An Edge-Based Framework for Fast Subgraph Matching in a Large Graph

Conference Paper · April 2011

DOI: 10.1007/978-3-642-20149-3_30 · Source: DBLP

Yoon Joon Lee
Korea Advanced Institute of Science and Technology
101 PUBLICATIONS **1,570** CITATIONS

An Edge-Based Framework for Fast Subgraph Matching in a Large Graph

Sangjae Kim, Inchul Song, and Yoon Joon Lee

Department of Computer Science, KAIST, Republic of Korea
{sjkim,icsong}@dbserver.kaist.ac.kr, yoonjoon.lee@kaist.ac.kr

Abstract. In subgraph matching, we want to find all subgraphs of a database graph that are isomorphic to a query graph. Subgraph matching requires subgraph isomorphism testing, which is NP-complete. Recently, some techniques specifically designed for subgraph matching in a large graph have been proposed. They are based on a filtering-and-verification framework. In the filtering phase, they filter out vertices that are not qualified for subgraph isomorphism testing. In the verification phase, subgraph isomorphism testing is performed and all matched subgraphs are returned to the user. We call them a vertex-based framework in the sense that they use vertex information when filtering out unqualified vertices. Edge information, however, can also be used for efficient subgraph matching. In this paper, we propose an edge-based framework for fast subgraph matching in a large graph. By using edge connectivity information, our framework not only filters out more vertices in the filtering phase, but also avoids unnecessary edge connectivity checking operations in the verification phase. The experimental results show that our method significantly outperforms existing approaches for subgraph matching in a large graph.

1 Introduction

Since graphs are useful to represent structured, complex data, they have been used in many application areas such as Web, social networks, communication networks, bioinformatics, ontology engineering, software modeling, VLSI reverse engineering, etc. Subgraph matching, which is to find all subgraphs of a database graph that are isomorphic to a query graph, is one of the most frequently used operations in graph databases. For example, a financial crime investigator may want to find all occurrences of matches to a spurious pattern in a financial network where vertices represent account holders or banks and edges represent money transfer transactions [1]. A biologist may want to find all occurrences of matches to a specific biological pattern in protein-protein interaction networks [2] or gene regulatory networks. In addition, subgraph matching can be used for detecting and preventing some privacy attacks on anonymized social network data [3], [4].

There are two types of queries related to subgraph matching. A *subgraph containment query* finds all graphs that contain a subgraph which is isomorphic to the query graph. Much research has been done for this type of query [5],[6],[7],[8],[9]. In these works, they assume that a graph database contains many small graphs. A *subgraph matching query* finds, from a single database graph, all subgraphs that are isomorphic to the query graph. Many general purpose algorithms have been proposed for subgraph matching queries [10],[11]. Recently two techniques for subgraph matching queries in a large graph have been proposed in [12],[13]. In this paper, we focus on subgraph matching queries for a large graph.

Since subgraph matching requires subgraph isomorphism testing, which is NP-complete [14], existing methods typically use a *filtering-and-verification* framework. In the filtering phase, the vertices in the database graph are filtered out if they are not qualified as a matching vertex. This is accomplished by comparing *signatures* of vertices, which contain information about the vertices themselves and neighborhood information. After filtering, only the remaining vertices, which we call *candidate vertices* in this paper, are input to subgraph isomorphism testing. In the verification phase, subgraph isomorphism testing is performed and all subgraphs of the database graph that are isomorphic to the query graph are found and returned to the user. The main task of the verification phase is to check whether candidate vertices of different query graph vertices are properly connected to each other. Many kinds of heuristics can be employed to speed up the verification process.

Existing approaches mainly focus on reducing the input size to subgraph isomorphism testing. GADDI [12] proposed a technique for subgraph matching in a large graph based on data mining techniques. It uses *discriminative substructures*, which are small substructures in induced intersection graph between the neighborhoods of two vertices, as vertex signatures. NOVA [13] is another technique for subgraph matching in a large graph. It uses label distribution information around vertices as vertex signatures. Both of these methods are a *vertex-based framework* in the sense that they use only vertex information to filter out unqualified vertices. Edge information, however, also can be used in the filtering process. For example, by selectively checking connectivity between vertices in the database graph, we can further filter out those candidate vertices that do not have required connections to other vertices.

In this paper, we propose an edge-based framework for fast subgraph matching in a large graph. Our method follows a filtering-and-verification framework. Unlike existing vertex-based frameworks, our method uses edge connectivity information in both of the filtering and verification phases for fast subgraph matching. In the filtering phase, it filters out more candidate vertices based on edge connectivity information. Edge connectivity information is also used in the verification phase to reduce extensive connectivity checking operations between vertices. Some of the connectivity checking operations can be removed altogether. The experimental results show that our method significantly outperforms existing approaches for subgraph matching in a large graph.

The rest of this paper is organized as follows. Section 2 gives the background information and an overview of our method. Section 3 describes an index structure used in our method. The filtering and verification phases are explained in Section 4 and 5, respectively. We evaluate our method in Section 6. Section 7 discusses related work and Section 8 concludes the paper.

2 Preliminaries

In this section, we introduce the basic definitions used in the paper and give the formal problem statement. Our proposed method supports both directed and undirected graphs with labeled vertices and/or labeled edges. For ease of presentation, we assume a simple graph with labeled vertices. It is straightforward to apply our method to other types of graphs. We also assume that every query graph and database graph considered in this paper are connected, i.e., every pair of vertices is connected by a path.

Definition 1. Vertex-labeled graph. A vertex labeled graph is denoted as G=(V, E, L, *l*), where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, L is the set of vertex labels, and *l* is a mapping function: $V \rightarrow L$.

Definition 2. Subgraph isomorphism. Given two graphs G = (V, E, L, *l*) and G' = (V', E', L', *l*'), G is *subgraph isomorphic* to G', if there exists an injective function *f*: $V \rightarrow V'$ such that

1. $\forall v \in V, l(v) = l'(f(v))$,
2. $\forall (u, v) \in E \Rightarrow (f(u), f(v)) \in E'$.

Such an injective function is called a *subgraph isomorphism mapping*.

Problem Statement. Given a query graph q and a database graph G, find all subgraph isomorphism mappings from q to G.

2.1 Filtering and Verification Framework

Our method uses the *filtering-and-verification* framework. In subgraph isomorphism testing, for each vertex v_q in the query graph q, we need to try every vertex v_G in the database graph G as a matching vertex of v_q. In the filtering phase, we filter out those vertices v_G that cannot be a matching vertex of v_q. To this end, we encode each vertex and produce its *signature*. The signature of a vertex contains information about the vertex itself and neighbor information. Unqualified vertices in the database graph are filtered out by comparing their signatures with those of vertices in the query graph. Unlike existing methods where only vertex signatures are used for filtering, our method uses edge signatures as well to filter out more vertices. The remaining vertices from the filtering phase, which we call *candidate vertices*, are used as input to subgraph isomorphism testing. In the verification phase, subgraph isomorphism testing is performed and all possible subgraph isomorphism mappings are produced and returned to the user.

2.2 Representing Vertices and Edges

Various information can be used as vertex signatures and edge signatures. For example, NOVA [13] uses a vertex label, degree, and neighbor information as vertex signatures. In order for a signature to be used for filtering out unqualified vertices, it must satisfy the *inequality property* [12], [13]. More specifically, let *sig*(v) and *sig*(u) be the signature of vertex v in the query graph and that of vertex u in the database graph, respectively. Then for vertex u to be a matching vertex of v, *sig*(v) must be less than or equal to *sig*(u), i.e., $sig(v) \leq sig(u)$. The operator $\leq$ must be properly defined for a specific signature to enforce the inequality property. Our method is designed to work with any vertex signature that satisfies the inequality property.

2.3 Vertex and Edge Signatures

In this paper, we consider two kinds of vertex signatures, namely NOVA [13] and NPV [9]. They both use a vertex label and degree, and neighbor information in their signatures. The difference between them lies in the neighbor information used. NOVA uses label distribution information around a vertex up to a user-specified distance as neighbor information. In NPV, simple paths from a vertex up to a pre-defined length are used as neighbor information. Signature comparison between two vertices $sig(\mathrm{v}) \leq sig(\mathrm{u})$ is performed by checking the following conditions:

$$l(\mathrm{v}) = l(\mathrm{u}) \tag{1}$$
$$deg(\mathrm{v}) \leq deg(\mathrm{u}) \tag{2}$$
$$nInfo(\mathrm{v}) \leq nInfo(\mathrm{u}), \tag{3}$$

where l(v) and l(u) are the labels of v and u, deg(v) and deg(u) are the degrees of v and u, and $nInfo$(v) and $nInfo$(u) are the neighbor information of v and u, respectively. How to check Condition (3) is specific to each kind of signature, and information such as the number of distinct vertex labels around a vertex is commonly used in condition checking. For more details, refer to [12] or [13].

Similarly, we also define the edge signature. The edge signature for an edge contains the labels of its endpoint vertices, the sum of their degrees, and their signatures. Given an edge $\mathrm{e_q}=(\mathrm{v_l},\mathrm{v_r})$ in the query graph q and $\mathrm{e_G}=(\mathrm{u_l},\mathrm{u_r})$ in the database graph G, we use the following conditions to check if $sig(\mathrm{e_q}) \leq sig(\mathrm{e_G})$:

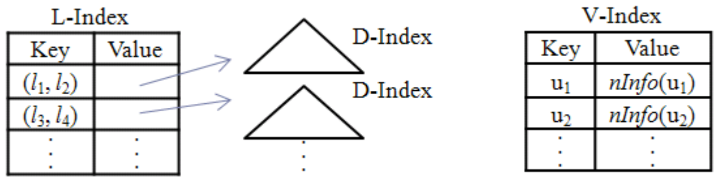
$$l(\mathrm{v_l}) = l(\mathrm{u_l}) \wedge l(\mathrm{v_r}) = l(\mathrm{u_r}) \tag{4}$$
$$deg(\mathrm{v_l}) + \mathrm{deg}(\mathrm{v_r}) \leq deg(\mathrm{u_l}) + \mathrm{deg}(\mathrm{u_r}) \tag{5}$$
$$nInfo(\mathrm{v_l}) \leq nInfo(\mathrm{u_l}) \wedge nInfo(\mathrm{v_r}) \leq nInfo(\mathrm{u_r}) \tag{6}$$

3 Pre-processing

In this section, we describe the *Edge Index* (E-Index) and *Vertex Index* (V-Index) that are used to speed up the filtering phase. The purpose of E-Index is to find candidate edges of each query graph edge from the database graph. Given an edge ($\mathrm{v_1}$, $\mathrm{v_2}$) in the query graph q, an edge ($\mathrm{u_1}$, $\mathrm{u_2}$) in the database graph G is its candidate edge if 1) the labels of corresponding vertices are the same, i.e., Condition (4), 2) the degree sum of two vertices in edge ($\mathrm{v_1}$, $\mathrm{v_2}$) is less than or equal to the degree sum of two vertices in edge ($\mathrm{u_1}$, $\mathrm{u_2}$), i.e., Condition (5), and 3) the neighbor information of the corresponding vertices satisfies the inequality property, i.e., Condition (6). Here we need three comparisons. The E-Index is used to speed up the first two comparisons, i.e. the label comparison and degree sum comparison.

We may construct a separate index to speed up each of these two operations. For example, we construct an index whose key is a pair of vertex labels and value is the list of edges that have those vertex labels. We also construct another index whose key is a degree sum and value is the list of edges having that degree sum. We can find candidate edges by first retrieving candidate edges from each of the two indexes and then intersecting them.

Instead of having two separate indexes, the E-Index combines them and builds a two-level index to further reduce index search time. The first level index is called *Label Index* (L-Index). Its key is a pair of vertex labels and value is a pointer to a second level index. The second level index is called *Degree Index* (D-Index). Each D-Index is constructed separately for the edges having the identical vertex label pair. Given a D-Index, its key is a degree sum and value is a list of edges having that degree sum. Figure 1(a) shows the two-level structure of E-Index. Given an edge (v_1, v_2) in the query graph, we can find candidate edges as follows. First we obtain a pointer to a D-Index by querying L-Index with key ($l(v_1)$, $l(v_2)$). Then we find the candidate edges by issuing a range degree sum query over the D-Index. Both L-Index and D-Index can be easily implemented by using B+-tree index structure.



(a) The two level structure of E-Index (b) V-Index

Fig. 1. A simple example of E-Index and V-Index

Note that we still need a signature comparison between the edge in the query graph and the candidate edges to find the final candidate edges. For this last comparison, we need to retrieve neighbor information of the vertices in the candidate edges. To efficiently retrieve neighbor information of vertices, we construct a *Vertex Index* (V-Index), whose key is a vertex identifier and value is its neighbor information. Figure 1(b) shows the structure of V-Index.

4 Filtering

In this section, we describe how to find candidate vertices from the database graph by using E-Index and V-Index described in the previous section. In the filtering phase, the main task is to find candidate vertices of each query graph vertex from the database graph. Our method has two advantages over the existing methods. First, we reduce time to retrieve candidate vertices by using E-Index, a pre-constructed index structure. Second, since E-Index stores information on vertex pairs that are directly connected to each other, we can retrieve only those candidate vertices that are directly connected to each other from E-Index. This may reduce the size of the candidate vertices.

Before we proceed, let us first give an overview of the filtering phase. In our approach, we obtain candidate vertices indirectly through candidate edges. In other words, endpoint vertices of the candidate edges will be our candidate vertices. To this end, we need to find candidate edges of the edges in the query graph. Note that we do not need to find candidate edges of every edge in the query graph. This is because we need only those edges that are enough to cover every vertex in the query graph. Here a spanning tree of the query graph is useful. Thus, we select a spanning tree of the query graph and find candidate edges of the edges in the spanning tree. Finally, we obtain candidate vertices from the candidate edges found. In what follows, we describe the filtering phase in more detail. Section 4.1 describes the spanning tree selection process and section 4.2 explains how to obtain candidate vertices.

4.1 Selecting a Spanning Tree

Given a query graph, we select a spanning tree of the query graph whose edges are to be used to find candidate edges. There may exist many different spanning trees of the query graph. Here we need a way to pick a "good" spanning tree for filtering out candidate edges. We have the following observation. Given an edge in the query graph, the number of its candidate edges can be roughly estimated by the degree sum of its two endpoint vertices. This is because candidate edges must have degree sums large than or equal to that of the query graph edge, and the larger degree sum of the query graph edge, the smaller possibility that there are many candidate edges with larger degree sums. Thus degree sums may indicate the "goodness" of the query graph edges. Based on this observation, we take the degree sum of each edge as its weight, compute the maximum cost spanning tree (we can obtain the maximum cost spanning tree by multiplying each edge weight by -1 and applying minimum spanning tree algorithms such as Kruskal's or Prim's algorithms), and use the resulting tree to retrieve candidate edges.

4.2 Discovering Candidate Vertices

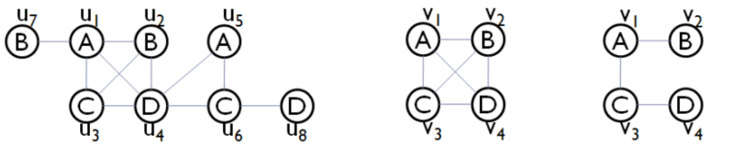
After selecting a spanning tree, we find candidate vertices through candidate edges. We first obtain candidate edges of the edges in the spanning tree. We need to decide the order of visiting edges in the spanning tree. Either breadth-first search or depth-first search over the spanning tree can be used to determine the edge visiting order. During graph search, we record edge visiting order. Let the determined order be e_1, e_2,…, $e_{|V(q)-1|}$, where|V(q)| is the number of vertices in the query graph. Now we visit each edge in the order determined, find candidate edges, and obtain candidate vertices from the candidate edges. This step proceeds as follows. For each vertex v in the query graph, we maintain a candidate vertex set, denoted C(v), and initialize it as empty. Now we visit each edge one by one. First, for the first edge $e_1 = (v_1, v_1')$ in the spanning tree, we probe L-Index by using the key $(l(v_1),l(v_1'))$ and obtain a pointer to a D-Index and then perform a range query over the D-Index to retrieve the list of candidate edges. We then compare neighbor information of the candidate edges with that of the query edge to find the final candidate edges. For each candidate edge found, we add its two endpoint vertices to the corresponding candidate vertex sets of v_1 and v_1', respectively.

For the remaining edges in the spanning tree, we process them slightly differently for better efficiency. For each edge $e_i = (v_i, v_i')$ (i>1) in the spanning tree, we probe L-Index and D-Index as before and obtain a list of candidate edges. Here we make the following observation to reduce the cost of comparing neighbor information of edge e_i with the candidate edges. Given a candidate edge $e_i'=(u_i, u_i')$, if u_i is not contained in the candidate vertex set of v_i, i.e., if $u_i \notin C(v_i)$, then e_i' cannot be the final candidate edge of e_i. This is because it means u_i has been already filtered out when finding candidate edges of the query edge of the form (v, v_i). Based on this observation, we first check if $u_i \in C(v_i)$. If this is true, we proceed to check whether $nInfo(v_i') \leq nInfo(u_i')$. If this is also true, we add u_i' to $C(v_i')$.

5 Verification

In this section, we describe our verification algorithm based on depth first search with backtracking. In the verification phase, we perform subgraph isomorphism testing. To this end, we generate subgraph isomorphism mappings between the query graph and the database graph. We start with an empty mapping M and expand it incrementally by adding possible matching vertices of the vertices in the query graph. Depth first search with backtracking is used to expand M systematically. We first determine the order of visiting the vertices in the query graph. At depth d, we visit the d-th vertex in the query graph and find possible matching vertices from its corresponding candidate vertex set. If we do not find any possible matching vertex at depth d, we remove the most recently added matching vertex from M and backtrack to depth d-1. If we arrive at depth |V(q)|, then we have found a subgraph isomorphism mapping.

Our verification algorithm, FastMatch, uses three kinds of heuristics to speed up the verification process. Before describing FastMatch, we first explain each of these heuristics and how they may speed up the verification process. Figure 2 shows our running example that will be used in the rest of the paper. Figure 2(a), 2(b), and 2(c) show a database graph, query graph, and the spanning tree selected in the filtering phase, respectively.



(a) Database graph (b) Query graph (c) Spanning tree

Fig. 2. Running example

5.1 Heuristics for Fast Verification

The first heuristic used by FastMatch is called *vertex ordering*. As mentioned above, before we start depth first search, we decide the order of visiting the vertices in the query graph. It is important to carefully decide the order of visiting the vertices in the query graph for the performance of the verification process. Given a vertex in the query graph, we can save more work if its candidate vertex size is large and it is visited more later than some vertices with smaller candidate sizes. Thus we visit the vertices of the query graph in the increasing order of their candidate sizes. This heuristic is also employed by NOVA [13]. In our method, we modify the heuristic in such a way that candidate edge information obtained from the filtering phase can be used. More specifically, we determine vertex visiting order by using the spanning tree selected in the filtering phase. We maintain a visited vertex set, denoted *Visit*. We start with a vertex with the smallest candidate size and add it to *Visit*. We choose the next vertex to visit among the vertices not in *Visit* and directly connected to any of the vertices in *Visit* on the spanning tree. The vertex with the smallest candidate size is selected as the next vertex to visit and added to *Visit*. This procedure is repeated until there is no vertex to visit.

The second heuristic is called *connection-aware forward checking*. After selecting a possible vertex mapping (v_d, u_d) between a vertex v_d in the query graph and one of its candidate vertices u_d at depth d, we check the connections between u_d and the candidate vertices of unvisited vertices v_i ($i>d$) that are directly connected to v_d. The candidate vertices that do not have any connection with u_d are marked as invalid. Unlike the conventional forward checking, in our method, we do not need to check the connections between u_d and the candidate vertices of v_i ($i>d$) if (v_d,v_i) is an edge in the spanning tree. This is because we can easily retrieve candidate vertices directly connected to u_d by using connectivity information between candidate vertices. How to do this will be described later. Note that our method eliminates the connectivity checking operations over the edges in the spanning tree altogether, which may result in a considerable performance gain.

The last heuristic is called *incompatibility learning*. When backtracking during depth first search, we lose the results of connectivity checking operations from deeper depths, which may lead to duplicated connectivity checking operations at later times. To avoid this problem, we keep track of the candidate vertices of v_i ($i>d$) that are invalidated by each matching vertex u_d of v_d during depth first search and use it to remove unnecessary connectivity checking operations. For ease of presentation and interest of space, we will omit the incompatibility learning heuristic in the description of our verification algorithm (Algorithm 1 in section 5.2).

5.2 The FastMatch Algorithm

The algorithm FastMatch is formally described in Algorithm 1. First it determines the order of visiting the vertices in the query graph by applying the vertex ordering heuristic, which is described in section 5.1. Then it calls RecursiveFastMatch to perform subgraph isomorphism testing.

RecursiveFastMatch performs depth first search and finds all possible subgraph isomorphism mappings. At depth d, the candidate vertices of vertex v_d are considered as possible matching vertices of v_d. By calling GetQualifedCandidateVertices, ResursiveFastMatch retrieves only those candidate vertices that are qualified as a matching vertex of v_d. The details of GetQualifiedCandidateVertices will be described later. For each qualified candidate vertex u, ResursiveFastMatch sets u as the matching vertex of v_d (line 6) and then check if we have arrived at depth $|V(q)|$ (line 7). If this is true, we have found a subgraph isomorphism mapping. Thus we output M and backtrack. If not, we expand M further by going down to the next level. Before going down, the connection-aware forward checking heuristic is applied to filter out unqualified candidate vertices with respect to u at deeper depths (line 10). The connection-aware forward checking heuristic is described in section 5.1. Finally, RecursiveFastMatch recursively calls itself with an increased depth.

Algorithm 1. FastMatch

```
Input   q: query graph, G: database graph, M: current mapping, initially empty
        d: depth, initially 1
Output  All subgraph isomorphism mappings
 1.     Apply the Vertex Ordering heuristic
 2.     RecursiveFastMatch(q,G,d,M)
 3.     function RecursiveFastMatch(q,G,d,M):
 4.         QC ← GetQualifiedCandidateVertices($v_d$,M)
 5.         for each u in QC do
 6.             M[d] = u
 7.             if d = |V(q)| then
 8.                 Output M and backtrack
 9.             else
10.                 Apply the Forward Checking heuristic
11.                 RecursiveFastMatch(q,G,d+1,M)
12.             end if
13.         end for
14.     end function
```

5.3 The GetQualifiedCandidateVertices Function

When we arrive at depth d, only those candidate vertices that are not marked as invalid can be a qualified matching vertex of v_d. To find qualified candidate vertices efficiently, GetQualifiedCandidateVertices uses a pre-constructed data structure. For each edge (v_i,v_j) in the spanning tree selected in the filtering phase, we build a *connection map* (CM). The order of constructing CMs for the edges in the spanning tree is determined by the vertex ordering heuristic. The key of CM for (v_i,v_j) is a candidate vertex u_i of v_i and the value is the list of candidate vertices of v_j that are directly connected to u_i. The CMs for the edges in the spanning tree can be easily constructed from the candidate edges discovered in the filtering phase.

For example, Table 1 shows the candidate edges for the edges in the spanning tree in Figure 2(c). Figure 3 shows the connection maps populated from Table 1. As an example, the CM for (v_1,v_2) in Figure 3 indicates that u_2 and u_7 are the candidate vertices of v_2 that are directly connected to the candidate vertex u_1 of v_1.

Table 1. Candidate edges for the edges in the spanning tree in Figure 2(c)

Edges in the spanning tree	(v_1,v_2)	(v_1,v_3)	(v_3,v_4)
Candidate edges	(u_1,u_2) (u_1,u_7)	(u_1,u_3) (u_5,u_6)	(u_3,u_4) (u_6,u_8)

Let (v_i,v_d) (i<d) be an edge in the spanning tree of the query graph. Note that, given v_d, such an edge is uniquely determined by the vertex ordering heuristic (for more details, see section 5.1.) GetQualifiedCandidateVertices finds the candidate vertices u_d of v_d that are directly connected to u_i (obtained from $M[v_i]$) by consulting the

(v_1,v_2)

Key	Value
u_1	u_2,u_7

(v_1,v_3)

Key	Value
u_1	u_3
u_5	u_6

(v_3,v_4)

Key	Value
u_3	u_4
u_6	u_4,u_8

Fig. 3. Connection maps for the edges in Table 1

connection map for the edge (v_i,v_d). Additionally, it removes, from the list of candidate vertices obtained, those candidate vertices that are marked as invalid and returns the remaining candidate vertices as the qualified candidate vertices.

5.4 Improving the Connection-Aware Forward Checking Heuristic

Connection maps introduced in the previous section can also be used to enhance the connection-aware forward checking heuristic. In this section, we briefly describe how to do that. In the connection-aware forward checking heuristic, given a vertex mapping (v_d, u_d) at depth d, we check the connections between u_d and the candidate vertices u_i of unvisited vertices v_i (i>d) that are directly connected to v_d. Note that we need to consider only u_i's that are not marked as invalid. Instead of checking whether every u_i is not marked as invalid, we can retrieve valid u_i's more efficiently by using connection maps. Given an edge (v_d,v_i), if an edge (v_j,v_i) (j<d) exists in the query graph, we can retrieve u_i's that are directly connected to u_j by probing the connection map for edge (v_j,v_i) with u_j as a key. We need to check validity for only these u_i's. For the other edges, we cannot use connection maps to reduce the number of validity checking operations.

5.5 Discussion

We can estimate how many connectivity checking operations can be reduced by connection-aware forward checking as follows. Let us first calculate how many of them can be reduced at depth d over an edge (v_d,v_j) on the spanning tree T during depth first search. In the worst case, depth first search arrives at depth d for $\prod_{k=1}^{d}|C(v_k)|$ number of times, where $|C(v_k)|$ is the number of candidate vertices of v_k (this happens when all required connections between candidate vertices exist in the database graph; in such a case, the basic forward checking cannot eliminate any of the candidate vertices during depth first search). For each arrival at depth d, connection-aware forward checking eliminates $|C(v_j)|$ connectivity checking operations over edge (v_d,v_j). Thus, the number of connectivity checking operations reduced at depth d over edge (v_d,v_j) is $|C(v_j)| \times \prod_{k=1}^{d}|C(v_k)|$. Finally, the total number of connectivity checking operations reduced over all edges on the spanning tree T is:

$$\sum_{(v_d,v_j)\in E(T)} \left(|C(v_j)| \times \prod_{k=1}^{d} |C(v_k)| \right)$$

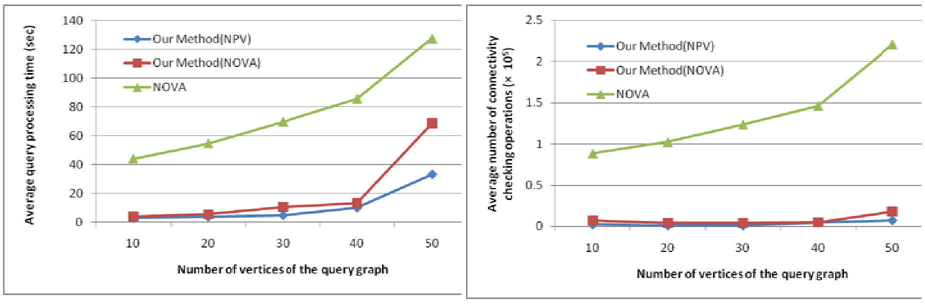
Note that the above formula computes the number of connectivity checking operations that can be reduced by connection-aware forward checking in the best case scenario. In general, it might be less than the number suggested by the formula.

6 Evaluation

In this section, we compare the performance of our method with that of NOVA [13], a state-of-the-art subgraph matching technique using synthetic data sets. Our method and NOVA have been implemented using C++ with the STL library. The experiments were conducted on a PC with a 3.0GHz quad core CPU and 4GB main memory running windows server 2003. We use two kinds of vertex signatures in our method. In the graphs that show experimental results below, "Our Method (NOVA)" represents the one that uses the vertex signature from NOVA and "Our Method (NPV)" the one that uses that from NPV [9]. In various experiments, ten database graphs are randomly generated and query graphs are selected by extracting subgraphs from the database graphs. We evaluate each method over the ten randomly generated database graphs and average the results.

6.1 Effect of the Size of the Query Graph

In this section, we evaluate our method over various query graph sizes from 10 to 50 (the size of a query graph is the number of its vertices.) Each of the randomly generated database graphs has 5,000 vertices, 80,000 edges and 20 distinct vertex labels. The average vertex degree of each query graph is set to 2.8. Figure 4 shows the experimental results. Figure 4(a) shows the query processing time and figure 4(b) shows the number of connectivity checking operations required as the query graph size increases. Because our verification algorithm reduces the number of connectivity checking operations greatly, both versions of our method show significantly better performance than that of NOVA. Our Method (NPV) outperforms Our Method (NOVA) because the vertex signature of NPV has a superior filtering power than that of NOVA.

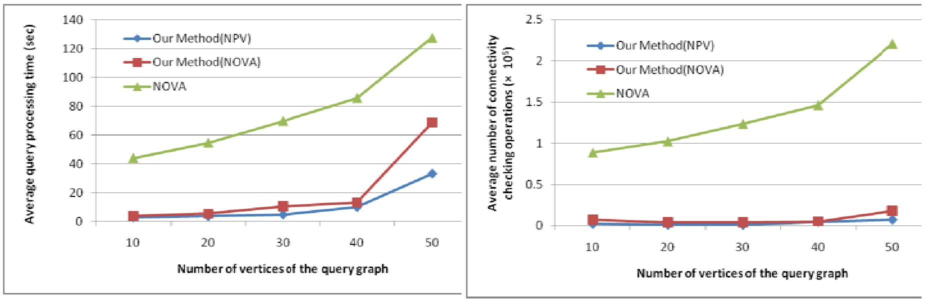


(a) Query processing time (b) Number of connectivity checking operations

Fig. 4. Experimental results over various query graph sizes

6.2 Effect of the Average Degree of the Query Graph

In this section, we evaluate our method by varying the average degree of the query graph from 3.4 to 5.8. This time we generate ten random database graphs, each of which has 1,000 vertices, 40,000 edges and 20 distinct vertex labels. The query graph size is set to 20. Figure 5 shows the results of this experiment. Figure 5(a) shows the query processing time and figure 5(b) shows the number of connectivity checking operations as the degree of the query graph increases. As the degree of a vertex in the query graph becomes large, the number of its candidate vertices in the database graph decreases. This is because a qualified candidate vertex must have a degree larger than or equal to that of the query graph vertex. Thus as the average degree of the query graph increases, more vertices are filtered out in the filtering phase, which results in low query processing time in every method. When the average degree is small, however, many candidate vertices are used as input to subgraph isomorphism testing in the verification phase. In this case, the number of connectivity checking operations determines the performance of the verification phase. Both kinds of our method significantly outperform NOVA in such a case.



(a) Query processing time (b) Number of connectivity checking operations

Fig. 5. Experimental results over various average query graph degrees

7 Related Work

Subgraph matching, which is to find all subgraphs of a database graph that are isomorphic to a query graph, is an important operation that has many applications over a wide range of areas. Subgraph matching requires subgraph isomorphism testing, which is NP-complete [14]. Many techniques for subgraph matching have been proposed thus far.

In graph databases, queries related to subgraph matching can be divided into two types. A subgraph containment query finds all graphs that contain a subgraph which is isomorphic to the query graph. Many techniques have been proposed for this type of query [5],[6],[7],[8],[9]. In these techniques, they assume that a graph database contains many small graphs. A *subgraph matching query* finds, from a single database graph, all subgraphs that are isomorphic to the query graph. General purpose subgraph matching algorithms are proposed in [10],[11]. Recently, techniques specifically designed for subgraphs matching queries in a large graph have been proposed in [12],[13].

Both of [12] and [13] are based on a *filtering-and-verification* framework. In the filtering phase, they compare the signatures of vertices and filter out candidate vertices if they are not qualified as a matching vertex. In the verification phase, subgraph isomorphism testing is performed and all matched subgraphs of the database graph are returned to the user. GADDI [12] uses *discriminative substructures*, which are small substructures in the intersection of the neighborhoods of two vertices, as vertex signatures. In NOVA [13], label distribution information around vertices are used as vertex signatures. Both of these methods can be classified into a *vertex-based framework* in the sense that they use only vertex information to filter out unqualified vertices. Unlike these methods, our method, which is an *edge-based framework*, uses edge connectivity information in both the filtering and verification phases for fast subgraph matching.

8 Conclusions

In this paper, we have proposed an edge-based framework for fast subgraph matching in a large graph. Our method is based on a filtering-and-verification framework. Unlike existing vertex-based frameworks, our method uses edge connectivity information in both of the filtering and verification phases for fast subgraph matching. It not only reduces the size of input to subgraph isomorphism testing, but also avoids unnecessary connectivity checking operations. We verify through experimental evaluation that our method significantly outperforms existing approaches for subgraph matching in a large graph.

Acknowledgments

This work was supported in part by Korea Institute of Science and Technology Information (KiSTi) and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0018865).

References

1. Bröcheler, M., Pugliese, A., Subrahmanian, V.: COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In: ASONAM (2010)
2. Tian, Y., McEachin, R.C., Santos, C., States, D.J., Patel, J.M.: SAGA: a subgraph matching tool for biological graphs. Bioinformatics 23, 232–239 (2007)
3. Backstrom, L., Dwork, C., Kleinberg, J.: Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In: WWW (2007)
4. Cheng, J., Fu, A.W.c., Liu, J.: K-isomorphism: privacy preserving network publication against structural attacks. In: SIGMOD (2010)
5. Giugno, R., Shasha, D.: GraphGrep: A Fast and Universal Method for Querying Graphs. In: ICPR (2002)
6. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD (2004)
7. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: SIGMOD (2007)
8. Jiang, H., Wang, H., Yu, P.S., Zhou, S.: Gstring: A novel approach for efficient search in graph databases. In: ICDE (2007)

9. Wang, C., Chen, L.: Continuous Subgraph Pattern Search over Graph Streams. In: ICDE (2009)
10. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. J. ACM 23(1), 31–42 (1976)
11. 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, 1367–1372 (2004)
12. Zhang, S., Li, S., Yang, J.: GADDI: distance index based subgraph matching in biological networks. In: EDBT (2009)
13. 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)
14. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158 (1971)