Finding All Subgraph Isomorphisms Efficiently In a Large Database

Kaoru Katayama, Tokyo Metropolitan University
Ernest Weke Maina, Tokyo Metropolitan University

We study the problem of scalable enumeration of all subgraph isomorphisms to a query graph that exist in a graph database. We propose several improvements and extensions to known algorithms to solve this problem that result in order of magnitude improvement in scalability.

Graphs have become increasingly important structures for representing and understanding complex structures. Deciding whether a graph is a subgraph of another graph is called the subgraph isomorphism problem. In this study we are concerned with the retrieval of all subgraph isomorphisms to a query that are present in a graph database. We define the retrieval of isomorphic subgraphs to satisfy two conditions. First, a decision whether or not a graph isomorphic to a subgraph of a query graph exists in the graph database. Second, the actual enumeration of all graphs present in the database isomorphic to subgraphs of the query. However the retrieval of all subgraph isomorphisms is complicated and expensive because the subgraph isomorphism problem is NP-complete. Recently, there have been several studies on efficiency of retrieval of subgraph isomorphisms from a graph database but few have used decomposition methodology to reduce the larger, difficult graph matching problem to smaller more managable sub-problems in a divide and conquer fashion. Messmer et al. studied an interesting method for retrieval of induced subgraph isomorphisms of a query graph. They proposed a so-called *Network Method* that stores graphs in a network structure by recursively decomposing the individual graphs in a divide and conquer fashion, and the algorithms to facilitate retrieval using the structure.

In this paper, we reformulate the decomposition strategy for the *Network Method* and extend it to cover retrieval of non-induced subgraph isomorphisms of a query graph. We also propose a *Fast Network Method* where the recursive random decomposition of graphs is followed by the rearrangement of the fragments so that only two subgraphs result.

We present experimental results where we compare our proposed algorithms with two well known algorithms: Messmer's all subgraph isomorphism detection Algorithm that efficiently aggregates multiple graphs in a database and the isomorphism detection only algorithm, VF2, which is the state-of-the-art for sequential one-on-one isomorphism testing. The results show that the proposed improvements result in an order of magnitude improvement in scalability over the original *Network Method* for query graphs of up to 500 nodes to a database of 20,000 graphs. We also show a substantial improvement over the VF2 algorithm despite the increased workload of detecting all subgraph isomorphisms. Our method is particularly suited to larger query graphs or very larger graph databases.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Experientation, Algorithms, Performance

Additional Key Words and Phrases: Graph Database, Graph Decomposition, Subgraph Isomorphism

ACM Reference Format:
Katayama, K., and Maina, E.W.,2011. Finding All Subgraph Isomorphisms Efficiently In a Large Database.

This work is supported by the National Science Foundation, under grant CNS-0435060, grant CCR-0325197 and grant EN-CS-0329609.
Author's addresses: K. Katayama and E.W. Maina, Graduate School of System Design, Tokyo Metropolitan University.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ACM Trans. Embedd. Comput. Syst. 9, 4, Article 39 (March 2010), 28 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The power and importance of graph structures is clearly evident in the numerous and varied use of graphs in such applications as: modeling of proteins [Chi et al. 2005], molecular structures of compounds in chemistry[Willett et al. 1998][Agrafiotis et al. 2007], organization of entities in images[Petrakis and Faloutsos 1997] [Burge and Kropatsch 1999], representation of components in computer aided design (CAD) drawings [Cordella et al. 2000], topology of sensor networks[Li et al. 2003],and social and information networks[Cai et al. 2005]. As a result a tremendous amount of structured data is being accumulated in large databases. An essential problem in managing the large amount of graphs and graph queries is the efficiency of query processing. In this article we address the problem of scalable enumeration of all subgraph isomorphisms existing in a database of graphs.

A subproblem of the subgraph isomorphism enumeration problem is the issue of deciding whether a graph contains another graph, which is called the *subgraph isomorphism problem*. A subgraph isomorphism exist between two graphs if there exists one-to-one mapping between the smaller graph and a subgraph of the larger graph such that edge the adjacencies are preserved. Testing for subgraph isomorphism for an n-node graph in an m-node larger graph is a combinatorial matching exercise with m^n possibilities, hence it is expensive. Therefore for a given query, evaluating a database for subgraph isomorphism by individually testing each graph in the database is inefficient. In fact, the subgraph isomorphism problem is known to be NP-complete [Cook 1971]. Enumeration of all subgraph isomorphisms clearly requires solving the subgraph isomorphism problem multiple times, one for each instance of isomorphism at the very least. In many cases, the graph database is also very large which makes it necessary to build a framework to facilitate efficient query processing.

There are two basic approaches that past research has taken regarding the problem of finding graph and subgraph isomorphisms. The first approach is based on group theoretic concepts and aims to classify the adjacency matrices of graphs into permutation groups. With this, it is possible to prove that there exists a moderately exponential bound for the general graph isomorphism problem [Babai 1981]. Also, by imposing certain restrictions on graphs it is possible to derive algorithms that have polynomial bound on complexity. For example Luks and Hoffman[Hoffmann 1982] describe a polynomially bound method for the isomorphism detection of graphs with bounded valence. These methods, though theoretically interesting, in practice they have a large constant overhead and are therefore impractical. They also apply only to graph isomorphism detection, but not the detection and enumeration of subgraph isomorphism which is required for a wide range of applications.

The second approach to graph and subgraph isomorphism is the use of heuristics for graph matching. This approach is more practically oriented and aims directly at developing practical procedures. Most of the algorithms are based on search tree with backtracking. One of the first publications in this field was Corniel and Gotlieb [Corneil and Gotlieb 1970]. A major improvement of the bactracking method was then presented by Ullmann, who introduced a refinement method which reduces the search space of the backtracking procedure remarkably [Ullmann 1976].

The methods for subgraph isomorphism detection mentioned so far work only with two graphs at a time. However in many applications there is more than one model that must be matched with the input graph. Consequently it is necessary to apply the subgraph isomorphism algorithm to each input pair resulting in a computation time that is linearly dependent on the size of the database. This dependency becomes prohibitive

when the number of graphs in the database is large. We can classify a third, more recent and interesting Approach. This is represented by the network based approach to graph detection proposed by Messmer and Bunke [Messmer and Bunke 2000] . In the new approach the tree is replaced by a network whose nodes contain subgraphs of the model graphs and traversing the network is equivalent to constructing a graph from the subgraphs. In this method, a powerful tool for obtaining efficient solutions to graph isomorphism detection, enumeration and problems is divide and conquer, which is expressed in the form of random graph decomposition. Decomposition techniques are an instantiation of the divide and conquer paradigm to overcome redundant work for independent partial problems. The main features of the Network based approach are as follows:

(1) Offline decomposition of the model graphs into smaller graphs using random decomposition
(2) Decomposed subgraphs of model graph are saved in a network structure as the database. The process is repeated for all the model graphs in the database.
(3) Subgraphs that appear in the same or multiple model graphs multiple times are represented in the network structure only once.
(4) During graph matching the subgraphs are matched only once to the input graphs, the algorithm is only sublinearly dependent on the number of model graphs.

For the above algorithms, given a database D comprising of a set of graphs $g = g_1, g_2, \ldots g_n$ and query Q consisting of a set of graphs $q = q_1, q_2, \ldots q_n$, there are, in general, two kinds of graph search problems that we can define. By far the traditional and the most popular one is the subgraph query. Given a graph database D and a subgraph query Q with a query graph q, the answer to Q is the set of $\{g \| g \in D$ and q is a subgraph of $g\}$. The second search problem is a graph containment search where Given a graph database D and a subgraph query Q with a query graph q, the answer to Q is the set of $\{g \mid g \in D$ and q is a supergraph of $g\}$. Graph containment search with exclusion logic has recently become a active area of research [Chen et al. 2007] [Zhang et al. 2011].

In this paper, we extend the method proposed by Messer et al.[Messmer and Bunke 2000], so-called *Network Method,(NA)*, to support *subgraph isomorphism* queries. Fig.1 shows an example of a subgraph isomorphism query. We also reformulate a *Fast Network Method* to increase the scalability of this method. Our main contributions are as follows.

(1) The *Network Method* originally only supports induced subgraph isomorphism query. We extend it cover *subgraph isomorphism* queries.
(2) In the *Network Method*, much of graph decomposition is performed on graphs at random potentially creating many graph fragments. We add a new recombining process after each decomposition which is active when more than a pair of subgraphs result from the decomposition. Recombining results in exactly two larger graphs for each decomposition step, where possible. As a consequence, we are able to drastically reduce the potential rapid increase in matchings.
(3) We formulate and implement a *Fast Network Method* on the offline preparation of the database. The new algorithm performs recombination on each recursive decomposition in the preprocessing during database creation. as well as during the actual query processing.
(4) We formulate and implement the *Fast Network Method* for recombination of the decomposed query during online query processing on the database.

In this work we refer to Messmer's formulation as the *Network Method*.

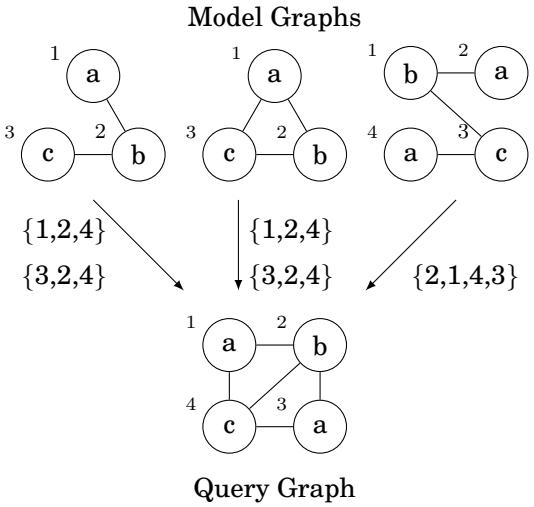

Fig. 1: Subgraph isomorphism query

We present experimental results where we compare our proposed algorithms with two well known algorithms: Messmer's[Messmer and Bunke 2000] all subgraph isomorphism detection Algorithm that efficiently aggregates multiple graphs in a database and the isomorphism detection only algorithm, VF2[Cordella et al. 2001], which is the state-of-the-art on sequential one-on-one isomorphism testing. The results show that the proposed improvements result in an order of magnitude increase in scalability over the original *Network Method* for query graphs of up to 500 nodes to a database of 20,000 graphs. We also show a substantial improvement over the VF2 algorithm despite the increased workload of detecting all subgraph isomorphisms. Our method is particularly suited to larger query graphs or very larger graph databases.

2. RELATED WORK

The subgraph isomorphism problem is a known NP-complete problem[Cook 1971]. In order to solve this problem efficiently, many algorithms have been proposed such as Ullman[Ullmann 1976], Nauty[McKay 1981] and VF2[Cordella et al. 2001].

Recently, there have been several studies of graph search on large graph databases. We can classify these studies into two broad categories.

The first one is retrieval of graphs, typically larger than the query graph, that include the query graph from a graph database. In this category graph indexing methods have been extensively proposed. For example, GraphGrep[Shasha et al. 2002] takes the path as the basic indexing unit while gIndex[Yan et al. 2004] generates an index composed of frequent subgraphs, except for redundant subgraphs. FG-Index[Cheng et al. 2007] also utilizes frequent subgraphs and does not need to calculate subgraph isomorphism if query graph is included in frequent subgraphs. GDIndex[Williams et al. 2007] takes all subgraphs in graph database as the index features in order to facilitate the retrieval.

The second category is the retrieval of model graphs, typically smaller than the query graph and resemble a part of the query graph. A good example of this is approach is the *Network Method*[Messmer and Bunke 2000]. More recently, Chen et. al. proposed a containment search algorithm, cIndex[Chen et al. 2007]. The cIndex indexing model is based on *contrast subgraphs* that capture differences between feature graphs derived from database model graphs, and graphs. The cIndex definition of containment search makes the *Network Method*, in some respect, the more general case of containment search, where, the index features are derived from the database by random decomposition and the query is applied in a decomposed form as multiple graph fragments. In this way the objective of the *Network Method* and cIndex are identical, but cIndex performs aggressive optmisation of the model feature set graphs while the *Network Method* relies on a form of exclusion logic at runtime to reduce the problem space.

3. PRELIMINARIES

In this section, we present the definitions relevant to graph and graph isomorphism. For convenience, we focus on undirected labeled graphs. Extension to other kinds of graphs is straightforward.

Definition 3.1. A *labeled graph* is a six-element tuple $g = (V, E, L^v, L^e, \mu, \nu)$ where V is a vertex set, E is a edge set, both μ and ν are functions assigning labels to the vertices and edges respectively. L^v and L^e denotes the set of vertex and edge labels respectively.

Following the convention, we denote a vertex set of a graph g_1 as V_1 and the edge set as E_1.

Definition 3.2. A graph $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$ is *isomorphic* to another graph $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ if there exists a bijection ϕ such that

(1) for every vertex $v \in V_1$, $\phi(v) \in V_2$ and $\mu_1(v) = \mu_2(\phi(v))$,
(2) for every edge $(v, u) \in E_1$, $(\phi(v), \phi(u)) \in E_2$ and $\nu_1((u, v)) = \nu_2((\phi(v), \phi(u)))$

If g_1 is isomorphic to g_2, ϕ is called a *graph isomorphism* and we use the notation $g_1 \underset{iso}{=} g_2$.

Definition 3.3. A graph $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$ is *subgraph isomorphic* to another graph $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ if there exists a injection ϕ such that

(1) for every vertex $v \in V_1$, $\phi(v) \in V_2$ and $\mu_1(v) = \mu_2(\phi(v))$,
(2) for every edge $(v, u) \in E_1$, $(\phi(v), \phi(u)) \in E_2$ and $\nu_1((u, v)) = \nu_2((\phi(v), \phi(u)))$

If g_1 is subgraph isomorphic to g_2, ϕ is called a *subgraph isomorphism*, g_1 is called a *subgraph* of g_2 and we use the notation $g_1 \subseteq g_2$.

Definition 3.4. Given a graph $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$, a subgraph $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ and a subgraph isomorphism $\phi(\phi : V_2 \rightarrow V_1)$, g_2 is *induced subgraph* of g_1 if it satisfies

— $\exists(\phi_1(u), \phi_1(v)) \in E_1 \Rightarrow \exists(u, v) \in E_2$ for all $u, v \in V_2$

If g_2 an is induced subgraph of g_1, ϕ is called an *induced subgraph isomorphism* and we use the notation $g_2 \underset{ind}{\subseteq} g_1$.

We further introduce definitions and notations that we use in discussing the processing of graphs. Since we need to support induced subgraphs as well as subgraphs, we provide the respective definitions for the decomposition, union and subtraction of

graphs for both these cases. Note that during random decomposition, the graphs are partitioned into roughly two equal number of vertices or edges for efficient processing.

Definition 3.5. The union of induced subgraphs $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ with respect to a set of interconnecting edges $E^{'} \subseteq (V_1 \times V_2)$ with a labeling function $\nu : E^{'} \rightarrow L_E$, is the graph $g = (V, E, L^v, L^e, \mu, \nu)$ if:

1. $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$,
2. $E = E_1 \cup E_2 \cup E^{'}$
3. $$\mu(v) = \begin{cases} \mu_1(v) & \text{if } v \in V_1 \\ \mu_2(v) & \text{if } v \in V_2 \end{cases}$$
4. $$\nu(e) = \begin{cases} \nu_1(e) & \text{if } e \in E_1 \\ \nu_2(e) & \text{if } e \in E_2 \\ \nu(e) & \text{if } e \in E^{'} \end{cases}$$

We denote this union of induced subgraphs g_1 and g_2, with respect to a set of interconnecting edges by $g_1 \cup_E g_2$.

Definition 3.6. The *decomposition to induced subgraph* of graph $g = (V, E, L^v, L^e, \mu, \nu)$ is the partitioning of the set of vertices (also known as edge cutting) to form a pair of induced subgraphs $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ where the following conditions hold:

1. $V_1 \subseteq V$, $V_2 \subseteq V$ and $V_1 \cap V_2 = \emptyset$
2. $E_1 = E \cap (V_1 \times V_1)$, $E_2 = E \cap (V_2 \times V_2)$, $E^{'} \subseteq (V_1 \times V_2)$ and $E_1 \cup E_2 \cup E^{'} = E$ where $\nu : E^{'} \rightarrow L_E$ is the edge labeling function for edges to V_1 and V_2.
3. $\mu_1, \mu_2, \nu_1, \nu_2$ are the restrictions of μ and ν to V_1, V_2 and E_1, E_2 respectively, that is:

$$\mu_1(v) = \begin{cases} \mu(v) & \text{if } v \in V_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mu_2(v) = \begin{cases} \mu(v) & \text{if } v \in V_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu_1(e) = \begin{cases} \nu(e) & \text{if } e \in E_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu_2(e) = \begin{cases} \nu(e) & \text{if } e \in E_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We denote the *decomposition to induced subgraphs* of graph g by partition of a set edges to subgraphs g_1 and g_2 as $g \underset{V}{\rightarrow} \{g_1, g_2\}$.

If the assignment of the edges to g_1 and g_2 is performed at random, we get randomly decomposed subgraphs. We denote *random decomposition to subgraph* by $g \underset{V_r}{\rightarrow} \{g_1, g_2\}$.

Decomposition to induced subgraphs uniquely defines induced subgraphs of g denoted by $g_1 \underset{ind}{\subseteq} g$, $g_2 \underset{ind}{\subseteq} g$ and satisfies $g_1 \cup_E g_2 \underset{iso}{=} g$.

Definition 3.7. The *difference* between a graph g and its induced subgraph $g_1 \underset{ind}{\subseteq} g$ is the induced subgraph $g_2 \underset{ind}{\subseteq} g$ that is defined by $V_2 = V - V_1$. We use the notation $g \underset{ind}{-} g_1$ to represent the subtraction of induced subgraph.

Note that both g_1 and g_2 are induced subgraphs of g and there are edges that are not included by either g_1 or g_2 as shown in definition 3.6

Next we define union, decomposition and subtraction for subgraphs.

Definition 3.8. The union of subgraphs $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ given a set of common vertices $V^{'} = V_1 \cap V_2$ with respect to $V^{'}$ is the graph $g = (V, E, L^v, L^e, \mu, \nu)$ if:

1. $E = E_1 \cup E_2$ and $E_1 \cap E_2 = \emptyset$
2. $V = V_1 \cup V_2 \cup V^{'}$
3. $$\mu(v) = \begin{cases} \mu_1(v) & \text{if } v \in V_1 - V_2 \\ \mu_2(v) & \text{if } v \in V_2 - V_1 \\ \mu(v) & \text{if } v \in V_1 \cap V_2 \end{cases}$$
4. $$\nu(e) = \begin{cases} \nu_1(e) & \text{if } e \in E_1 \\ \nu_2(e) & \text{if } e \in E_2 \end{cases}$$

We denote this union of subgraphs g_1 and g_2, with respect to a set of common vertices V by $g_1 \cup_V g_2$.

Definition 3.9. The *decomposition to subgraph* of a graph $g = (V, E, L^v, L^e, \mu, \nu)$ is the partitioning of the set of edges (also known as vertex sharing) to form a set of subgraphs $g_1 = (V_1, E_1, L_1^v, L_1^e, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, L_2^v, L_2^e, \mu_2, \nu_2)$ where the following conditions hold:

1. $E_1 \subseteq E$, $E_2 \subseteq E$, $E_1 \cup E_2 = E$ and $E_1 \cap E_2 = \emptyset$
2. $\cup_{i \in [1,2]} V_i = V$ and $V_1 \cap V_2 = V^{'}$ where $V^{'}$ are vertices common to sets V_1 and V_2.
3. $\mu_1, \mu_2, \nu_1, \nu_2$ are the restrictions of μ and ν to V_1, V_2 and E_1, E_2 respectively, such that:

$$\mu_1(v) = \begin{cases} \mu(v) & \text{if } v \in V_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mu_2(v) = \begin{cases} \mu(v) & \text{if } v \in V_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu_1(e) = \begin{cases} \nu(e) & \text{if } e \in E_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu_2(e) = \begin{cases} \nu(e) & \text{if } e \in E_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We denote the *decomposition to subgraphs* of graph g by partition of a set edges to subgraphs g_1 and g_2 as $g \underset{E}{\rightarrow} \{g_1, g_2\}$.

If the assignment of the edges to g_1 and g_2 is performed at random, we get randomly decomposed subgraphs. We denote *random decomposition to subgraph* by $g \underset{E_r}{\rightarrow} \{g_1, g_2\}$.

Decomposition to subgraphs uniquely defines subgraphs of g denoted by $g_1 \subseteq g$, $g_2 \subseteq g$ and satisfies $g_1 \cup_V g_2 \underset{iso}{=} g$

Definition 3.10. The *difference* between a graph g and a subgraph $g_1 \subseteq g$ is the subgraph $g_2 \subseteq g$ that is induced by $E_2 = E - E_1$. We use the notation $g - g_1$ to represent the subtraction of subgraph.

Note that both of g_1 and g_2 are subgraphs of g and there are vertices both are included in both g_1 and g_2 as shown in definition 3.9.

Definition 3.11. Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a query graph q, *induced subgraph isomorphism query* detects isomorphisms of all induced subgraphs from $g \in D$ to q. Similarly, *subgraph isomorphism query* detects all isomorphisms of all subgraphs from $g \in D$ to q.

We call a graph that is present in the graph database a *model graph*.

4. OUR APPROACH

In this section we describe briefly the database creation and the graph query search processes of *Network Method*. We detail our main contributions: A set of new subgraph query search processing algorithms that implement a fast subgraph search algorithm and on a modified DAG data structure. The improved DAG data structure is able to process both subgraph and induced subgraph queries.

4.1. The *Network Method*

Messmer et al.[Messmer and Bunke 2000] proposed a method to search graphs that are induced subgraphs of a query graph from a graph database. Central to the method is a DAG data structure that is used to store decomposed subgraphs of a graph database in a compact form. Messmer called this data structure a network so here we will refer to their method as the *Network Method*. A simplified schematic of the Network Method is shown in fig.2. This method finds all induced subgraph isomorphisms from graphs in the graph database to a query graph. We divide the *Network Method* into two parts.

In the first part a directed acyclic graph DAG is constructed by recursively decomposing model graphs and storing the decomposed graphs at the nodes of the DAG. In the DAG, the creation of the decomposed graph is directed by the original graph and on decomposition the directed graph is a *child* graph and the directing graph is the parent graph. Fig.2 (II) shows an example of two model graphs that are decomposed and stored in the nodes of a DAG to form a database in (II). A query is processed on this database in (III). In the process of decomposition, Model graphs g_1 and g_2 are decomposed into two graphs each, in such a way that each pair of decomposed graphs have as close to equal number of vertices as possible. Each of the decomposed pair of graphs is further decomposed into two graphs. This process continues until the decomposition results in singleton graphs. Before any graph is decomposed, it is checked for containment of an induced subgraph previously created by the decomposition process so far. If one is found, the graph is decomposed into the induced subgraph and a rest subgraph. In this way, an induced subgraph can be shared by multiple parents as shown in Fig.2 where g_1 is decomposed randomly into subgraph 6 and subgraph 8 by partitioning approximately an equal number of vertices. If g_2 contains an induced subgraph 8, g_2 is decomposed into graph 8 and graph 5 after the all decomposition of g_1 has completed.

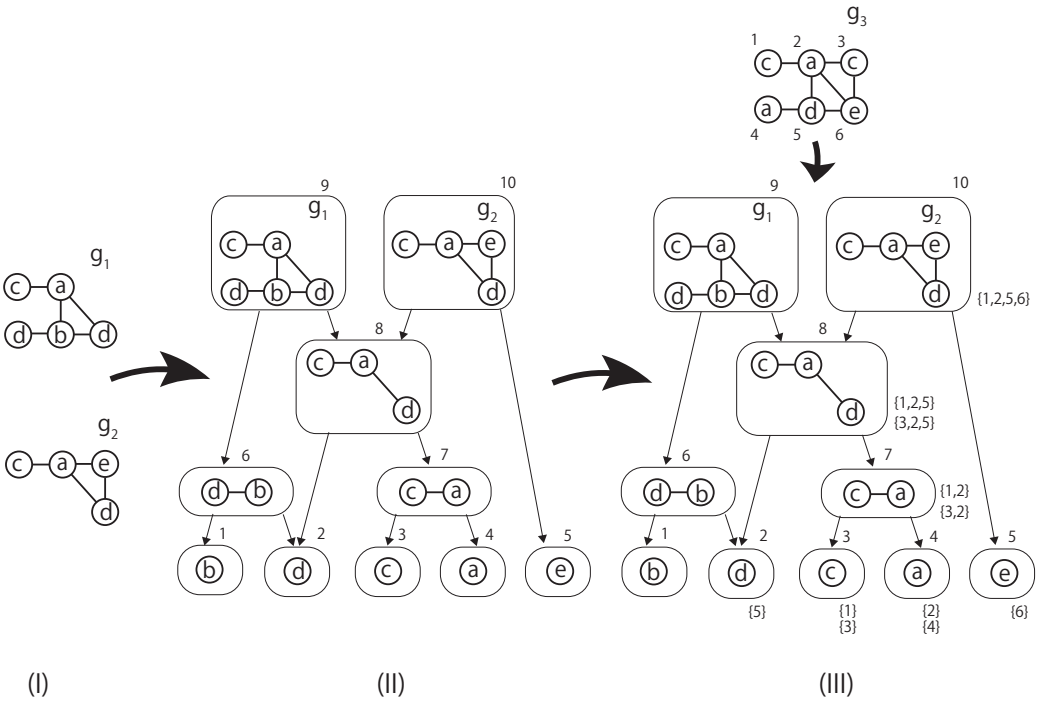

Fig. 2: Schematic of the *Network Method*: (I) Model Graphs g_1 and g_2, (II) The Graph Database:A DAG is Constructed using the model graphs g_1 and g_2 and their decomposed subgraphs. DAG $node$ 8 contains a subgraph common to model graphs g_1 and g_2 (III) Query Processing: Query Graph g_3 processed on the Graph Database and the matching query graph node IDs shown in braces

In the second part an induced subgraph isomorphism query is processed using the DAG constructed in first part. First, all induced subgraph isomorphisms from all singleton graphs in the DAG to query graph are detected. Based on the induced subgraph isomorphisms, induced subgraph isomorphisms of the respective parent graphs are calculated by combining the induced subgraph isomorphisms of its child graphs. This process repeats until finally, all induced subgraph isomorphisms of model graphs to the query are detected. The *Network Methood* makes use of the fact that if a graph in DAG is not an induced subgraph of query graph, its ancestors are also not induced subgraphs. In Fig.2 part(III), if *node 8* is not an induced subgraph of g_3, we know each of g_1 and g_2 are also not induced subgraph of $g_3$without performing any additional computation. Using this knowledge the time required to process the induced subgraph query is reduced by avoiding this unnecessary computation. The main advantages of the *Network Method* are:

(1) The DAG is constructed by decomposing graphs recursively which allows efficient solution of queries by divide and conquer.
(2) The induced subgraph isomorphisms of induced subgraphs discovered in the DAG that are common to multiple model graphs are computed only once and the result is shared.

	Vertex partition	Edge partition
Subgraph	sometimes infeasible	feasible
Induced subgraph	feasible	sometimes infeasible

Table I: Appropriate method for decompositions

4.2. Our Contributions

We have two main contributions. First, we propose a new search strategy for the graph database that can be used for both induced and non-induced subgraphs. We use this new algorithm to implement a framework for fast and scalable graph database creation and subgraph query processing. Secondly, we propose modifications to the DAG data structure to enable processing both subgraph query search and induced subgraph query search on the modified DAG data structure.

We implement these proposals in a framework where we create a graph database and evaluate the performance of subgraph isomorphism query search on the database. This is different from *Network Method* which can only processes induced subgraph isomorphism queries.

We observe an order of magnitude improvement in processing time over the state-of-the-art isomorphism test algorithm VF2. We also evaluate the performance of induced subgraph query search. Here we observe several orders of magnitude improvement in processing time over the *Network Method* and an order of magnitude improvement over VF2[Cordella et al. 2001] for random graphs.

We call the new framework for database creation and subgraph search *Fast Network Method*. The details of the proposed *Fast Network Method* are described below where we contrast it with the *Network Method*.

(1) A Fast Search Strategy for graph database Search
The search strategy works by minimizing the search space as follows.
(a) Localized subgraph Search
As shown in Fig.3 for each model graph in the database, the search process begins at the top, say g_1 and proceeds recursively in a depth first search(DFS) fashion until the state of the present node is in *unsolved* or *dead* state, or until the subgraph at the node is a singleton. When a node in *dead* state is encountered, the processing terminates and the current model graph assigned *dead* state. This is different from the *network method* where the search proceeds by first processing all leaf nodes in the DAG. It then proceeds to the second level in the DAG in the order of the number on top of the nodes till the model graphs at the top of the DAG are reached. Only after processing the nodes in the DAG is the isomorphism of the model graphs determined for those that are in *alive* state. As a consequence, processing of the whole DAG continues regardless of some of the subgraphs being determined to be in *dead* state.
By localizing subgraph search, the minimum processing required to determine a Model Graph to be in *dead* state is processing of a single singleton mode. This case is illustrated in Fig.3 where node 1 is found to be in *dead* state on processing.
Once a subgraph of the model graph is determined to be in *dead* state, further subgraph isomorphism processing terminates. This usually results in some subgraphs of the model graph remaining in unprocessed state which is a very desirable situation as unnecessary computation effort is avoided. In the case of Fig.3, in spite of node 2,3,4,and 14 being in *unsolved* state, model graph g_1 is conclusively determined to be in *dead* state, hence it is not isomorphic subgraph of the query graph, g_3.

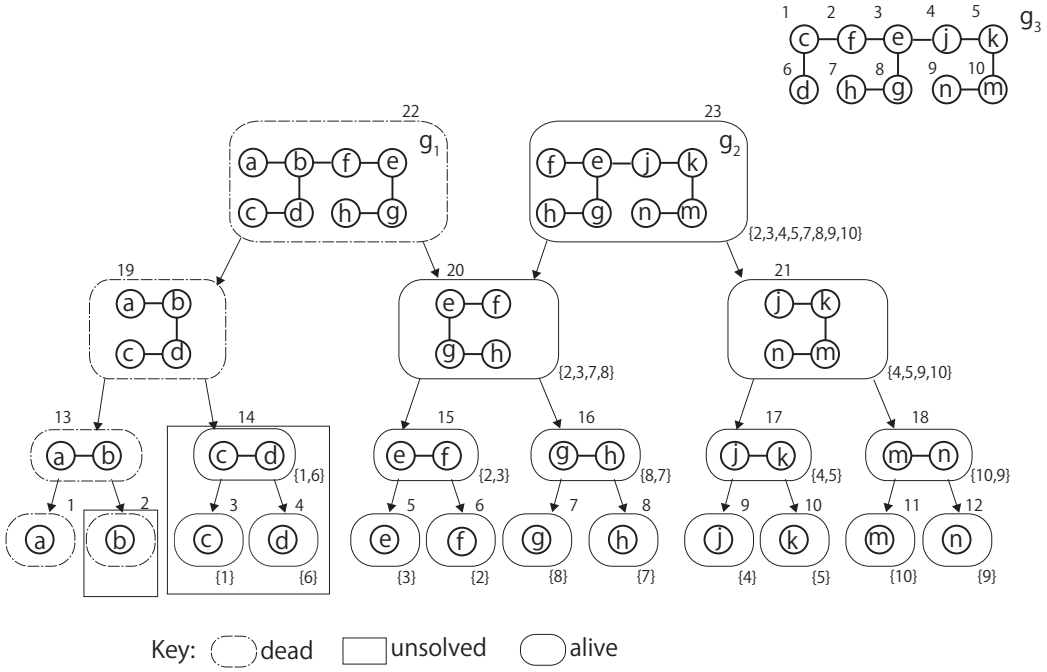

Fig. 3: The search strategy for the *Fast Network Method*: For a simplified database composed of model graphs g_1, g_2 and given a query graph g_3, each model graph is processed locally by starting at the top of the DAG in a depth first search (DFS) fashion. The search concludes when the first *dead* state is encountered. Specifically, we conclude that nodes 1,13,19, 22 (hence model graph g_1) are in *dead* state as soon as node 1 is processed. After processing g_1 and g_2 we have the result in spite of nodes 2,3,4, and 14 remaining in *unsolved* state. In contrast, the *Network Method* processes *all* the nodes in the DAG in the order of the numbers on the top right of the nodes before any output is delivered.

(b) Rapid propagation of subgraph search state
To process a query over a graph database, the query is performed for each model graph stored in the DAG. The processing of subgraph matchings for isomorphism for each model graph terminates as soon a subgraph isomorphism text fails or previously processed subgraph with a *dead* state is encountered, or the processing of an isomorphic model graph completes.
On failed termination or a *dead* state encounter, all the ancestors of the subgraph are set to *dead* state. As an example, in Fig.3 the label for the singleton subgraph, node 1 does not appear in any vertex in the query g_3, therefore the node 1 is not a match. We assign a *dead* state. Since we arrived at node 1 by repeated recursive calls from nodes 13, 19 and 22, we return the *dead* state to each of these nodes as the recursion terminates. This process of Rapid Propagating of the subgraph states up the *parent* graph, all the way to the Model graph that enables early discovery of the state of larger common subgraphs sooner. When the common subgraphs are not isomorphic to a query, large savings in computation are possible.

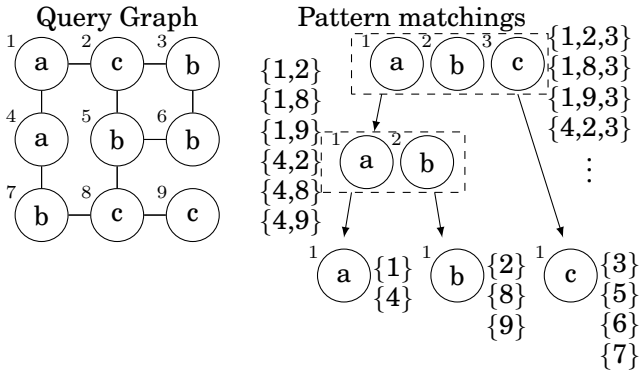

Fig. 4: An example of the explosion of number of matchings

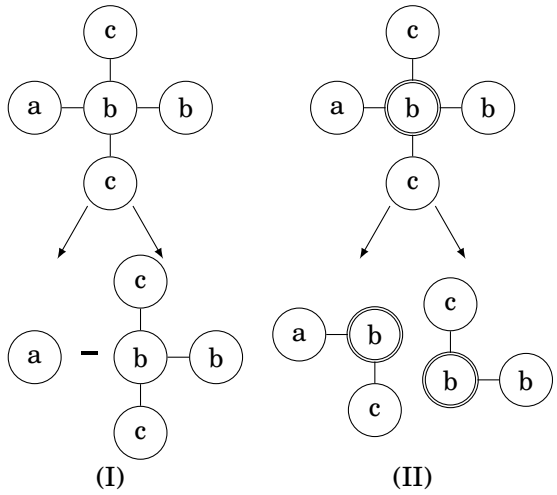

Fig. 5: Two ways to decompose a graph, (I) *Decomposition to induced subgraph* by partitioning the set vertices (interconnecting edge cutting). (II) *Decomposition to subgraph* by partitioning the set of edges (common vertex sharing).

By processing each model graph to completion, the *Fast Network Method* allows for progressively more processing efficiency as the state of larger common subgraphs is determined for each model subgraph. When the shared larger subgraph is non-isomorphic to a query, the potential search space is reduced significantly.

(2) Subgraph query processing using *Fast Network Method*
   (a) Data structure: Method of Decomposition
   In the processing of the DAG we have make two proposals.

The data structure used for subgraph storage in both the *Network Method* as well as the *Fast Network Method* consists of a directed acyclic graph(DAG). A simplified schematic of the DAG is show in Fig.2. A decomposed subgraph of the model graph is stored in each node of the DAG starting with the complete Model graph at the top. The nodes in the DAG represent decomposed subgraphs of the model graphs at various degrees of decomposition. To implement a DAG suitable for storage and search of subgraphs as well as induced subgraphs we need to provide procedures that are suitable for decomposition of the subgraphs. We define a decomposition method by partitioning edges, that is suited for processing subgraphs on the DAG. For subgraph query, we need to decompose graphs by partitioning edges and the rationale for this is as follows. This is due to the requirement that both children must be subgraphs of their parent graphs. We also face the limitation that we are unable to decompose a graph into an arbitrary subgraph and the rest graph by partitioning vertices so that rest graph becomes subgraph of its parent. Fig.7 shows an example where a decomposition of a graph into two subgraphs is not feasible by partitioning vertices. If we subtract s from g by partitioning vertices, s_{rest} cannot be represented as a graph as only an edge remains. We need to subtract the subgraphs by partitioning edges and then include the vertices on the edges as s'_{rest} shows. Hence for subgraph query processing we decompose the subgraphs by partitioning edges and include the corresponding vertices. The subtraction of subgraphs by partitioning edges (edge subtraction)is defined in *Definition 3.10*.

Fig.5 shows the difference between two methods of decomposition. In table.II we see the feasible approach to subgraph decomposition. In the *Network Method* only induced graphs are processed and the graphs are only decomposed by partitioning vertices (cutting edges). In order to process subgraph query, we have implemented *decomposition to subgraph* by partitioning edges (shared vertices). We use the appropriate decomposition for both decompositions and also reform corresponding algorithms.

(b) Data Structure: Connected Subgraphs

We formulate the decomposition algorithm with the restriction that only a connected graphs can be processed and it must be decomposed to strictly result in only two connected subgraphs. When decomposition results in one or both graph disconnected, we post-process the resulting subgraphs by redistributing one or both components between the children at random until two connected graphs result. Decomposing a graph at random and without any restriction on the resulting subgraphs risks that the result may be multiple disconnected subgraphs. Hence, decomposition possibly creates multiple connected graphs composed of many disconnected graph fragments. This in turn may result in number of matchings from a large number of small graphs to the query graph to become intractable. Fig.4 shows an extreme example of the rapid increase of the matchings that results when a graph is decomposed to singletons. The number of matchings correlate with processing time as well as memory requirements, hence the need to restrict number of required matchings.

(3) Induced Subgraph query processing using the *Fast Network Method*

We implement the all the contributions that are mentioned above in the induced subgraph query processing as well. The detailed algorithms are shown in Section 5. As the *Network Method* is able to process induced subgraphs, we perform a comparison of the processing times and show the results is section 6.

(a) Data Structure: Method of Decomposition

For induced subgraph query, we have to decompose the graph by partitioning vertices to form two induced subgraphs and store the information about the

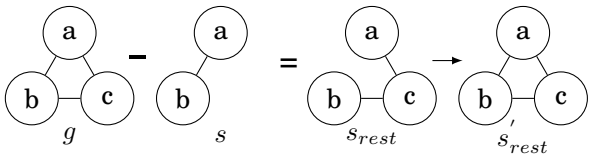

Fig. 6: A case where decomposition of induced subgraph by partitioning edges (shared vertices) is not feasible

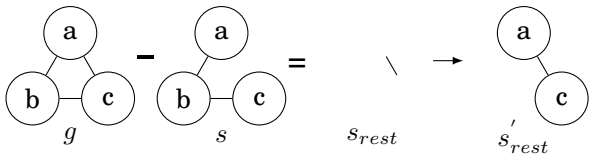

Fig. 7: A case where decomposition of subgraph by partitioning vertices (cutting edges) is not feasible

interconnecting cut edges. Both the resulting children must be induced subgraphs of their parent to enable processing of induced subgraph queries in a divide and conquer fashion. However, Decomposing an arbitrary graph into its induced subgraph and rest graph by shared vertices such that both children are induced subgraphs of the parent is not assured.

In Fig.6 we see an example where the decomposition of a graph into two induced subgraphs is infeasible by partitioning edges (sharing vertices). If we decompose g by partitioning edges, the rest graph is s_{rest}. But s_{rest} must be an induced subgraph of g, so the edge is complemented as s'_{rest} and s'_{rest} is equal to g. This indicates that we cannot decompose a graph g any further as g is a complete graph. The subtraction of induced subgraph by partitioning vertices (vertex subtraction) is defined in *Definition 3.7*.

Fig.5 shows the difference between two methods of decomposition. Table.II Here we can see the feasible approach to induced subgraph decomposition.

(b) Data Structure: Connected Graphs

We formulate the decomposition algorithm with the restriction that only a connected induced graphs can be processed and it must be decomposed to strictly result in only two connected induced subgraphs. When decomposition results in one or both induced graph being disconnected, we post-process the resulting subgraphs by redistributing one or both components between the children at random until two connected induced graphs result. The *Network Method* however decomposes an induced graph at random and without any restriction on the resulting subgraphs that maybe disconnected. As a result, decomposition may create multiple connected graphs composed of many disconnected graph fragments. This in turn may result in the number of matchings from a large number of small graphs to the query graph to become intractable. As an extreme example, Fig.4 shows the rapid increase of the matchings that results when a graph is decomposed to singletons. The number of matchings correlate with processing time as well as memory requirements, hence the need to restrict number of required matchings.

Author	Messmer et.al.	Proposed
Algorithm	Network Method	Fast Network Method
Data Structure	DAG (Induced Subgraph)	DAG (Subgraph & Induced Subgraph)
Method of graph Decomposition	Partition to Subgraph	Partition to Subgraph Partition to Induced Subgraph
Processing Optimization	No Restriction	Connected graph input Connected graph Output
Search Strategy	Whole DAG bottom-up search	Local Depth First Search(LDFS) of Model Graphs in DAG
Search graph type	Induced Graph	Subgraph and Induced Graph

Table II: Comparison of Network Method and Fast Network Method

5. PROPOSED ALGORITHMS

In this section, we present details for the proposed *New Network algorithm* and related algorithms. There are two main contributions: a new faster search strategy and extension to processing of subgraph queries. Further, we also describe in detail the improved algorithms for processing of induced subgraph query.
As an overview, the algorithms solve two problems, database creation and query processing. Accordingly we divide the discussion into them into two corresponding parts. The first part is concerned with the construction of a dynamic acyclic graph DAG to store the model graphs. The model graphs are stored in a network with their decomposed subgraphs and the related links and state information. Specifically, we construct a DAG consisting of a set of 5-Tuples, each of which stores information about a graph, its state, its parents, its children and the Edges connecting the pair of child graphs. Creation of the DAG is performed by recursively decomposing the Model graphs and creating a 5-tuple for each resulting graph to store the information. The 5-tuples can be considered as nodes in the DAG as they are connected by a directed edge from parent to child.
The second group of Algorithms concerns the process of detecting subgraph isomorphisms using DAG constructed in previous part. The DAG is the central data structure for this work and is considered a global variant in all the algorithms.

5.1. DAG definition

First we make a formal definition to help us write the algorithms for the DAG more concisely.
Let $G = \{g_1, \ldots, g_n\}$ be a set of model graph.

Definition 5.1. A ***DAG*** $D(G)$ is a finite set of 5-tuple (g, s, P, C, E) constructed by decomposition of a set of model graphs G where g is a graph, s is one of $\{unsolved, dead, alive\}$, P is a set of one or more parent graphs $\{p^{'}, p^{''} \ldots\}$, C is a set of two child graphs $\{c^{'}, c^{''}\}$ of graph g, and E is a set of edges connecting the child graphs c' and $c^{''}$. The DAG has to satisfy the following conditions.
(1) $c^{'} \subset g$ and $c^{''} \subset g$ for each child $c^{'}, c^{''}$ of graph g
(2) $g \subset p^{'}$, $g \subset p^{''} \ldots$, for each parent in p of graph g
(3) $g = c^{'} \cup_E c^{''}$
(4) For each g_i in G there exists a $5 - tuple$ $(g, s, P, C, E) \in D(G)$
(5) For each $5-tuple$ $(g, s, P, C, E) \in D(G)$ there exists no other $5-tuple$ $(g, s, P, C, E) \in D(G)$ with $g = g_1$.

(6) For each $5-tuple$ $(g, s, P, C, E) \in D(G)$
   a. if $c^{'}, c^{''} \in C$ consist of graphs with more than one vertex, then there exists a corresponding pair of $5-tuple$
   b. if $c^{'}, c^{''} \in C$ consist of one vertex then there exists no corresponding $5-tuple$

Definition 5.2. A graph g is disconnected if its vertex set V can be partitioned into two nonempty, disjoint subsets V_1 and V_2 such that there exists no edge in g whose one end is in subset V_1 and the other end is in subset V_2

Definition 5.3. The size of a graph g is defined as $|V|+|E|$, the sum of the number of Vertices and Edges in the graph, and is denoted by $|g|$.

Definition 5.4. The *minimal component* of a disconnected graph g is defined as a disjoint component graph of a disconnected graph such that no other disjoint component has fewer vertices. The minimal component of disconnected graph g is denoted by g_{min}.

5.2. Expansion to Subgraph Isomorphism Query with reduced search space

The method of graph decomposition used in the *Network Algorithm* is restricted in its support to induced subgraph isomorphism query processing only. However, in practice, subgraph isomorphism query is more likely to suit real world problems therefore we reformulate the *Network Algorithm* to process subgraph isomorphism query. In this subsection, we describe the details of the expansion of the *Network Algorithm* to allow subgraph query processing and also propose some addition improvements. Some of our improvements applied to subgraph query are also applicable to induced subgraph queries which is addressed in the next subsection. For these algorithms, we use parenthesis such as (induced) subgraph to separate the two cases. We first discuss the subgraph isomorphism so ignore the inside of the parentheses throughout this subsection.

5.2.1. DAG Construction. The DAG is the main data structure for the database construction and query processing. In this section we outline the steps required to construct the DAG as well as the use of the DAG to process queries. First the algorithm for the construction of the DAG is show in Algorithm 1 below.

ALGORITHM 1: createDAG(G)

Input: Model Graphs $G = \{g_1, g_2, \ldots, g_m\}$
Output: $D(G) = \{(g, s, P, C, E), \ldots, (g_n, s_n, P_n, C_n, E_n)\}$

```
1: Initialize DAG, D := ∅
2: for i = 1 to m do
3:    D ←createTuples(g_i, D)
4: end for
5: return D(G)
```

CreateTuples(Algorithm2) outlines the dynamic tuple insertion algorithm for DAG. In order to construct DAG with model graphs , we first decompose the model graphs. Then we create and insert the corresponding 5-tuples and the into the DAG to form a graph database. Model graphs are inserted in the DAG sequentially.
In constructing the DAG, as a rule, each graph in a tuple directs two (induced) subgraphs. The exception being that g will have no children if g is a singleton graph. The information about the shared vertices in the children and the mappings between a parent and children is stored with a parent-child relationship in the 5-tuple.

ALGORITHM 2: CreateTuples(g, D)

Input: Connected, non-singleton Graph g, Current DAG D

Output: DAG with current tuples and new tuples from graph g added, $D^{'}$

```
1: Isomorphic graph set Z ← ∅
2: graph z ← ∅
3: Default node state s ← unsolved
4: Set of Parent Graphs, P ← ∅
5: Local DAG initialized with current tuples D' ← D
6: for each Tuple {(z, s_z, P_z, {c', c''}, E)} ∈ D' where vertex count |z| ≤ |g| do
7:    if graph z is ⊆_(induced) g then
8:       Add graph z to Z
9:    end if
10: end for
11: while Z ≠ ∅ do
12:    if z = g then
13:       P_z ← P_g
14:       D' ← D' ∪ {(z, s, P_z, {∅}, ∅)}
15:       return D'
16:    else
17:       for each graph z in Z do
18:          z_rest ← (g −_(induced) z)
19:          if z_rest is connected then
20:             D' ← D' ∪ {(g, s_g, P_g, {∅}, ∅)}
21:             D' ← D' ∪ {(z_rest, s_{z_rest}, {g}, {∅}, ∅)}
22:             CreateTuples(z_rest)
23:             return D'
24:          end if
25:       end for
26:    end if
27: end while
28: (g, s_g, ∅, {z, z_rest}, E) ← RandomPartition(g, s_g, ∅, {∅}, ∅)
29: D' ← D' ∪ {(g, s_g, ∅, {z, z_rest}, E)}
30: D' ← D' ∪ {(z, s_z, {g}, {∅}, ∅)}
31: D' ← D' ∪ {(z_rest, s_{z_rest}, {g}, {∅}, ∅)}
32: CreateTuples(z)
33: CreateTuples(z_rest)
34: return D'
```

First, the algorithm searches all (induced) subgraphs of g_i in DAG(line 6-10). For this (induced) subgraph search, SubgraphQuery (Algorithm 5.3.2) explained in the latter section is used. The discovered (induced) subgraphs are inserted to a queue Z.

If we find that a graph z in Z is isomorphic to g_i, we replace the graph by setting a directed edge from the parent of g_i to z (line 12-15). The existence of graph isomorphism is easily ensured by checking whether $E(g_i)$ is equals to $E(z)$ or not. In most cases, a (induced) subgraph z of g_i will be discovered. In that case, the algorithm calculates $z_{rest} = g_i -_{(induced)} z$ and g_i is decomposed into the z and the rest graph z_{rest}. If z_{rest} is connected, z_{rest} is inserted to DAG recursively(line 17-25). There is the case that z_{rest} is disconnected, which is a violation of the requirement that a connected graph is decomposed into two connected graphs. When faced with this vi-

ALGORITHM 3: ConnectGraph($g, s, P, \{c^{'}, c^{''}\}, E$)

Input: graph with disconnected childgraphs ($g, s, P, \{c^{'}, c^{''}\}, E$)
Output: graph with connected childgraphs ($g, s, P, \{c^{'}, c^{''}\}, E$)

1: $c^{'} \leftarrow \emptyset$
2: $c^{''} \leftarrow \emptyset$
3: **if** graph g has a single edge e(v_1,v_2) **then**
4: Insert v_1 to $c^{'}$, v_2 to $c^{''}$
5: **else**
6: **repeat**
7: **if** $c^{'}$ **and** $c^{''}$ are disconnected **then**
8: **if** $|c^{'}| > |c^{''}|$ **then**
9: $c^{''} \leftarrow c^{''} \cup c^{'}_{min}$
10: **else**
11: $c^{'} \leftarrow c^{'} \cup c^{''}_{min}$
12: **end if**
13: **else**
14: **if** $c^{'}$ is disconnected **then**
15: $c^{''} \leftarrow c^{''} \cup c^{'}_{min}$
16: **else**
17: $c^{'} \leftarrow c^{'} \cup c^{''}_{min}$
18: **end if**
19: **end if**
20: **until** $c^{'}$ **and** $c^{''}$ are connected
21: **end if**
22: **return** $(g, s, P, \{c^{'}, c^{''}\}, E)$

ALGORITHM 4: RandomPartition($g, s, P, \{c^{'}, c^{''}\}, E$)

Input: graph g
Output: $(g, s, P, \{c^{'}, c^{''}\}, E)$ is a 5-tuple consisting of graph g, a state s children $\{c^{'}, c^{''}\}$ and a set of edges between the children E

1: **if** g has a single edge $e = (v_1, v_2)$ **then**
2: $c^{'} \leftarrow \{v_1\}$
3: $c^{''} \leftarrow \{v_2\}$
4: **else**
5: $(g, s, P, \{c^{'}, c^{''}\}, E) \leftarrow$ RandomPartitionInduced($g, s, P, \{\emptyset\}, E$)
6: **end if**
7: **return** $(g, s, P, \{c^{'}, c^{''}\}, E)$

olation the algorithm abandons the result and tests the subtractions of all (induced) subgraph isomorphisms of all (induced) subgraphs until z_{rest} becomes connected.
The order in which the (induced) subgraph are tested for subtraction relies ordering of the graph in Z. We know that The larger a graph is, the more costly it is to compute the (induced) subgraph isomorphisms therefore we sort Z to make larger graphs precede to ensure larger subgraphs are shared.
If no (induced) subgraph is discovered or no (induced) subgraph can make z_{rest} connected, g_i is decomposed by the RandomPartition (Algorithm4). The decomposed (induced) subgraphs are inserted to DAG recursively(line 38-33).
The *Network Algorithm* decomposes a graph by distributing vertices randomly and creating subgraphs which are induced by the distributed vertices. This decomposi-

ALGORITHM 5: New Network Algorithm, NNA(D, q)

Input: Model graphs in DAG, $D(G) = \{(g_1, s_1, P_1, \{c_1^{'}, c_1^{''}\}, E), \ldots, (g_n, s_n, P_n, \{c_n^{'}, c_n^{''}\}, E_n)\}$, Query q
and the set of all subgraphs in DAG, $G_{all} = \cup_{i=1}^{n}\{g_1, c_1^{'}, c_1^{''}, g_2, c_2^{'}, c_2^{''} \ldots\}$
Output: Z, the set of all subgraph isomorphisms in Model graphs in D to query q

$F \leftarrow \emptyset$
$Z \leftarrow \emptyset$
for each subgraph in G_{all} **do**
  $s \leftarrow unsolved$
end for
for each $(g, s, P, \{c^{'}, c^{''}\}) \in D$ **do**
  $F \leftarrow$SubgraphQuery$(g, s, P, \{c^{'}, c^{''}\}, q)$
  if $F \neq \emptyset$ **then**
    $Z \leftarrow Z \cup \{F\}$
  end if
end for
return Z

ALGORITHM 6: SubgraphQuery$(g, s, P, \{c^{'}, c^{''}\}, E, q)$

Input: Model graph 5-Tuple $(g, s, P, \{c^{'}, c^{''}\}, E)$, query q
Output: Induced Subgraph Isomorphisms, F

if s = $unsolved$ **then**
  $F \leftarrow \emptyset$
  if g is a singleton **then**
    $F \leftarrow$ AssignVertex(g,q)
  else
    if $s_{c'} = dead$ **or** $s_{c''} = dead$ **then**
      $s \leftarrow dead$
      return F
    else if SubgraphQuery$(c^{'}, s_{c'}, P_{c'}, \{c_1^{'}, c_1^{''}\}, E_1, q) = \emptyset$
    or SubgraphQuery$(c^{''}, s_{c''}, P_{c''}, \{c_2^{'}, c_2^{''}\}, E_2, q) = \emptyset$ **then**
      $s \leftarrow dead$
      return F
    else
      $F \leftarrow$ CombineInduced$(g, s, P, \{c^{'}, c^{''}\}, E, q)$
    end if
  end if
  if $F \neq \emptyset$ **then**
    $s \leftarrow alive$
  else
    $s \leftarrow dead$
    return F
  end if
else
  return F
end if

tion may create disconnected subgraphs composed of many tiny components. Conceivably, the number of matchings resulting from a large number of such disconnected graphs to a query graph can easily skyrocket hence need to make sure that the decomposed subgraphs are connected to avoid this problem. Effectively, the *Network Algorithm* decomposes a graph by partitioning vertices for induced subgraph query

We now focus on processing of subgraphs which are not induced. In this case, the subtraction of an arbitrary subgraph from a graph requires the decomposition of a graph by partitioning edges.

RandomPartition (Algorithm4) decomposes a connected graph g into two connected subgraphs. First, the algorithm distributes edges at random to form two graphs. If at least one subgraph is disconnected, the two subgraphs transfer their minimal components to the opposite side. This movement of components likely results in connected graphs because the parent graph g is connected. The transfer of components is repeated until both subgraphs become connected.

The limit for decomposing a graph comes when there is a single edge, which we cannot decompose further by partitioning vertices. Hence, RandomPartition algorithm cannot process a singleton graph or a disconnected graph which includes one or more isolated vertices. To overcome this limitation, we decompose a single edge by partitioning vertices.

5.2.2. Query Processing. There are two types of query processing to the graph database; induced subgraph query and the subgraph query. In this section we outline the algorithms and discuss both types of processing. The *New Network Algorithm* (Algorithm 5) outlines the procedure for the (induced) subgraph isomorphism query processing. Using the SubgraphQuery (Algorithm 5.3.2) we can calculate the (induced) subgraph isomorphism of any graph stored in a node in DAG to a query graph q. We evaluate (induced) subgraph isomorphism query by processing SubgraphQuery with all model graphs as input.

In the procedure for the (induced) subgraph search, all nodes in DAG are set to one of three states using tags, $alive$, $dead$ or $unsolved$. Those tags indicate three states:(a) already matched with the query graph and (induced) subgraph isomorphisms exists, (b) already matched and (induced) subgraph isomorphism does not exist, (c) not been matched yet respectively. Initially, all nodes in DAG are tagged $unsolved$.

SubgraphQuery processes the decomposed input graph in the DAG in a recursive manner. If the input graph g is already tagged $alive$ or $dead$, there is nothing to do and the algorithm only returns subgraph isomorphism F already calculated.

If g is a singleton graph there are no child graphs so SubgraphQuery calls AssignVertex(Algorithm7) to simply search the occurrences of the label of the single vertex in a query graph q.

If g marked $unsolved$, we check (induced) subgraph isomorphism of its children by calling SubgraphQuery recursively. We exploit the knowledge that if a child is not a (induced) subgraph of a query graph, its parent cannot be a (induced) subgraph of the query graph. First, in order to avoid unnecessary isomorphism computation we check the tags of children of g before we call SubgraphQuery. Next, we call SubgraphQuery for g's children. If at least one child tagged $dead$ or returns an empty set, we are can tag g as $dead$. Only in the case that both of the children are tagged with or return the state $alive$ do we have to calculate the (induced) subgraph isomorphisms from g to q.

To process the query q, we can use already computed child graph's (induced) subgraph isomorphism to q to reduce the computational effort of calculating (induced)

subgraph isomorphism from g to q. For this purpose, a procedure for combining of induced subgraph isomorphisms is required.
Combine (Algorithm8) is the procedure for the combining subgraph isomorphisms. When Combine is called with g as input, the subgraph isomorphisms F_1, F_2 from its children $c^{'}, c^{''}$ to q are supposed to be already computed. In the case that g is connected, two conditions must be satisfied to combine two functions, $f_1 \in F_1, f_2 \in F_2$. First, it we must ensure that each corresponding common vertices is mapped correctly onto same vertex (line 11). Second, the image of non-common vertices of f_1 and f_2 must be disjoint (line 12). Satisfaction of these conditions ensures that the combining of f_1 and f_2 is injective. If g is disconnected, we only need to check one condition that the images of f_1 and f_2 are disjoint (line 20). We define the combination of (induced) subgraph isomorphisms as follows

$$f(v) = \begin{cases} f_1(\bar{p_1}(v)) & \bar{p_1}(v) \in V_{c'} \\ f_2(\bar{p_2}(v)) & \bar{p_2}(v) \in V_{c''} \end{cases} \quad (1)$$

Where $\bar{p_1}$ and $\bar{p_2}$ are functions from parent graph g to childgraphs $c^{'}$ and $c^{''}$ respectively. The Combine algorithm checks all pairs of functions f_1 and f_2 and when a pair is found that satisfies these conditions, the algorithm evaluates and returns it. However this algorithm is limited to graphs decomposable by partitioning edges such that if a graph consisting of a single edge is input, the Combine algorithm has to call Combineinduced (Algorithm10) which is able to process graphs decomposable by partitioning vertices..

ALGORITHM 7: AssignVertex(g, q)

Input: Singleton graph $g = (V, E, L^v, L^e, \mu, \nu)$, Query graph $q = (V_q, E_q, L_q^v, L_q^e, \mu_q, \nu_q)$
Output: Subgraph isomorphisms F

```
1: F ← ∅
2: let v ∈ V
3: for each v_q ∈ V_q do
4:     if μ(v) = μ_q(v_q) then
5:         add f(v) = v_q to F
6:     end if
7: end for
8: return F
```

5.3. Induced Subgraph Query

The improvement algorithms proposed in previous subsection are also efficient for induced subgraph query. In this subsection, we explain the detail algorithms for our induced subgraph query processing. Many of algorithms of induced subgraph query are same as that of subgraph query, therefore when we reference the previous subsection, take the insides of the parenthesises into consideration to indicate the induced versions.

5.3.1. DAG Construction. The procedure for the construction of DAG and dynamic insertion algorithm are also almost identical to the case of subgraph query. We now explain the differences. First, a parent-child relationship must store the information about the edges between partitioned vertices instead of the vertices common graphs. This is due of the difference in the manner of decomposition of the graphs. Second, the existence of a graph isomorphism is assured by checking whether input graph $V(g_i)$ is equal to a graph $V(s)$ in the DAG or not.

ALGORITHM 8: Combine$(g, s, P, \{c^{'}, c^{''}\}, E, q)$

Input: Graph 5-Tuple $(g, s, P, \{c^{'}, c^{''}\}, E)$, Query graph q
Output: Subgraph isomorphisms F

if g is composed of a single edge **then**
  $F \leftarrow$ CombineInduced$(g, s, P, \{c^{'}, c^{''}\}, E, q)$
else
  $F_1 \leftarrow$ Combine$(c^{'}, s_{c'}, P_{c'}, \{c_1^{'}, c_1^{''}\}, E_1, q)$
  $F_2 \leftarrow$ Combine$(c^{''}, s_{c''}, P_{c''}, \{c_2^{'}, c_2^{''}\}, E_2, q)$
  $F \leftarrow \emptyset$
  if g is connected **then**
    $V_c = V_g - (V_1 \cup V_2)$
    for each pair (f_1, f_2) in (F_1, F_2) **do**
      if $f_1(\bar{p_1}(V_c)) = f_2(\bar{p_2}(V_c))$ **then**
        if $f_1(V_{c'}) \cap f_2(V_{c''}) = f_1(\bar{p_1}(V_c))$ **then**
          $g \leftarrow (c^{'} \cup_E c^{''})$
          $V_g \leftarrow f(V_{c'} \cup V_{c''})$
          $F \leftarrow F \cup \{f\}$
        end if
      end if
    end for
  else
    for each $f_1 \in F_1, f_2 \in F_2$ **do**
      if $f_1(V_{c'}) \cap f_2(V_{c''}) = \emptyset$ **then**
        $g \leftarrow (c^{'} \cup_E c^{''})$
        $V_g \leftarrow f(V_{c'} \cup V_{c''})$
        $F \leftarrow F \cup \{f\}$
      end if
    end for
  end if
end if
return F

In order to decompose an arbitrary graph into two smaller induced subgraphs we have to decompose a graph by partitioning vertices hence we define a procedure for decomposing a randomly connected graph in this manner.
RandomPartitionInduced (Algorithm9) decomposes a connected graph into two connected induced subgraphs. First, the algorithm initializes two induced subgraphs by distributed vertices and the edges that the vertices induce. Then, two subgraphs transfer their minimum components until they become connected, a process similar to the one in the algorithm for subgraph query.
The decomposition of a disconnected graph is identical to the one for subgraph query.

5.3.2. Query Processing. The search processing algorithms for subgraphs, *NNA* (Algorithm5), SubgraphQuery (Algorithm5.3.2) and AssignVertex(Algorithm7) are also applicable to induced subgraph query. However, due to the difference in the method of decomposition, we need a new procedure for the combining the induced subgraph isomorphisms. CombineInduced(Algorithm10) outlines that procedure. When CombineInduced is called with graph g as input, the sets of induced subgraph isomorphisms F_1, F_2 from its children $c^{'}, c^{''}$ to the query graph must have been already calculated, as part recursive calling of . There are two conditions that must be satisfied to CombineInduced two functions $f_1 \in F_1 and f_2 \in F_2$. First, there

ALGORITHM 9: RandomPartitionInduced

Input: 5-Tuple of Graph $(g, s, P, \{c^{'}, c^{''}\}, E)$
Output: 5-Tuple of a pair of graphs $(c^{'}, s, P_1, \{c_1^{'}, c_1^{''}\}, E_1)$, $(c^{''}, s, P_2, \{c_2^{'}, c_2^{''}\}, E_2)$

1: $c^{'} \leftarrow \emptyset$
2: $c^{''} \leftarrow \emptyset$
3: $s_1 \leftarrow s$
4: $s_2 \leftarrow s$
5: **while** $|V_{c'}| \leq \lfloor |V_g|/2 \rfloor$ **do**
6: $\quad v \leftarrow$ random vertex of V_g
7: $\quad V_{c'} \leftarrow V_{c'} \cup v$
8: $\quad V_g \leftarrow V_g - v$
9: **end while**
10: $V_{c''} \leftarrow V_g$
11: $E_{c'} \leftarrow$ Edges induced in $V_{c'}$ by g
12: $E_{c''} \leftarrow$ Edges induced in $V_{c''}$ by g
13: $c^{'} \leftarrow \{V_{c'}, E_{c'}\}$
14: $c^{''} \leftarrow \{V_{c''}, E_{c''}\}$
15: **repeat**
16: $\quad$ **if** $c^{'}$ **and** $c^{''}$ are disconnected **then**
17: $\quad\quad$ **if** $|c^{'}| > |c^{''}|$ **then**
18: $\quad\quad\quad c^{''} \leftarrow c^{''} \cup c^{'}_{min}$
19: $\quad\quad$ **else**
20: $\quad\quad\quad c^{'} \leftarrow c^{'} \cup c^{''}_{min}$
21: $\quad\quad$ **end if**
22: $\quad$ **else**
23: $\quad\quad$ **if** $c^{'}$ is disconnected **then**
24: $\quad\quad\quad c^{''} \leftarrow c^{''} \cup c^{'}_{min}$
25: $\quad\quad$ **else**
26: $\quad\quad\quad c^{'} \leftarrow c^{'} \cup c^{''}_{min}$
27: $\quad\quad$ **end if**
28: $\quad$ **end if**
29: **until** $c^{'}$ **and** $c^{''}$ are connected
30: $E_1 \leftarrow E - (E_{c'} \cup E_{c''})$
31: $E_2 \leftarrow E - (E_{c'} \cup E_{c''})$
32: $P_1 \leftarrow \{g\}$
33: $P_2 \leftarrow \{g\}$
34: **return** $((c^{'}, s, P, \{c_1^{'}, c_1^{''}\}, E_1), (c^{''}, s, P, \{c_2^{'}, c_2^{''}\}, E_2))$

must exist edges in the query graph correspond to cut edges and vice versa(line 7). Second, we must ensure that f_1 and f_2 are disjoint(line 8), otherwise the combination of f_1 and f_2 will not be injective function. CombineInduced algorithm checks all pairs of the two functions and returns the sets of its unions that satisfy these conditions.

5.4. Network Algorithm with Graph Pruning

The *Network Algorithm* cannot process a local query efficiently (i.e. In the case that you want to know subgraph isomorphisms for only some graphs in a graph database to a query graph, the *Network Algorithm* calculates subgraph isomorphisms for all the nodes in DAG). In contrast our recursive formulation of the *Network Algorithm* only checks the relevant descendants of selected nodes and this allows us to solve local queries more efficiently.

ALGORITHM 10: CombineInduced

Input: Graph 5-Tuple $(g, s, P, \{c^{'}, c^{''}\}, E)$, Query graph q
Output: Subgraph isomorphisms F

1: $F_1 \leftarrow$CombineInduced$(c^{'}, s_1, P_1, \{c_1^{'}, c_1^{''}\}, E_1, q)$
2: $F_2 \leftarrow$CombineInduced$(c^{''}, s_2, P_2, \{c_2^{'}, c_2^{''}\}, E_2, q)$
3: $F \leftarrow \emptyset$
4: $E \leftarrow E_g - (E_1 \cup E_2)$
5: **for** each pair (f_1, f_2) in (F_1, F_2) **do**
6: **if** for each $e = (v_1, v_2) \in E$ there exists $e_q = (f_1(v_1), f_2(v_2)) \in E_q$ with $\nu(e) = \nu_q(e_q)$
7: **and** for each $e_q = (u_1, u_2) \in E_q$ such that $u_1 \in f_1(V_{c'}), u_2 \in f_2(V_{c''})$ there exists $e = (f_1^{-1}(u_1), f_2^{-1}(u_2)) \in E$ with $\nu_q(e_q) = \nu(e)$ **then**
8: **if** $f_1(V_{c'}) \cap f_2(V_{c''}) = \emptyset$ **then**
9: $g \leftarrow (c^{'} \cup_E c^{''})$
10: $V_g \leftarrow f(V_{c'} \cup V_{c''})$
11: $F \leftarrow F \cup \{f\}$
12: **end if**
13: **end if**
14: **end for**
15: **return** F

One result of the support of local query is that we can make use of some graph pruning techniques. Let us consider a simple example *label check*. If a graph database stores the numeric labels of attributed graphs, it may be easily determined when they have no subgraph isomorphisms to a query graph just by comparing the attributes of the graph with those of the query graph. In this way, we can efficiently reduce the candidates graphs for the answer. Consequently, by pruning graphs in advance, we can significantly reduce the related procesing time.

6. PERFORMANCE EVALUATION

In this section, we evaluate our algorithms on two datasets, a set of synthetic graphs and a set of graphs modeling active ingredients in antiviral drugs for the treatment of AIDS. We call the former the synthetic graph dataset and the later the real graph dataset. We perform two sets of tests; In the first set, we evaluate our algorithms on the retrieval of induced subgraph to query and In the second set, we evaluate retrieval of non induced sugraph to query. To test the efficacy of our algorithm with regard to induced subgraph query, we evaluate the proposed algorithm in comparison with Messmer's Network Algorithm and a sequential SCAN using the state-of-the-art subgraph isomorphism detection algorithm VF2[Cordella et al. 2001]. We also evaluate our algorithm with regard to non-induced subgraph query, where we compare proposed algorithm with the sequential scan only. All the algorithms were implemented using the C++ programming language and run on a Intel Core i-3 CPU 3.09 GHZ, 16 Gbyte memory, Personal Computer running Windows 7 Professional Operating system.

6.1. Graph Datasets

We evaluate our algorithms by processing the retrieval of descriptors from the compounds dataset. We use AIDS Antiviral Screen dataset to provide a real graph dataset. This dataset contains around 43,000 chemical compounds and is available publicly from NCI [1]. We denote this dataset as "AIDS" in our experiment. The

[1]National Cancer Institute http://dtp.nci.nih.gov/

graphs of AIDS have an average number of 25 vertices and 27 edges, a maximum of 438 vertices and 441 edges, 63 distinct vertex labels and 3 distinct edge labels.
In order to evaluate our algorithms over a larger database to test scalability, we use synthetic large dataset. The graph generator is configured to emit only connected graphs thathave an edge probability of 50 percent. Except where explicitly stated, 10 distinct vertex labels and 10 distinct.

6.2. Chemical Descriptor Search

In chemistry, substructures of compounds that imply a chemical, physical property of the compound are called as descriptor. Fast retrievals of descriptors from compounds aids research of compounds. We evaluate our algorihm by processing the retrievals of descriptors. We builded a model graph database D by extracting graph database W composed of 10,000 compounds whose size are less than 40 from AIDS and applying frequent graph mining to W. We set minimum support as 5 percent. D is composed of 18930 distictive frequent subgraphs.
We vary average size of query graphs and evaluate query processing time. For each plot, 100 query graphs are extracted from W. Query processing time is time to process the all query graphs and the time to construct the DAG is excluded. In the results we compare the processing time of two methods with our proposed *Fast Network Algorithm*. These methods are:
The *Network algorithm*, in which the use of DAG data structure to store the graphs was pioneered. This method however was only implemented for induced subgraph queries, so we are only able to perform comparisons for half of the experiments.
The state-of-the-art one-on-one graph query algorithm *VF2*. This algorithm tests a graph for subgraph isomorphism or induced subgraph isomorphism to another single graph only. This is repeated sequentially for a large graph database. The solution is a binary true or false answer. It cannot be used to find all isomorphisms to a graph in a database as one is not able to differentiate multiple isomorphisms from a binary response.
We now analyse the results.
The top left of Fig.8 shows a Linear-linear plot of induced subgraph query processing time for increasing size of query graph. In this experiment, the smallest query graph contains 10 vertices while the largest query graph contains 60 vertices. For this real world graph database the *Fast Network Method* and the textitNetwork Method show almost identical processing times. In this evaluation *VF2* method is the slowest, taking more than 50 seconds to process query graphs containing more than 10 vertices. The fact that the *Fast Network Method* is not substantially better than the *network Method* implies that we do not have many fragmented graphs during decomposition, hence there is no advantage in ensuring that all decomposed graphs are connected.
Top Right of Fig.8 shows a Linear-log plot of the induce subgraph graph query processing time with query graph size. We observe that both the *Fast Network Method* and the textitNetwork Method, while almost indistinguishable, show an order of magnitude advantage in processing time over the state-of-the-art *VF2* Method.
Bottom left of Fig.8 shows a Linear-linear plot of query processing time with increasing time in the graph query. The experiment was performed using query graphs containing 10 vertices to 60 vertices. Here *Fast Network Method* is faster than *VF2* for subgraphs too. We also observe that the *VF2* method takes more than 50s to process query graphs large than 10 vertices.
Bottom Right of Fig.8 shows a Linear-log plot of the graph query processing time with increasing query graph size. We observe that both the *Fast Network Method*

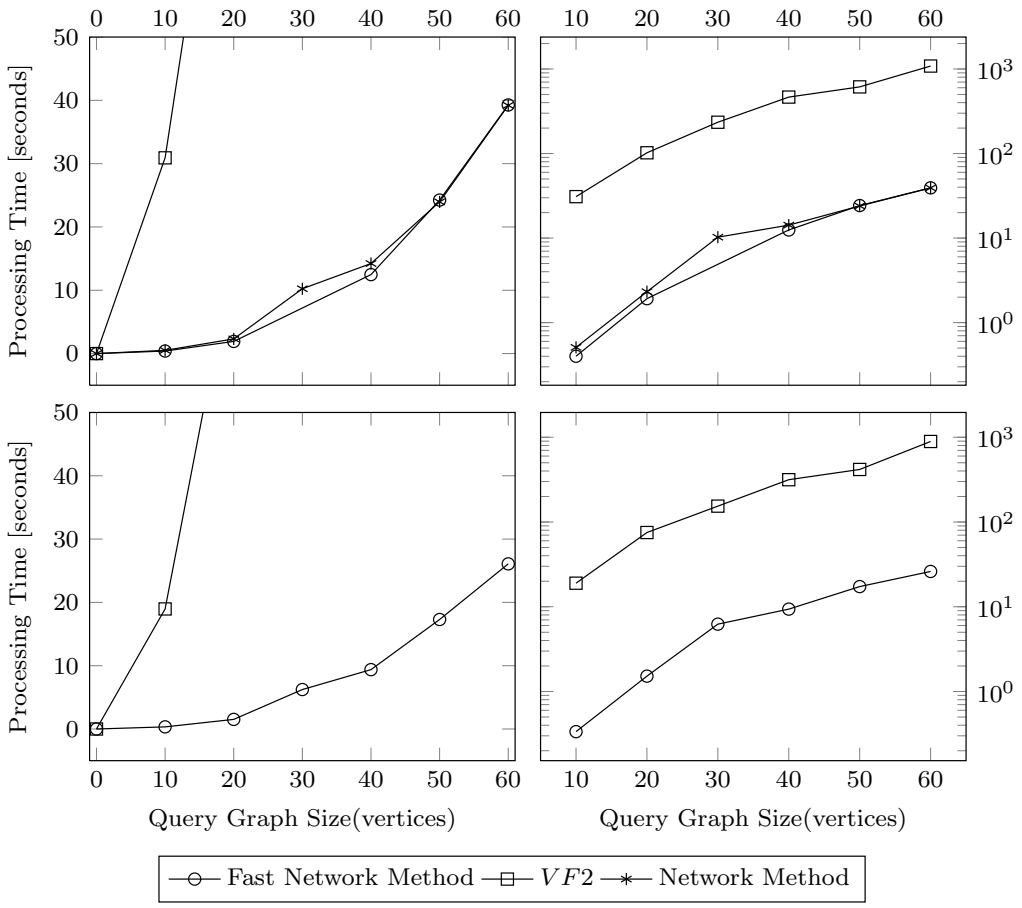

Fig. 8: **Results of the Experiments on the Real World Graph Data Set**
Top left: Linear-linear plot of induced subgraph query processing time for increasing size of query graph.The smallest query graph contains 10 vertices while the largest query graph contains 60 vertices. The *Fast Network Method* and the textitNetwork Method show almost identical processing times. The *VF2* method is noticably slower.
Top Right: Linear-log plot of the induce subgraph graph query processing time with query graph size. The *Fast Network Method* and the textitNetwork Method are almost indistinguishable and show an order of magnitude advantage over the state-of-the-art *VF2* Method
Bottom left: Linear-linear plot of query processing time for increasing size of graph query. The smallest query graph contains 10 vertices while the largest query graph contains 60 vertices. The *Fast Network Method* is faster than *VF2* for subgraphs as well. The *VF2* method takes more than 50s to process query graphs large than 10 vertices.
Bottom Right: Linear-log plot of the graph query processing time with query graph size. Both the *Fast Network Method* and the textitNetwork Method are almost indistinguishable and show more than an order of magnitude advantage over the *VF2* Method.

and the *Network Method* while almost indistinguishable, show more than an order of magnitude advantage in processing time over the state-of-the-art *VF2* Method.
It is because that their are originally few decompositions that creates disconnected graphs and also few redundant subgraph isomorphism detections in DAG. So in this experiment, our improvements did not work.
Fig.4 shows the processing time for subgraph isomorphism query. Fig.4 indicates the subgraph isomorphism query is faster than SCAN same as Messmer et al's algorithm.

6.3. Synthetic Data Search

In chemical descriptor search, we can not show improvements of proposed algorithm in term of processing time. In order to demonstrate improvements of proposed algorithm, we process subgraph isomorphism query on synthetic data.
We vary average size of query graphs and evaluate query processing time. Model graph dataset is composed of 20,000 graphs whose average number of size are 10. 100 query graphs are generated for each plot.
Fig.**??** shows the processing time for induced subgraph isomorphism query. Fig.**??** indicates that proposed algorithm is faster than Messmer et al.'s algorithm. The processing time of proposed algorithm is stable against the variation of the size of query graphs. On the contrary, the processing time of Messmer et al.'s algorithms is skyrockets when the size of query graphs becomes large.
We vary size of model graph database and evaluate the construction time of DAG. Fig.7 shows the construction time of DAG for induced subgraph isomorphism query. The construction Time of DAG of proposed algorithm is almost same as the one of Messmer et al.'s algorithm.

7. CONCLUSIONS

In this article, we have studied the problem of scalable enumeration of all subgraph isomorphisms in a graph database. We took the graph decomposition process of the *Network Algorithm* first proposed by Messmer and Bunke [Messmer and Bunke 2000] and extended it from processing only induced subgraph isomorphism to include subgraph isomorphism. The extended Algorithm likely suits more research oriented as well as real world problems. We proposed improvements on *Network Algorithm* enable it to process larger query or a larger graph database while maintaining good performance. We did this mainly by restraining the rapid increase in running time due to the creation of disconnected graphs in the DAG during query processing.
For small graphs our method shows performance that is indistinguishable from that is Messmer's *Network Algorithm*. We have shown that for large query graph and for large database our method is an order of magnitude faster than the *Network Algorithm* or VF2. This is in spite of the fact that, unlike VF2 algorithm, our algorithm processes enumeration of all isomorphic subgraphs, while VF2 will stop as soon as an isomorphic subgraph is detected in the database.

REFERENCES

AGRAFIOTIS, D. K., BANDYOPADHYAY, D., WEGNER, J. K., AND VAN VLIJMEN, H. 2007. Recent advances in chemoinformatics. *Journal of Chemical Information and Modeling 47,* 4, 1279–1293.

BABAI, L. 1981. Moderately exponential bound for graph isomorphism. In *Fundamentals of Computation Theory*, F. Gcseg, Ed. Lecture Notes in Computer Science Series, vol. 117. Springer Berlin / Heidelberg, 34–50.

BURGE, M. AND KROPATSCH, W. G. 1999. A minimal line property preserving representation of line images. *Computing 62*, 355–368. 10.1007/s006070050029.

CAI, D., SHAO, Z., HE, X., YAN, X., AND HAN, J. 2005. Community mining from multi-relational networks. In *PKDD*. 445–452.

CHEN, C., YAN, X., YU, P. S., HAN, J., ZHANG, D.-Q., AND GU, X. 2007. Towards graph containment search and indexing. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB '07. VLDB Endowment, 926–937.

CHENG, J., KE, Y., NG, W., AND LU, A. 2007. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. SIGMOD '07. ACM, New York, NY, USA, 857–872.

CHI, Y., MUNTZ, R. R., NIJSSEN, S., AND KOK, J. N. 2005. Frequent subtree mining - an overview. *Fundam. Inform. 66,* 1-2, 161–198.

COOK, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. STOC '71. ACM, New York, NY, USA, 151–158.

CORDELLA, L. P., FOGGIA, P., SANSONE, C., AND VENTO, M. 2000. Fast graph matching for detecting cad image components. In *ICPR*. 6034–6037.

CORDELLA, L. P., FOGGIA, P., SANSONE, C., AND VENTO, M. 2001. *An improved algorithm for matching large graphs*. Citeseer, 149159.

CORNEIL, D. G. AND GOTLIEB, C. C. 1970. An efficient algorithm for graph isomorphism. *J. ACM 17*, 51–64.

HOFFMANN, C. M. 1982. *Group-Theoretic Algorithms and Graph Isomorphism*. Lecture Notes in Computer Science Series, vol. 136. Springer.

LI, X.-Y., WAN, P.-J., 0003, Y. W., AND YI, C.-W. 2003. Fault tolerant deployment and topology control in wireless networks. In *MobiHoc*. 117–128.

MCKAY, B. 1981. Practical graph isomorphism. *or [Online],[cited 2004, October 21], Available from http://cs. anu. edu. au/ bdm/nauty/PGI*, 45–87.

MESSMER, B. AND BUNKE, H. 2000. Efficient subgraph isomorphism detection: a decomposition approach. *Knowledge and Data Engineering, IEEE Transactions on 12,* 2, 307 –323.

PETRAKIS, E. G. M. AND FALOUTSOS, C. 1997. Similarity searching in medical image databases. *IEEE Trans. Knowl. Data Eng. 9,* 3, 435–447.

SHASHA, D., WANG, J. T. L., AND GIUGNO, R. 2002. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '02. ACM, New York, NY, USA, 39–52.

ULLMANN, J. R. 1976. An algorithm for subgraph isomorphism. *J. ACM 23*, 31–42.

WILLETT, P., BARNARD, J. M., AND DOWNS, G. M. 1998. Chemical similarity searching. *Journal of Chemical Information and Computer Sciences 38,* 6, 983–996.

WILLIAMS, D., HUAN, J., AND WANG, W. 2007. Graph database indexing using structured graph decomposition. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. 976 –985.

YAN, X., YU, P. S., AND HAN, J. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. SIGMOD '04. ACM, New York, NY, USA, 335–346.

ZHANG, S., GAO, X., WU, W., LI, J., AND GAO, H. 2011. Efficient algorithms for supergraph query processing on graph databases. *Journal of Combinatorial Optimization 21*, 159–191. 10.1007/s10878-009-9221-1.

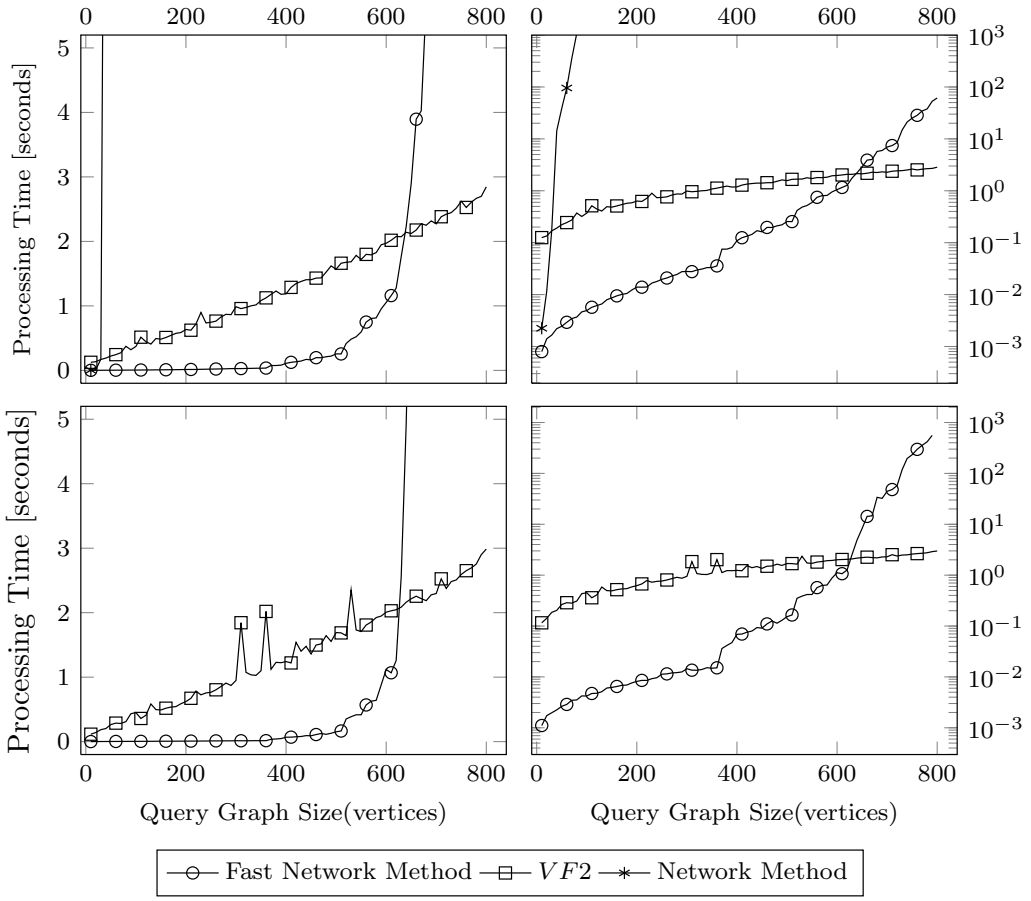

Fig. 9: **Results of the Experiments on the Synthetic Graph Data Set**
Top left: Linear-linear plot of induced subgraph query processing time against query graph vertex count. The query graph size ranges from 10 vertices to 800 vertices. The *Fast Network Method* and the *Network Method* show the fastest processing time for induced subgraph queries of up to 600 vertices. For larger query graphs, the *VF2* method is faster. The *Network Method* takes more than 5sec for any induced subgraph query graph greater than 10 vertices. The *VF2* method shows an almost linear time dependency of query graph vertex count, an advantage for induced subgraph queries larger than 600 vertices.
Top Right: Linear-log plot of the induce subgraph graph query processing time with query graph size. We observe that for induced subgraph query size less than 400 Vertices, there is an order of magnitude or better advantage to the *Fast Network Method*. Beyond query graphs of about 600 vertices the *VF2* method is clearly superior. We observe that the *Network Method* is slowest, taking about 15min for an induced subgraph query of just 80 vertices.
Bottom left: Linear-linear plot of query processing time for increasing size of graph query.The smallest query graph contains 10 vertices while the largest query graph contains 800 vertices. The *Fast Network Method* processes query graphs faster than *VF2* for subgraphs not larger than 600 vertices. Here too, *VF2* method shows an almost linear processing time dependency with query graph size.
Bottom Right: Linear-log plot of the induce subgraph graph query processing time with query graph size. For subgraph query size less than about 400 Vertices, there is an order of magnitude or better advantage to the *Fast Network Method*. For synthetic graphs of less that 50 vertices, the advantage is about two orders of magnitude. Beyond query graphs of about 600 vertices the *VF2* method is superior.