. One of them is exact subgraph query, which means all the vertices and edges are matched exactly. The other one is approximate subgraph query, which usually concerns the structure information and allows some of the vertices or edges not be matched exactly[19,4,8,11]. However, we focus on the exact subgraph query in this paper.

In the exact subgraph query, there are two categories of related techniques: non-feature-based index and feature-based index. The algorithm Ullmann [15] to solve the problem of subgraph isomorphism is in the first branch. However, it is very expensive

when the graph is large. Some other algorithms were proposed [3,10], of which VF2 [3] is relatively efficient. The first step in VF2 algorithm is to compute the candidate set P(s) of vertices. In this step, only the edge information is considered, thus the size of P(s) is too large. In the verification phase, the feasibility rules are not efficient enough to verify the candidates either. The authors of Closure-tree [7] propose pseudo subgraph isomorphism using the strategy of checking the existence of semi-perfect matching between the query graph and database graphs. Noticed that the task of finding semi-perfect matching is very cost. So this algorithm will not work effectively for a large graph.

Recently, many index-based subgraph query algorithms have been proposed. Most of these algorithms utilize the paths, trees or subgraphs to construct the indexes [18,13,22,20]. Haichuan Shang develops an algorithm QuickSI [13] to test subgraph isomorphism. In the filtering phase, the features of prefix tree are considered to accommodate this algorithm. Shijie Zhang proposes the algorithm GADDI [18] based on the idea of frequent substructures. The authors of NOVA [22] construct the indexes based on the neighborhood label distribution of vertices. Peixiang Zhao investigates the SPath [20] algorithm, which utilizes shortest paths around the vertex as basic index units. Due to the complicated indexes, the index building cost of GADDI, Nova and SPath are all very expensive.

3 Problem Definition

In this section, we formally define our problem in this paper.

Definition 1. *A labeled graph G is defined as $G = \{V, E, \sum_V, \sum_E, F_G\}$, where V is the set of vertices, E is the set of edges, $\sum V$ is the set of vertex labels, $\sum E$ is the set of edge label, and F_G is the mapping function that maps the vertices and edges to their labels respectively.*

Definition 2. *A graph $G = \{V, E, \sum_V, \sum_E, F_G\}$ is* isomorphic *to another graph $G' = \{V', E', \sum'_V, \sum'_E, F'_G\}$, denoted by $G \approx G'$, if and only if there exists a bijection function $g : V(G) \rightarrow V'(G')$ $s.t.$*

$$1)\forall v \in V(G), F_G(v) = F'_G(g(v)); 2)\forall v_1, v_2 \in V(G), \overrightarrow{v_1v_2} \in E \Leftrightarrow \overrightarrow{g(v_1)g(v_2)} \in E'$$

Given two graphs Q and G, Q is subgraph isomorphic *to G, denoted as $Q \subseteq G$, if Q is* isomorphic *to at least one subgraph G' of G, and G' is a* match *of Q in G.*

Definition 3. (Problem Statement) *Given a large data graph G and a query graph Q, where $|V(Q)| \ll |V(G)|$, the problem that we conduct in this paper is defined as to find all* matches *of Q in G, where matches are defined in Definition 2.*

For example, in Figure 1, Q is a query graph, and G is a database graph. One match of Q in G is denoted as dotted lines in Figure 1. In this paper, we develop an algorithm to find all the subgraph matches of Q in G. For the purposes of presentation, we only discuss our method in undirected vertex-labeled graphs. Note that, it is very easy to extend our methods to directed and weighted labeled graph without a loss of generality.

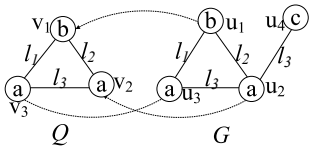


Fig. 1. An example of matching between Q and G

4 Subgraph Search Algorithm

In this section, based on the neighborhood structure, we first introduce a simple yet effective *vertex code* for each vertex. Based on the codes, we then propose a novel subgraph isomorphism algorithm.

4.1 Vertex Codes

As we know, subgraph isomorphism is a NP-complete problem. The key problem is the large search space. Let us recall Ullmann algorithm. Initially, a vertex v in query Q can match to any vertex u with the same label in graph G. Based on the neighborhood's structure around the vertex, some vertices u can be pruned, even though they have the same labels as v in Q. Although SPath also proposes to use shortest paths around one vertex as basic index units [20], it is very expensive to build the index. Furthermore, the index has to be re-built, if any update happens to graph G. Thus, in this work, we propose to use some simple yet effective structures to encode each vertex.

Definition 4. ***Vertex Code.*** *Given a vertex u in graph G, its* vertex code *is defined as $C(u) = [L(u), NLS(u) = \{[l_i, (d_{i1}, ..., d_{im})]\}]$, where $L(u)$ is the label of vertex u, l_i is a kind of label of neighbor vertices of u, $d_{i1}, ..., d_{im}$ is the degree list of u' neighbor vertices with label l_i and $d_{i1} \geq d_{i2} \geq ... \geq d_{im}$.*

Definition 5. ***PreSequence.*** *Given two sequences of numbers in decreasing order, $S = \{s_1, s_2, s_3, \ldots s_m\}$ and $T = \{t_1, t_2, t_3, \ldots t_n\}$, where s_i ($1 \leq i \leq m$) and t_j ($1 \leq j \leq n$) are both integers. S is called a* PreSequence *of T if and only if 1) $n \leq m$; and 2) $\forall s_i \in S$, $|\{t_j | t_j \geq s_i\}| \geq i$.*

Definition 6. ***Subnode.*** *Given two vertices v and u in query Q and graph G, respectively. v is a* subnode *of u if and only if 1) $L(v) = L(u)$; and 2) $\forall\ l_i \in NLS(v)$, $\exists\ l_j \in NLS(u)$ and $l_i = l_j$ and the degree list $\{d_{i1}, ..., d_{im}\}$ is PreSecquence of the degree list $\{d_{j1}, ..., d_{jn}\}$.*

If v is a subnode of u, we can also say u is supernode of v, which is denoted as $v \sqsubseteq u$.

Based on *Vertex Code*, the indexes of graph G (denoted as LVG) is constructed as follows: the id and vertex code of each vertex.

For example, in Figure 2, the index structures for G are: $LVG = \{[1, a, [a, (3); b, (3); c, (4, 2)]]; [2, c, [a, (4); c, (4)]]; [3, c, [a, (4, 3); b, (3); (c, 2)]]; [4, a, [a, (4); b, (2); c, (4)]]; [5, b, [a, (4, 3)]]; [6, a, [a, (4, 3); b, (3, 2)]]; [7, b, [a, (4, 4); c, (4)]]\}$.

Similarly, we can also encode vertices by vertex codes into LVQ. For example, given a query Q in Figure 2, $LVQ = \{[1, c, [a, (3); b, (2); c, (2)]]; [2, c, [a, (3); c, (3)]];$

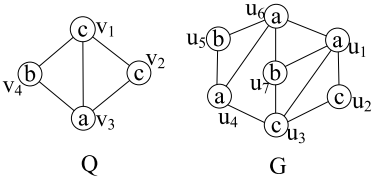


Fig. 2. A query graph Q and a database graph G

$[3, a, [b, (2); c, (3, 2)]]; [4, b, [a, (3); c, (3)]]\}$. Based on the indexes, we can easy find that u_1 is the supernode of v_3.

Lemma 1. *Given two graphs Q and G, if there is a subgraph G' (in G) that matches to Q, $\forall\, v \in V(Q)$, $\exists u \in V(G)$, $v \sqsubseteq u$.*

Proof. Based on the problem definition, it is obviously proved.

With the vertex codes, it is easy to test whether a vertex v is a Subnode of another vertex u. For example, u_1 is the supernode of v_3 but u_4 and u_6 not, so the candidate vertex of v_3 in G is only u_1.

4.2 Framework of the Algorithm

With the proper filtering approach introduced in the former subsection, the framework of our algorithm is presented as follows: first,encode the database graph G and query graph Q based on 4. Then traverse each vertex v of Q in breadth first search and filter the vertices of G to get the candidate vertex set of v. During the process of traversing vertices in Q, the operation of verification in the depth first search is conducted. That is also to say, the filter-and-refine is combined together, which will be presented in details in the following subsection.

4.3 Subgraph Query Algorithm

Definition 7. ***Match Sequence.*** *Given two graphs Q and G, match sequence between Q and G is a sequence of matched vertex pairs MS(n)=$\{(v_{i_1}, u_{j_1}), \ldots, (v_{i_n}, u_{j_n})\}$, where u_{j_k} ($1 \leq k \leq n$)is the vertex in G that matches with the vertex v_{i_k} in Q. If the size of MS is equal to the size of Q, Q is a subgraph of G.*

In the query Algorithm 1 , the vertex pairs of Match Sequence are stored in two vectors, which are denoted as SMQ and SMG respectively. The first step in query process is to select a vertex as shown in Algorithm 1. If we select a vertex whose supernode set is least in size the outer loop decreases, especially when the size of the other candidate sets is larger. Similarly, to put the neighbors of current vertex into the match vector SMQ, the vertex v whose degree is least should be chosen first, and then select the vertex whose degree is least in the rest neighbor vertices. At last, all the vertices will be selected.

Algorithm 1. Subgraph **M**atch pre**S**equency (SMS) Algorithm

Require: Input: LVG, LVQ, G and Q.
Output: Matches of Q over G
1: Select the id of 1st vertex in LVQ into SMQ, push its $neighbors'$ ids into the vector.
2: **for** each $v \in SMQ$ **do**
3: compute the candidate vertices C(v) for v.
4: **for** each $u \in C(v)$ **do**
5: quickly verify the current match.
6: **if** u matches v **then**
7: push u into SMG.
8: **for** each v' $\in$ neighbors of v **do**
9: **if** v' does not exist in SMQ **then**
10: put v' into SMQ.
11: SMS(LVG, LVQ, G,Q).

Definition 8. ***Prior Vertex.*** *In the matching sequence, the prior vertex of v is the earliest selected vertex in the neighbor vertices of v.*

In the matching sequence, each vertex except the first one has a prior vertex. In the matching process, to get all the candidate vertices of v, first we find all the supernodes, which are denoted as set S_1. Then find the vertex matched with the prior vertex of v, and get its neighbor vertices denoted as set S_2. Last the candidate vertices of v is the intersection of S_1 and S_2.

Noticed that no matter how effective the filtering ability is, it is unavoidable to conduct the verification operation. So designing an effective and efficient verification approach is rather important and necessary.

If Q is subgraph isomorphic to G, for 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]. That is also to say, If Q is subgraph isomorphic to G, there must be a match sequence at least. For each subset vertices of the match sequence, the vertex in Q must be subnode of the corresponding vertex in G. This has been proved in Lemma1. However, there is another important theorem as presented in Lemma 2.

Definition 9. ***Sequence Number Set.*** *Given two graphs Q and G, as to the match sequence MS(n), Sequence Number Set of the next vertex v to be matched is a set of number, denoted as SNS(v) , and its size is equal to the degree of v. For each neighbor v' of the vertex v, if v' is in the MS(n), the sequence number of v' in the MS(n) is pushed into SNS(v), or the value (n+j+1) is pushed into SNS(v), where j is the number of neighbors of v that are not in MS(n) currently.*

Lemma 2. *In the process of finding the match sequences between Q and G, the pair of new vertices to be matched are v and v' in Q and G respectively. If SNS(v) $\nsubseteq$ SNS(v'), v' does not match v in the sequence, or they will match.*

Proof. If current match sequence MS(k)={ $(v_t, u_s), (v_{t+1}, u_{s+1}) \ldots (v_{t+k}, u_{s+k})$}. When trying to find next match pair, the next candidate vertex in Q is v_{k+1} , and the next candidate vertex in G is u_{k+1}. If SNS(v_{k+1}) $\nsubseteq$ SNS(u_{k+1}), there must be

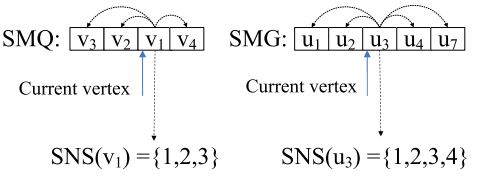


Fig. 3. SNS of current vertices in Q and G

some numbers which are not in SNS(u_{k+1}). Supposing the corresponding vertices are $\{v_{m1}, v_{m2} \ldots v_{mp}\}$, then the edges $(v_{m1}, v_{k+1}), (v_{m2}, v_{k+1}) \ldots (v_{mp}, v_{k+1})$ cannot be matched in G, so v_{k+1} is not matched with u_{k+1}.

Lemma 3. *For each pair of vertices in each subsequence of a sequence S of vertices between Q and G, if the vertex in Q is a subnode of the vertex in G, the sequence S is a match sequence of Q and G.*

Proof. If each subsequence of S could succeed, we can assign a sequence from 1 to the length of S, and this sequence assigned must succeed. Then Lemma3 is deduced. Since Lemma2 could be proved, so does Lemma 3.

For example, in Figure 3 supposing that two vertices have been matched currently, MS(2)=$\{(v_3, u_1), (v_2, u_2)\}$, if the next vertex to be matched in Q is v_1, a candidate vertex of v_1 in G is u_3. As is shown in Figure 3, SNS(v_1)=$\{1,2,3\}$, SNS(u_3)=$\{1,2,3,4\}$, SNS(v_1) is a subset of SNS(u_3), so v_1 can match u_3 temporarily.

If all the vertices in Q are included in the MS, the match sequence is a solution. In the match process, if the degree of a vertex is larger, the more edges will be test at a time. That is also to say graph-at-a-time is more cost-effective than traditional edge-at-a-time query processing.So this verification is rather efficient and effective for the good properties.

5 Subgraph Query Based on Partition

Considering the problem of subgraph isomorphism is NP-Complete, isomorphism test will be costly and inefficient in time and space cost if the size of database graph is very large. However, if the size of the graph is small or the graph is sparse, isomorphism test will be much easier.

Intuitively, we can partition the database graph into some small subgraphs. However, there are two major challenges: what the number of the partitioned subgraphs is better and dealing with matches crossing in multi-blocks.

5.1 Offline Processing

First, we partition the database graph into N blocks with METIS[9]. Noticed that if the size of each block is too small, the number of subgraphs and crossing edges are both increasing. Based on our experiments, when the size of the subgraph is 10 to 100 times

as large as the query graph, the query performance is better. In order to improve the query performance, we propose the block codes for each block to filter out some blocks that do not contain any subgraph that matches the query graph Q.

Definition 10. ***Block Codes.*** *$GivenablockBofG, itsblockcodeisdefinedasLLB = \{l_1, m_1, (d_1, d_2, \ldots d_{m_1}); \ldots; l_n, m_n, (d_1, d_2, \ldots d_{m_n})\}$, where l is the label, m is the number of vertices with label l and d_i is the degree of ith node which has the label l.*

The same code LLQ is also used for Q.

Lemma 4. *Given two graphs Q and G, if Q is a subgraph of G if and only if $\forall l \in LLQ, \exists l \in LLG$ and the corresponding degree list LLQ must be the PreSequence of that in LLG, where PreSequence if defined in Definition 5.*

Proof. It is easy to be proved according to the definition of the subgraph isomorphism.

In figure 4, the database graph is partitioned into three blocks (P_1, P_2, P_3).$LLQ = \{a, 1, (2); b, 1, (2); c, 2, (3, 3)\}, LLP_1 = \{a, 1, (3); b, 1, (3); c, 2, (4, 3); d, 1, (3)\}$, $LLP_2 = \{a, 2, (3, 2); b, 1, (3); c, 1, (4); e, 1, (4)\}, LLP_3 = \{a, 1, (4); b, 1, (2); c, 2, (3, 2); e, 1, (3)\}$. To find the matches without this strategy, we have to try to traverse every partitioned subgraph. For example, when traversing P_1, one candidate match sequence is "u_4-u_3-u_2", in that case the algorithm does not return until after trying to match the last vertex with label "c" in Q. However, if employing Lemma 4, we can filter P_2 and P_3 safely. So before dropping into recursion deeply, we can conclude that Q cannot match any subgraph in P_2 and P_3. Undoubtedly, this will make our algorithm more efficient and effective.

There is another big challenge that dealing with the possible matches crossing several blocks. It is easy to get the idea that extending every block over x hops so that every two blocks have a overlap. However, the real database graph is dense, so the extended blocks will be as large as the database graph due to the "Six Degrees of Separation"[12].

To reduce the crossing edges, the algorithm of "Min-Cut" is employed as mentioned before. Noticed that after partition,if we only consider the crossing edges, the graph will be sparse as shown in Figure 7. Thus each partition edge with two labels can be viewed as a special "vertex". The inverse index "LVE" similar to the previous LVG(or LVQ) can be constructed.

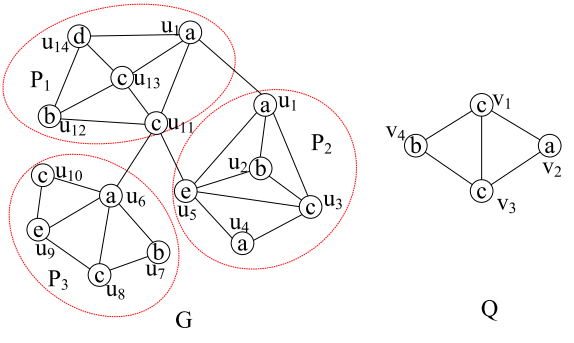


Fig. 4. The partitioned graph G and the query graph Q

Definition 11. *Given a crossing edge "u_1u_2", its crossing edge code is defined as LVE=[(l_1,l_2){u_1,u_2,NLS(u_1),NLS(u_2) }],where l_1 and l_2 are the corresponding labels of u_1 and u_2 ,NLS(u_1)and NLS(u_2)are the neighbor degree lists of u_1 and u_2 respectively,which are the same as that in definition 4.*

For example, the "LVE" of crossing edge "$u_6u_1$1" is $[(a, c)\{u_6, u_1, [b(2), c(2, 2), e(3)]$, $[a(4), b(4), d(3), e(5)]\}]$. It is obvious that the prune ability of LVE will be much more evident, because it stores more information. What is more, if a database graph contain 100,000 vertices,500,000 edges and 200 different labels, the size of candidate set of each kind vertex label is 500 on average. But if we combine two vertices as a special "vertex" and the the number of crossing edges is 300,000, the size of candidate set of each kind "vertex"label is 7.5 on average. Undoubtedly, the search space be reduced greatly.

5.2 Online Query Based On Partition

With the indexes constructed in former subsection,we conduct the match process in this subsection.The framework of improved algorithm with partition is shown in Algorithm 2.

First, we find the matches in each block, the process of which is the same as that in SMS. What need to be noted is that the size of the "G" is much smaller relatively. So the time consumed in this phase is far less than that in the whole database graph.

Algorithm 2. Subgraph **M**atch pre**S**equency Based on **P**artition (SMSP) Algorithm

```
1: For each block calls Algorithm 1.
2: for each edge e ∈ Q do
3:     find the crossing edges that match e in G.
4:     put the two vertices of e into SMQ.
5:     call Algorithm 1.
```

Second, we find the possible matches through the crossing edges. In order to avoid traversing a possible match many times, a lexicographic order should be assigned. The match process is similar to SMS. The main difference lies in the match sequence of Q. Here, the match spreads in two directions in the first step. For each edge e in Q,we select a edge e' from crossing edges which matches e and push their vertices into SMQ and SMG respectively. And then the neighbors of vertices of e are pushed into SMQ. The rest process is the same as SMS.

6 Experimental Evaluation

In this section, we evaluate the query performance of our algorithms SMS and SMSP over both synthetic and real data sets, which prove that our approaches outperform some state-of-the-art algorithms well by more than one order of magnitude.

6.1 Experiment Preparation

The codes of GADDI[18] and Nova[22] are provided by authors. Furthermore, we implemented another algorithm SPath according to [20]. In our algorithm, we use the software metis to partition the database graph. All these algorithms compared were implemented using standard C++. The experiments are conducted on a P4 2.0GHz machine with 2Gbytes RAM running Linux.

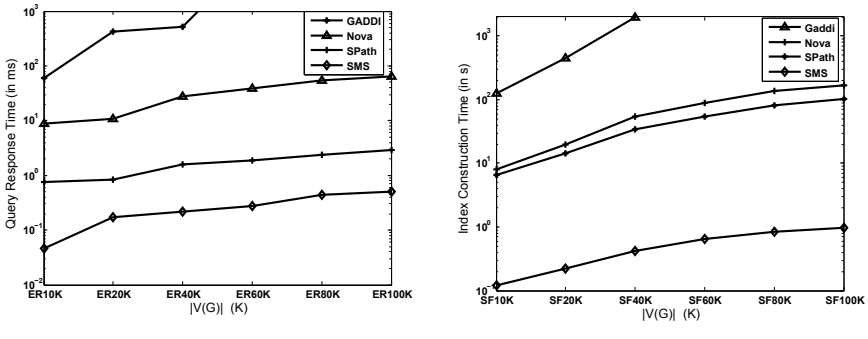
Synthetic Data Sets. We construct two data sets using two classical random graph models, Erdos Renyi model and Scale-Free model. The two data sets are denoted as ER and SF respectively. The number of vertices in ER and SF both vary from 10K to 100K. And the number of vertex labels of all the database graphs is 250.

Real Data Sets. We use a human protein interaction network(HPRD) and a RDF data(Yago) as the real data sets. HPRD consists of 9, 460 vertices,37, 000 edges and 307 generated vertex labels with the GO term description.Actually, Yago is a RDF graph consists of 368, 587 vertices, 543, 815 edges and 45, 450 vertex labels, where vertices, edges and labels corresponds to subjects(or objects), properties and the classes of subjects(or objects) respectively. The edge labels and direction are ignored in our experiments.

6.2 Experimental Results

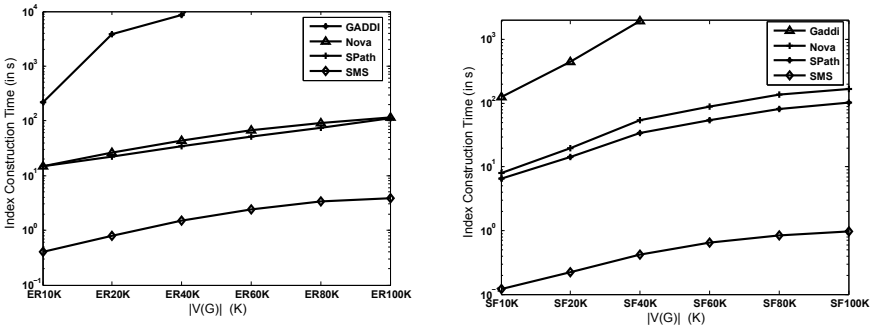
In this sub-section, we compare our algorithm SMS with GADDI [18], Nova [22] and SPath [20]. And then study the query performance of SMSP.

Experiment 1. Performance of SMS versus $|V(G)|$. Figure 5(a) and Figure 5(b) summarize the time consumed by the four methods,where the corresponding query graph is 10-vertex graph and size of the data graph of ER and SF varies from 10k to 100k. Noted that the query respond time of SMS is less than 1 second even though the size of G is 100k. What is more,the index construction time of our algorithm is also less than the other three algorithms as shown in Figure 6(a) and Figure 6(b). Undoubtedly,it proves our algorithm has a good scalability as $|V(G)|$ increasing.



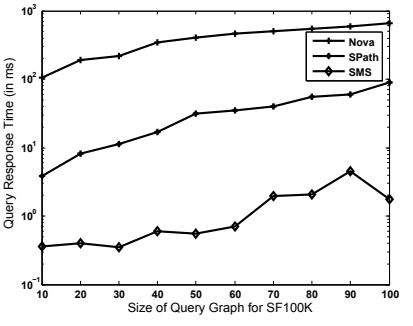
(a) Query Response Time (in milliseconds) over ER Graphs (b) Query Response Time (in milliseconds) over SF Graphs

Fig. 5. Performance VS. $|V(G)|$ in Query Response

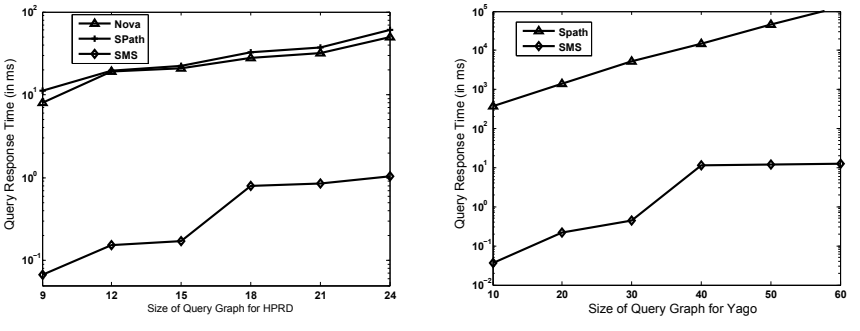


(a) Index Building Time (in seconds) over ER Graphs (b) Index Building Time (in seconds) over SF Graphs

Fig. 6. Performance VS. $|V(G)|$ in Index Building



(a) Query Response Time (in milliseconds) Over SF100K



(b) Query Response Time (in milliseconds) Over HPRD (c) Query Response Time (in milliseconds)Over Yago

Fig. 7. Performance VS. $|V(Q)|$ Over SF100K, HPRD, Yago

Experiment 2. Performance of SMS versus $|V(Q)|$. To further study the scalability and efficiency of our methods,we fix the database graph to be SF100k, HPRD, Yago respectively. And then, vary the the size of query graph. The corresponding results are shown in Figure 7(a),Figure 7(b) and Figure 7(c).

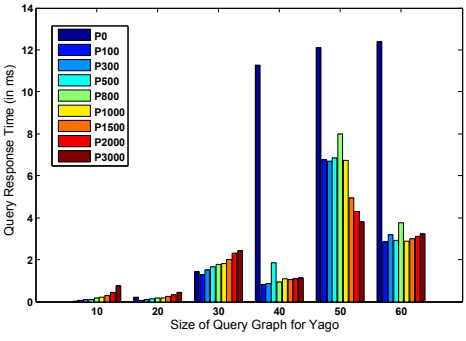


Fig. 8. Performance SMSP VS. SMS

In HPRD data set, SMS takes 0.25 seconds to construct the index.To finish the index construction Nova and GADDI need about 20 seconds and 600 seconds respectively. In Yago data set, SMS takes about 4.22 seconds to finish the index process, which is also far faster than SPath.

Experiment 3. Performance of SMSP versus partition number. In this experiment, we use the data set of Yago and vary the number of partition blockes from 100 to 3000. The query response time is shown in Figure 8, where Pn means partition the database graph into n parts. Specially,P0 means no partition,namely SMS.

Noted that, when the query graph is small(such as the size of Q is 10,20 or 30), the SMS is faster than SMSP no matter how many parts the database graph partitioned into. While when the size of the query graph is larger relatively, the superiority of SMSP appears, but it is very difficult to determine how many blocks the database graph partitioned into is best.

To sum up, it is obvious that the algorithm shows high efficiency and good scalability in all these data sets. Generally speaking, our algorithm is orders of magnitude faster than the compared existing three algorithms. When the query graph is larger, SMSP outperforms SMS well, but the optimal number of blocks needs to be studied in further step.

7 Conclusions

In this paper, we consider the query graph problem on large graph-structured data. Most the of the recent algorithms construct the index based on the structure information of the graph, which are expensive due to the mining of the structure feature. We carefully design vertex code based on the information of each vertex and its neighbors. What is more, we propose the strategy of partitioning the large graph to improve the query performance. No matter what the index is constructed based on and how effective the pruning method is, the efficient verification strategy is important and critical. The algorithm presented in this paper is very effective and efficient both in time and memory for its simple index.

Acknowledgments. This work was supported by NSFC under Grant No.61003009 and RFDP under Grant No. 20100001120029.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: VLDB, pp. 487–499 (1994)
2. Cheng, J., Ke, Y., Ng, W., Lu, A.: fg-index: Towards verification-free query processing on graph databases. In: SIGMOD (2007)
3. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. 26(10), 1367–1372 (2004)
4. Dost, B., Shlomi, T., Gupta, N., Ruppin, E., Bafna, V., Sharan, R.: Qnet: A tool for querying protein interaction networks. In: Speed, T., Huang, H. (eds.) RECOMB 2007. LNCS (LNBI), vol. 4453, pp. 1–15. Springer, Heidelberg (2007)
5. Williams, J.H.D.W., Wang, W.: Graph database indexing using structured graph decomposition. In: ICDE (2007)
6. Jiang, P.Y.H., Wang, H., Zhou, S.: Gstring: A novel approach for efficient search in graph databases. In: ICDE (2007)
7. He, H., Singh, A.K.: Closure-tree: An index structure for graph queries. In: ICDE (2006)
8. Jiang, H., Wang, H., Yu, P.S., Zhou, S.: Gstring: A novel approach for efficient search in graph databases. In: ICDE (2007)
9. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: SC (1995)
10. Liu, J., Lee, Y.T.: A graph-based method for face identification from a single 2d line drawing. IEEE Trans. Pattern Anal. Mach. Intell. 23(10) (2001)
11. Mandreoli, F., Martoglia, R., Villani, G., Penzo, W.: Flexible query answering on graph-modeled data. In: EDBT, pp. 216–227 (2009)
12. Milgram, S.: The small-world problem. In: PT, vol. 1, pp. 61–67 (1967)
13. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. PVLDB 1(1) (2008)
14. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS (2002)
15. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM 23(1) (1976)
16. Yan, X., Yu, P.S., Han, J.: Graph indexing: A frequent structure-based approach. In: SIGMOD (2004)
17. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. In: ICDE (2007)
18. Zhang, S., Li, S., Yang, J.: Gaddi: distance index based subgraph matching in biological networks. In: EDBT, pp. 192–203 (2009)
19. Zhang, S., Yang, J., Jin, W.: Sapper: Subgraph indexing and approximate matching in large graphs. PVLDB 3(1), 1185–1194 (2010)
20. Zhao, P., Han, J.: On graph query optimization in large networks. In: VLDB (2010)
21. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: Tree + delta $>=$ graph. In: VLDB (2007)
22. Zhu, K., Zhang, Y., Lin, X., Zhu, G., Wang, W.: Nova: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5981, pp. 140–154. Springer, Heidelberg (2010)