PAPER

SASUM: A Sharing-Based Approach to Fast Approximate Subgraph Matching for Large Graphs

Song-Hyon KIM^{†a)}, Student Member, Inchul SONG[†], Kyong-Ha LEE[†], and Yoon-Joon LEE[†], Nonmembers

SUMMARY Subgraph matching is a fundamental operation for querying graph-structured data. Due to potential errors and noises in real-world graph data, exact subgraph matching is sometimes inappropriate in practice. In this paper we consider an approximate subgraph matching model that allows missing edges. Based on this model, approximate subgraph matching finds all occurrences of a given query graph in a database graph, allowing missing edges. A straightforward approach is to first generate query subgraphs of a given query graph by deleting edges and then perform exact subgraph matching for each query subgraph. In this paper we propose a sharing-based approach to approximate subgraph matching, called SASUM. Our method is based on the fact that query subgraphs are highly overlapped. Due to this overlapping nature of query subgraphs, the matches of a query subgraph can be computed from the matches of a smaller query subgraph, which results in reducing the number of query subgraphs that require expensive exact subgraph matching. Our method uses a lattice framework to identify sharing opportunities between query subgraphs. To further reduce the number of graphs that need exact subgraph matching, SASUM generates small base graphs that are shared by query subgraphs and chooses the minimum number of base graphs whose matches are used to derive the matching results of all query subgraphs. A comprehensive set of experiments shows that our approach outperforms the state-of-the-art approach by orders of magnitude in terms of query execution time.

key words: graph database, approximate subgraph matching

1. Introduction

A graph is a useful data model that represents objects and their relationships in various applications. For example, a protein-protein interaction network (PPIN) is modeled as a graph where each vertex represents a protein and each edge represents an interaction between two proteins [1]. Graph data are often very large and complex, e.g., PPIN consists of tens of thousands of vertices and hundreds of thousands of edges.

One of fundamental operations in graph data processing is *subgraph matching*. Given a query graph Q and a database graph G , subgraph matching finds all occurrences of Q in G . Subgraph matching requires the *subgraph isomorphism test*, which is known to be an NP-complete problem [2]. A lot of effort has been devoted to solving the subgraph matching problem efficiently [3]–[8].

Graph data are incomplete in many cases. For example, when constructing PPIN, many PPI detection methods today produce a significant amount of false positive protein-to-

protein interactions. Moreover, they sometimes miss real interactions, generating false negatives [9]. Indeed, exact subgraph matching is inappropriate in such a case. Therefore, *approximate subgraph matching* is used instead in many applications.

In this paper we adopt an approximate subgraph matching model that allows missing edges. Missing edges represent noises in a database graph. In this model, the edge edit distance (the number of edge deletions needed to transform one graph to another) is used to identify an occurrence of Q . If the edge edit distance between a query graph Q and a subgraph g of G is no more than some user-specified threshold θ , then g is considered as an approximate match of Q . Note that we do not consider approximate matches with additional edges since such matches are always contained by the matches of the query graph.

A simple solution for approximate subgraph matching is to first generate all graphs whose edge edit distance to Q is no more than θ . We call these graphs *query subgraphs* in this paper and denote the set of all query subgraphs by $S(Q, \theta)$. Next, for each query subgraph q in $S(Q, \theta)$, we perform exact subgraph matching to find the exact occurrences of q in G . However, there are two shortcomings in this approach. First, exact subgraph matching itself is expensive since it needs the subgraph isomorphism test as mentioned above. Second, the number of query subgraphs can be very large especially when threshold θ is large.

To overcome these shortcomings, we propose a Sharing-based approach to Approximate SUBgraph Matching, called SASUM. Our approach is based on the observation that query subgraphs are highly overlapped. Due to this overlapping nature of query subgraphs, the matches of a query subgraph can be computed from the matches of a smaller query subgraph. For example, if a query subgraph q_i is a subgraph of another query subgraph q_j , then the matches of q_j can be computed from the matches of q_i by simply checking whether the additional edges, which are only contained by q_j , exist in the matches of q_i . Thus the number of graphs that need costly exact subgraph matching can be reduced. SASUM uses a lattice framework to identify sharing opportunities between query subgraphs. To further reduce the number of graphs that need exact subgraph matching, SASUM generates small *base graphs* that are shared by query subgraphs and chooses the minimum number of base graphs whose matches are used to derive the matching results of all query subgraphs. The selected base graphs are called *seed graphs*. SASUM performs subgraph match-

Manuscript received June 5, 2012.

Manuscript revised October 12, 2012.

[†]The authors are with the Department of Computer Science, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea.

a) E-mail: songhyon.kim@kaist.ac.kr

DOI: 10.1587/transinf.E96.D.624

ing only for the seed graphs and computes the matches of all query subgraphs from them systematically. We prove that the number of graphs that need subgraph matching in SASUM is less than or equal to the number required by the state-of-the-art method [10]. In many cases, SASUM reduces the number of graphs that need subgraph matching by more than half the number required by the existing method (details are described in Sect. 6). A comprehensive set of experiments with both synthetic and real datasets shows that our approach outperforms the state-of-the-art approach by orders of magnitude in terms of query execution time.

The rest of this paper is organized as follows. We present definitions and notations that are used throughout the paper in Sect. 2. We also give our formal problem statement in the section. Section 3 describes the lattice framework used in SASUM. Section 4 explains how SASUM finds the matches of all query subgraphs. In Sect. 5, we analyze our approach and compare it with the state-of-the-art approach in terms of the number of graphs that need subgraph matching. Experimental results are shown in Sect. 6. Related work is summarized in Sect. 7. Finally, we conclude the paper in Sect. 8.

2. Preliminaries

In this section we give necessary definitions and notations and present our formal problem statement. For simplicity, we investigate approximate subgraph matching for only vertex-labeled, undirected graphs. However, it is easy to extend our approach to edge-labeled and/or directed graphs without loss of generality.

Definition 1. A **labeled graph** G is a four element tuple $G = (V, E, L, l)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. L is the set of vertex labels, and the labeling function l defines the mapping: $V \rightarrow L$.

Definition 2. A labeled graph $G = (V, E, L, l)$ is **graph isomorphic** to another graph $G' = (V', E', L', l')$ if and only if there exists a bijective function $f: V \rightarrow V'$ such that

1. $\forall v \in V, l(v) = l'(f(v))$,
2. $\forall v_1, v_2 \in V, (v_1, v_2) \in E \Leftrightarrow (f(v_1), f(v_2)) \in E'$.

Definition 3. The **edge edit distance** from graph g_1 to g_2 is defined as the number of edge deletions required to transform g_1 to g_2 . We denote the edge edit distance as $\text{dist}(g_1, g_2)$. When graph g_1 cannot be transformed to graph g_2 by deleting edges, the edge edit distance from g_1 to g_2 is infinite, i.e., $\text{dist}(g_1, g_2) = \infty$.

For example, in Fig. 1, by deleting two edges from g_1 , we can transform g_1 into g_3 , i.e., $\text{dist}(g_1, g_3) = 2$.

Definition 4. Given a query graph Q and a positive integer threshold θ , a subgraph q of Q is called a **query subgraph** if $\text{dist}(Q, q) \leq \theta$. The set of query subgraphs of Q is denoted by $S(Q, \theta)$.

Fig. 1 An example of edge edit distance.

Fig. 2 Our running example.

Figure 2 shows our running example that consists of a query graph Q and a database graph G . Given the query graph Q in Fig. 2 (a) and threshold $\theta = 2$, Fig. 3 shows the query subgraphs in $S(Q, 2)$ whose edge edit distance is at least 1. Note that Q is also in $S(Q, 2)$.

Definition 5. Given a database graph G and a connected query graph Q , a connected subgraph g of G is defined as a **match** of Q in G if and only if g is graph isomorphic to Q . Given a positive integer θ as threshold, a connected subgraph g of G is defined as an **approximate match** of Q in G if and only if g is a match to some query subgraph in $S(Q, \theta)$. We say that a connected subgraph g of G is **approximately isomorphic** to Q if g is an approximate match of Q .

For example, in our running example, the subgraph of G that consists of vertices u_1, u_2, u_3 , and u_4 is a match of Q . The one with vertices u_4, u_7, u_8 , and u_9 is an approximate match of Q because it is graph isomorphic to graph q_4 in Fig. 3.

Given a graph q_i , the set of matches in a graph G is denoted by $M(q_i, G)$. A match m in $M(q_i, G)$ is represented as a set of mappings where each mapping associates a vertex of graph q_i with a vertex of graph G . A mapping is expressed as a pair (v, u) where v is in q_i and u is in G .

Example 1. Consider the query graph Q and the database graph G in Fig. 2. There is only one match of Q in G , which is represented as $M(Q, G) = \{(v_1, u_2), (v_2, u_1), (v_3, u_4), (v_4, u_3)\}$. For graph q_4 in Fig. 3, there are two matches, i.e., $M(q_4, G) = \{m_1, m_2\}$ where $m_1 = \{(v_1, u_2), (v_2, u_1), (v_3, u_4), (v_4, u_3)\}$ and $m_2 = \{(v_1, u_4), (v_2, u_8), (v_3, u_7), (v_4, u_9)\}$.

We assume that the average degree of the query graph is at least 2 (i.e., it has at least one cycle in it). Otherwise, there is no edge to delete. We also assume $\theta > 0$ for allowing missing edges in approximate matches.

Problem Statement: Given a database graph G , a query graph Q , and a positive integer threshold θ , our goal is to efficiently find all graphs that are approximately isomorphic to Q in G . In other words, we want to find each $M(q, G)$ where $q \in S(Q, \theta)$.

Fig. 3 Query subgraphs in $S(Q, 2)$ whose edge edit distance is at least 1.

3. The Lattice Framework

SASUM is a sharing-based approach to approximate subgraph matching. To reduce the number of graphs that need expensive subgraph matching, it exploits the observation that query subgraphs are highly overlapped each other.

Suppose that q_i and q_j be two query subgraphs, and q_i is a subgraph of q_j . If we have the matches of q_i in a database graph G , then we can obtain the matches of q_j in G by filtering out those matches of q_i that do not have the edges $E(q_j) \setminus E(q_i)$. For example, graph q_4 in Fig. 3 is a subgraph of Q in Fig. 2(a). In Example 1, we can compute the set $M(Q, G)$ of matches of Q from the two matches of q_4 , i.e., m_1 and m_2 , by filtering out those matches without necessary edges — in this case, m_2 is filtered out because the subgraph represented by this match does not have the edge (u_4, u_9) . Note that we must evaluate the query subgraph q_i before q_j to use the matches of q_i in computing those of q_j .

In this section we develop the notation for describing when the matches of one query subgraph can be computed from those of another query subgraph, and for determining the evaluation order of query subgraphs.

We use a lattice structure to identify sharing opportunities between query subgraphs. The subgraph relation $\subseteq$ imposes a partial ordering on the set of query subgraphs. We shall talk about identifying sharing opportunities between query subgraphs as forming a lattice. In order to be a lattice, any two elements (query subgraphs) must have a least upper bound and a greatest lower bound according to the $\subseteq$ ordering. However, in our problem, we only need the assumptions that the subgraph relation $\subseteq$ is a partial order and any two elements must have a least upper bound.

We denote a *query lattice* with set of graphs $S(Q, \theta)$ and subgraph relation $\subseteq$ by $\langle S(Q, \theta), \subseteq \rangle$. For graphs q_i, q_j of the query lattice, if $q_i \subseteq q_j$, we can compute the matches of q_j from the matches of q_i . It is common to represent a lattice by a Hasse diagram, a graph in which the lattice elements are nodes and there is a path upward from q_1 to q_2 if and only if $q_1 \subseteq q_2$. For example, Fig. 4 shows the Hasse diagram of the set of query subgraphs in $S(Q, 2)$ shown in Fig. 3.

For graphs q_i and q_j of the lattice, we say that q_j is a *successor* of q_i if and only if $q_i \subseteq q_j$ and $|E(q_i)| = |E(q_j)| - 1$. Inversely, q_i is called a *predecessor* of q_j . If q_j is a successor of q_i , we use a notation $q_i \rightarrow q_j$. If a graph q has no predecessor, it is called a *terminal graph*. Otherwise, it is a

Fig. 4 Query lattice over $S(Q, 2)$.

non-terminal graph. We denote the set of terminal graphs in the lattice by $T(Q, \theta)$.

We have to find at least the matches of every graph in $T(Q, \theta)$ by subgraph matching because we do not have any smaller graphs whose matches are used to compute the matches of the graphs in $T(Q, \theta)$. Suppose that we already have the matches of the graphs in $T(Q, \theta)$. Given a non-terminal graph q in $S(Q, \theta)$, if there is a path $q_t \rightarrow \dots \rightarrow q$ from a terminal graph q_t in $T(Q, \theta)$ to q , we can find the matches of q from those of q_t by repeatedly filtering out the matches of q_t not having necessary edges. The following lemma indicates that for every non-terminal graph in $S(Q, \theta)$, there always exists such a path.

Lemma 1. *For every non-terminal graph q in $S(Q, \theta)$, there exists a path from some terminal graph q_t in $T(Q, \theta)$ to q .*

Proof. Let q be a non-terminal graph. Then, there is a graph q_1 with $q_1 \rightarrow q$ by definition. If q_1 is a non-terminal graph, then there is a graph q_2 with $q_2 \rightarrow q_1$. Continuing this process, we arrive at a graph q_t with $q_t (= q_k) \rightarrow q_{k-1}$ where q_t is a terminal graph in $T(Q, \theta)$. $\square$

We can see from Lemma 1 that if we have the matches of all graphs in $T(Q, \theta)$, then we can compute the matches of every non-terminal graph in $S(Q, \theta)$ without performing subgraph matching from scratch. That is, we need to perform subgraph matching only for the graphs in $T(Q, \theta)$. The lattice approach tells us in what order to find the matches of query subgraphs. By using the matches of query subgraphs that have been already found, we can compute the matches of larger query subgraphs. A topological sort on the successor relation $\rightarrow$ gives us the required order of evaluation.

4. SASUM Query Processing

In this section we describe how to evaluate graphs in $S(Q, \theta)$ based on the lattice framework developed in the previous section. Lemma 1 in the previous section indicates that

Fig. 5 Overview of SASUM query processing.

we need to find the matches of only the graphs in $T(Q, \theta)$ by subgraph matching and can compute the matches of the other graphs in $S(Q, \theta)$ from them. Although the number of graphs in $T(Q, \theta)$ is smaller than that of graphs in $S(Q, \theta)$, it can still be large. For example, if a query graph Q has m edges, the number of graphs in $T(Q, \theta)$ could be $O(m^\theta)$.

To further reduce the number of graphs that need exact subgraph matching, SASUM generates a set of graphs called *base graphs*, denoted by $B(Q, \theta)$, that are shared by terminal graphs in $T(Q, \theta)$ by further deleting edges. Then it chooses the minimum number of base graphs whose matches are used to compute the matches of all graphs in $T(Q, \theta)$. The selected base graphs are called *seed graphs*, and the set of seed graphs are denoted by $B_{seed}(Q, \theta)$. The number of seed graphs in $B_{seed}(Q, \theta)$ is less than or equal to the number of terminal graphs in $T(Q, \theta)$ (we will provide a proof in Sect. 5), and it is much smaller in many cases. SASUM first finds the matches of the seed graphs by subgraph matching and then systematically computes the matches of all graphs in $S(Q, \theta)$. This sharing-based approach of SASUM leads to much reduction in the number of graphs that need costly exact subgraph matching.

Query processing in SASUM consists of three phases: *base graph generation*, *seed selection*, and *query evaluation*. Figure 5 shows an overview of query processing in SASUM. In the base graph generation phase, SASUM generates the set of base graphs $B(Q, \theta)$ from the terminal graphs in $T(Q, \theta)$. In the seed selection phase, it chooses the seed graphs from the base graphs. Finally, in the query evaluation phase, SASUM finds the matches of every graph in $S(Q, \theta)$ from the matches of the seed graphs. We describe each phase in more detail in the subsequent sections.

4.1 Base Graph Generation

SASUM generates a set of base graphs that are shared by terminal graphs through the operation of deleting edges called *edge pruning*, which is defined as follows:

Definition 6. In a graph G , pruning of edge e is the deletion of edge e and, if any, an isolated vertex (i.e., a vertex of degree 0) such that the resulting graph $G - e$ is connected and has one less edge than G . When there are two isolated vertices after edge deletion, i.e., graph G has one edge, an arbitrarily chosen vertex is deleted.

For example, Fig. 6 shows the results of pruning of

Fig. 6 An example of edge pruning.

edges e_1 and e_3 from G . Note that in $G - e_2$, the isolated vertex labeled 'A' has been removed. $G - e_1$ has one less edge than G , whereas $G - e_3$ has one less edge than G and also one less vertex.

Lemma 2. Given a connected graph G with at least one edge, edge pruning, $G - e$, always produces at least one graph with one less edge than G .

Proof. Assume that a graph G has a cycle in it. Then, we can delete any edge e in the cycle and obtain a graph $G - e$ with one less edge than G . Now assume that G has no cycle in it. Then it must be a tree (connected, acyclic graph). A leaf in a tree is a vertex of degree 1. Every tree with at least one edge has at least two leaves [11]. Take any leaf v , and delete the edge e attached to it and the vertex v itself. Then we obtain a new graph $G - e$ with one less edge and one less vertex. $\square$

Figure 7 shows the base graphs generated by edge pruning from the graphs in $T(Q, 2)$. For example, graph q_{14} can be obtained by either pruning of edge e_3 from graph q_6 , or edge e_5 from graph q_9 . Note that a base graph in $B(Q, \theta)$ may be shared by one or more terminal graphs in $T(Q, \theta)$ as you can see in this example.

4.2 Seed Selection

We formally state the *seed selection problem* where we want to select the minimum number of base graphs in $B(Q, \theta)$ whose matches are used to compute the matches of every graph in $T(Q, \theta)$. We say that a base graph q_b covers a terminal graph q_t if $q_b \subseteq q_t$. We denote the set of terminal graphs in $T(Q, \theta)$ that are covered by a base graph q_b in $B(Q, \theta)$ by $C(q_b)$. Now we are required to find the smallest subset $B_{seed}(Q, \theta)$ of $B(Q, \theta)$ such that the seed graphs in

Fig. 7 Base graphs generated from the graphs in $T(Q, 2)$.

Fig. 8 How the base graphs in $B(Q, 2)$ cover the terminal graphs in $T(Q, 2)$.

$B_{seed}(Q, \theta)$ collectively cover every graph in $T(Q, \theta)$: that is, $T(Q, \theta) = \bigcup_{q \in B_{seed}(Q, \theta)} C(q)$.

It is easy to see that the seed selection problem reduces to the set cover problem [2], whose decision version is NP-Complete. An instance of the set cover problem consists of a finite set U and a collection S of subsets of U , such that every element of U belongs to at least one subset in S . We want to find the smallest subset C of S whose union is U . The seed selection problem is transformed into the set cover problem by setting $U = T(Q, \theta)$ and $S = \{C(q) | q \in B(Q, \theta)\}$. The set cover problem requires that every element of U belongs to at least one subset in S , which is guaranteed by Lemma 2.

There is a greedy approximation algorithm for the set cover problem whose approximation ratio is $\ln|U| + 1$ [12]. This algorithm begins by selecting the largest subset from S and then deletes all its elements from U . The algorithm adds the subset containing the largest number of remaining uncovered elements repeatedly until all elements are covered. In the seed selection phase, SASUM obtains $B_{seed}(Q, \theta)$ from $B(Q, \theta)$ by using the greedy approximation algorithm just described.

Example 2. Figure 8 shows how the base graphs in $B(Q, 2)$ cover the terminal graphs in $T(Q, 2)$. There is an edge between a base graph and a terminal graph if the base graph covers the terminal graph. If we apply the greedy approximation algorithm to the base graphs in $B(Q, 2)$, the algorithm selects the base graphs in the following order: q_{15} , q_{21} , q_{14} , and q_{16} . Thus we have $B_{seed}(Q, 2) = \{q_{14}, q_{15}, q_{16}, q_{21}\}$, the size of which is only half of the size of $T(Q, 2)$.

4.3 Query Evaluation

In this section we describe how to find the matches of all graphs in $S(Q, \theta)$. SASUM first finds the matches of graphs in $B_{seed}(Q, \theta)$ by exact subgraph matching. Any exact subgraph matching algorithm can be used for this purpose. Then from these matches, it computes the matches of all graphs in $T(Q, \theta)$. Given a graph q_i in $T(Q, \theta)$, the matches of any graph q_b in $B_{seed}(Q, \theta)$ can be used to compute the matches of q_i if q_i is in $C(q_b)$. According to Lemma 1, the matches of graphs in $T(Q, \theta)$ can be used to compute the

matches of the other graphs in $S(Q, \theta)$. To reuse matching results, we have to determine the evaluation order of graphs in $S(Q, \theta)$ such that if $q_i \rightarrow q_j$, we find the matches of q_i first. By topologically sorting the query lattice $\langle S(Q, \theta), \subseteq \rangle$ on the successor relation $\rightarrow$, we can obtain the required evaluation order of graphs.

Let the determined topological order be $q_1, q_2, \dots, q_k$ where $k = |S(Q, \theta)|$. SASUM finds the matches of graphs in this order. If a graph q is a terminal graph in $T(Q, \theta)$, then SASUM already has computed the matches of q against the database graph G . If a query subgraph q is a non-terminal graph in $S(Q, \theta)$, SASUM finds the matches of q from the matches of q 's predecessors in the query lattice. It is guaranteed that the matches of these predecessors have already been found by the topological sorting order of graphs over the successor relation $\rightarrow$. We can reuse the matches of any predecessor graph here. To speed up the process of computing matches by reducing the size of intermediate matching results, SASUM uses the predecessor graph q^* with the smallest number of matches, i.e., $q^* = \arg \min_{q'} \{|M(q', G)| : q' \rightarrow q\}$ to compute the matches of q .

Example 3. In Fig. 4, the matches of q_4 can be computed from the matches of any one of graphs q_7 , q_{10} , q_{11} , and q_{13} . The number of matches of these graphs are as follows: $|M(q_7, G)| = 5$, $|M(q_{10}, G)| = 4$, $|M(q_{11}, G)| = 2$, and $|M(q_{13}, G)| = 6$. Therefore, SASUM computes the matches of q_4 from those of q_{11} .

Now we describe how to compute the matches of a graph from those of another graph. There are two cases to consider when reusing matching results: (1) from the matches of a graph in $B_{seed}(Q, \theta)$ to those of a graph in $T(Q, \theta)$, and (2) from the matches of a graph in $S(Q, \theta)$ to another graph in $S(Q, \theta)$. We consider the case (2) first. When q_i and q_j are both in $S(Q, \theta)$ and $q_i \rightarrow q_j$, then graph q_j has one additional edge than graph q_i . In this case, for each match in $M(q_i, G)$, SASUM checks whether that match has the additional edge and prunes those matches not having that edge.

We now discuss the case (1). When q_i is in $B_{seed}(Q, \theta)$, q_j is in $T(Q, \theta)$, and q_j is in $C(q_i)$, then q_j has either (i) one additional edge, or (ii) one additional edge and one additional vertex since q_i is generated by edge pruning. For the case (i), SASUM does the same as in the case (2) above. For the case (ii), however, we cannot compute the matches of q_j by only pruning the matches of q_i . Instead, we have to extend the matches of q_i with the mappings of the additional vertex. Let graph q_j have n vertices and q_i have $n-1$ vertices. Given a match m in $M(q_i, G)$, let vertices $u_1, u_2, \dots, u_{n-1}$ in G are the mappings of vertices $v_1, v_2, \dots, v_{n-1}$ in q_i . Suppose

Fig. 9 Computing the matches of graphs q_6 and q_1 from those of graph q_{14} .

that v_n is an additional vertex in q_j , and there is an additional edge (v_i, v_n) in q_j where $1 \leq i \leq n - 1$. Now we extend the match m with the mappings (v_n, u_n) for u_n in G such that the edge (u_i, u_n) exists in $E(G)$, and the label of u_n is equal to that of v_n , i.e., $l(u_n) = l(v_n)$. We can easily find such u_n 's from G by visiting the adjacent vertices u_j of u_i and checking whether $l(u_j) = l(v_n)$.

Example 4. We illustrate how to compute matches from other matches by an example, shown in Fig. 9. In this example, we shall compute the matches of graphs q_6 and q_1 from those of seed graph q_{14} . On the left, we show the query graph Q and the database graph G again for the convenience of the reader. On the right, we show the graphs q_{14} , q_6 , and q_1 where q_{14} covers q_6 , and $q_6 \rightarrow q_1$. The set of matches of a graph is shown below the graph, as a table. The first row of a table shows the vertices in the graph above it, and the remaining rows represent the matches of the graph. We use $m_1, m_2, \dots$ to denote the first match (row), the second match, and so on, in each set of matches.

Assume that we already have found the matches in $M(q_{14}, G)$ by subgraph matching. Now we compute $M(q_6, G)$ from $M(q_{14}, G)$. In the figure, graph q_6 has one additional edge (v_1, v_3) and one additional vertex v_3 than graph q_{14} (indicated by using solid lines). Thus, we need to compute $M(q_6, G)$ from $M(q_{14}, G)$ by extending each match in $M(q_{14}, G)$ with the mappings of vertex v_3 . Let us consider the match $m_1 = \{(v_1, u_2), (v_2, u_1), (v_4, u_3)\}$ first. Since the additional edge is between vertices v_1 and v_3 , and the label of vertex v_3 is 'A', we find the adjacent vertices of u_2 in G having label 'A'. Only vertex u_4 is eligible as the mapping of vertex v_3 . Thus, the match m_1 in $M(q_{14}, G)$ is extended by the mapping (v_3, u_4) to be the match m_1 in $M(q_6, G)$. The other matches in $M(q_{14}, G)$ can be extended in the same way just described. A dotted arrow from a match m in $M(q_{14}, G)$ to a match m' in $M(q_6, G)$ indicates that the match m is extended to be the match m' .

Next we compute $M(q_1, G)$ from $M(q_6, G)$. The graph q_1 has one additional edge (v_2, v_4) than q_6 . Here we compute $M(q_1, G)$ by filtering out those matches in $M(q_6, G)$ not having the edge corresponding to the edge (v_2, v_4) in G .

The matches m_2 and m_3 in $M(q_6, G)$ are filtered out because there is no edge between vertices u_8 and u_3 .

4.4 Outputting Matching Results

In the problem statement present in Sect. 2, we do not impose any restrictions on the order of outputting the matching results of the graphs in $S(Q, \theta)$. However, the user may want to get the matching results in the order of the edge edit distance from the query graph: that is, the matching results of the query graph, and those with one missing edge, those with two missing edges, and so on. In this case, we must keep the matching results of every graph in $S(Q, \theta)$ before outputting them.

If the user does not care about the order of outputting results, then SASUM can reduce the space usage by producing and removing intermediate matching results early. There are two cases to consider. First, let q be a graph in $B_{seed}(Q, \theta)$. If SASUM has obtained the matches of all graphs q' where q' is in $C(q)$, then SASUM can safely throw away the matches of q because they will not be used later. Second, let q be a graph in $S(Q, \theta)$. If SASUM has obtained the matches of all graphs that are successors of q , then SASUM can remove the matches of q right away for the same reason.

5. Analytical Study

This section analyzes our approach to approximate subgraph matching. We aim at proving the correctness of SASUM and showing the superiority of our approach compared to the state-of-the-art method in terms of the number of graphs that need subgraph matching.

5.1 Proof of Correctness

The following theorem shows the correctness of SASUM.

Theorem 1. Given a database graph G , a connected query graph Q , and a positive integer threshold θ , SASUM finds all matches of graphs in $S(Q, \theta)$.

Proof. In the evaluation phase, SASUM first finds the matches of graphs in $B_{seed}(Q, \theta)$ by subgraph matching. Then it computes the matches of graphs in $T(Q, \theta)$ from those of graphs in $B_{seed}(Q, \theta)$. Now it remains to show whether SASUM correctly computes the matches of all non-terminal graphs in $S(Q, \theta)$. According to Lemma 1, for every non-terminal graph q in $S(Q, \theta)$, there is a path from some terminal graph q_t in $T(Q, \theta)$ to q . Let the path be $q_1(= q_t) \rightarrow q_2 \rightarrow \dots \rightarrow q_k(= q)$. By the evaluation order of graphs, which is a topological order, the matches of q_i is computed from those of q_{i-1} for $2 \leq i \leq k$. We already have the matches of q_t . Therefore, SASUM computes the matches of q eventually. $\square$

5.2 Performance Guarantee of SASUM

We compare SASUM with the state-of-the-art approach in terms of the number of graphs that need costly subgraph matching. The number of graphs that need subgraph matching is a dominant factor in the overall performance of a given method for approximate subgraph matching (we will verify this in Sect. 6 through experimental evaluation). We compare three approaches: NAIVE, SHARE, and SASUM. The NAIVE approach is the one that finds the matches of each graph in $S(Q, \theta)$ independently. The SHARE approach is a basic sharing-based approach, which is employed by the state-of-the-art method [10]. It computes the matches of a query subgraph from those of another query subgraph. SHARE needs subgraph matching for the graphs in $T(Q, \theta)$. Our approach, SASUM, increases sharing opportunities by generating base graphs and selecting a small number of seed graphs. SASUM needs subgraph matching for the graphs in $B_{seed}(Q, \theta)$. Let C_{NAIVE} , C_{SHARE} , and C_{SASUM} denote the number of graphs that need subgraph matching by NAIVE, SHARE, and SASUM, respectively.

It is easy to see that $C_{SHARE} < C_{NAIVE}$ because $T(Q, \theta) \subset S(Q, \theta)$. The following theorem shows that SASUM needs no more graphs than SHARE that need subgraph matching:

Theorem 2. $C_{SASUM} \leq C_{SHARE}$.

Proof. We shall show $|B_{seed}(Q, \theta)| \leq |T(Q, \theta)|$. SASUM selects the seed graphs in $B_{seed}(Q, \theta)$ from the base graphs in $B(Q, \theta)$ by transforming the seed selection problem into the set cover problem and using the greedy approximation algorithm for the set cover problem, as described in Sect. 4.2. The greedy approximation algorithm selects one graph from $B(Q, \theta)$ at a time, and removes the graphs covered by the selected graph from $T(Q, \theta)$. Since every graph in $B(Q, \theta)$ covers at least one graph in $T(Q, \theta)$ (by Lemma 2), at least one graph is removed from $T(Q, \theta)$ at each iteration of the greedy approximation algorithm. Hence, the number of seed graphs in $B_{seed}(Q, \theta)$ selected by the greedy approximation algorithm is bounded by the number of terminal graphs in $T(Q, \theta)$, that is, $|B_{seed}(Q, \theta)| \leq |T(Q, \theta)|$. $\square$

Table 1 Default parameter values.

Parameter	Default value
Number of vertices $ V(G) $ in G	5000
Average degree $deg(G)$ of G	8
Number of distinct labels $ L(G) $	250
Number of vertices $ V(Q) $ in Q	20
Average degree $deg(Q)$ of Q	3
θ	1

6. Experimental Evaluation

6.1 Setup

We implemented our algorithm as a single threaded executable in C++. We compare our method with SAPPER [10], the state-of-the-art method for approximate subgraph matching. SAPPER evaluates the graphs in $S(Q, \theta)$ in the depth first search order and finds the matches of a graph q in $S(Q, \theta)$ from the matches of another graph q' in $S(Q, \theta)$ where $q' \rightarrow q$, similar to our approach. However, SAPPER always performs subgraph matching for every graph in $T(Q, \theta)$, and which q' to use is predetermined. We downloaded an executable of SAPPER from the authors' website[†]. We used the same exact subgraph matching algorithm used in SAPPER when implementing SASUM. All experiments were performed on a machine with Intel Xeon(R) CPU E5345 2.33 GHz and 8 GB main memory, running on Fedora 12 Linux operating system. We used synthetic datasets and two real-world datasets in the experiments.

6.2 Synthetic Datasets

We first compare the two approaches on synthetic datasets. We generated synthetic datasets with a graph generator, **gengraph** [13]^{††}. The default values of the parameters are listed in Table 1. Synthetic graphs generated by the graph generator have a heavy tailed degree sequence (i.e., a sequence of vertex degrees in descending order) to model real datasets^{†††}.

We analyze the performance of SASUM and SAPPER by independently varying each of $|V(G)|$, $|L(G)|$, θ , $|V(Q)|$, and $deg(Q)$. The results are shown in Figs. 10 and 12. In Fig. 10, SASUM outperforms SAPPER in all cases. In particular, consider the case when the database graph is large (in Fig. 10(a)) and the number of distinct labels is small (in Fig. 10(b)) where subgraph matching requires much time. In this case, SASUM performs far better than SAPPER because SASUM requires a less number of graphs that need subgraph matching. In Fig. 10(c) and Fig. 10(d), we vary

[†]<http://sites.google.com/site/shijiezhang/Home/grapham-a-graph-approximate-matching-tool> (accessed on April 12, 2011)

^{††}downloaded at <http://fabien.viger.free.fr/liafa/generation/>

^{†††}We also used graphs whose degree distribution of vertices is uniform. We obtained similar results, thus do not include the results here.

Fig. 10 Query execution time for synthetic datasets. **Fig. 11** Comparing the number of subgraph matching executions. **Fig. 12** Query execution time for different query graphs.

threshold θ in two different sets of database graphs. As we can see in the figures, SASUM outperforms SAPPER by orders of magnitude because SASUM requires a far less number of graphs that need subgraph matching. When threshold θ is large, the number of graphs in $T(Q, \theta)$ is large, but SASUM uses $B_{seed}(Q, \theta)$ instead, which is far smaller than $T(Q, \theta)$. We can verify this in Fig. 11, which shows the reduction ratio of SASUM in the number of graphs that need subgraph matching. For example, the reduction ratio of 0.6 indicates that SASUM reduces the number of graphs that need subgraph matching by 60% when compared to SAPPER. As you can see in the figure, SASUM requires less than half the number of graphs that need subgraph matching required by SAPPER in all cases, and as θ increases, the reduction ratio further increases.

We vary the number of vertices in Q and, the results are shown in Fig. 12 (a). With more vertices in Q , more vertices and edges need to be compared in subgraph matching. Thus the query times of both methods increase. However, SASUM performs better than SAPPER because it requires

Fig. 13 Breakdown of query execution time.

a smaller number of graphs that need subgraph matching.

The last parameter we vary is the average degree of Q . The results are shown in Fig. 12 (b). A high vertex degree generates more graphs in $S(Q, \theta)$ since the number of graphs in $S(Q, \theta)$ is exponential to the average degree of Q . And when the average degree of Q is large, subgraph matching needs to compare more edges. Thus, as the average degree of Q increases, the performance gap between SASUM and SAPPER widens.

We analyze query execution time of SASUM to access relative portion of time spent at subgraph matching and that spent at the other parts. The results are shown in Fig. 13 (a) and (b). Figure 13 (a) shows the portion of time spent by subgraph matching and the other parts in the overall query execution time. As the number of vertices in G increases, subgraph matching needs to compare more vertices. Thus, the portion of time spent by subgraph matching increases and dominates the overall performance. In Fig. 13 (b), we vary threshold θ . Here the portion of time spent by the other parts increases as θ increases. This is because the number of graphs in $S(Q, \theta)$ is exponential to θ . Nevertheless, the portion of time spent by subgraph matching is still dominant in all cases.

6.3 Real datasets

For real-world data, we prepared two large real graphs: a human protein interaction network [14] and a collaboration network [15]. The protein interaction network contains 10,527 vertices and 40,903 edges. Each vertex represents a protein, and the label of the vertex is its gene ontology term. An edge in the graph represents an interaction between the two proteins it connects. The collaboration network includes 5,241 vertices and 14,484 vertices. Each vertex represents an author, and there is an edge between two authors if they coauthored a paper. 250 labels are randomly distributed over the vertices in the collaboration network.

Fig. 14 Degree sequences of real graphs.

Fig. 15 Query execution time with real graphs.

Figure 14 shows sample degree sequences of the two real datasets. As you can see in the figure, their degree sequences show a heavy-tail behavior.

We compare the performance of SASUM and SAPPER over different query graphs extracted from the two graphs. Since most of the results are similar to those for the synthetic datasets, we show the results of varying only threshold θ , which are shown in Fig. 15. The figure shows that SASUM outperforms SAPPER by orders of magnitude in terms of query execution time in both datasets.

7. Related Work

Subgraph matching, which finds the occurrences of a specified graph pattern in a graph, is a fundamental operation in graph data processing. Ullmann [3] proposed a subgraph matching algorithm based on a state space search method with backtracking. VF2 [4] is a more recent work that introduces a set of feasibility rules for pruning the state space. These two methods, however, are prohibitively expensive for query processing against a large database graph since they do not use any index structure by preprocessing the database graph.

Several indexing based method have been developed for subgraph matching. In graph indexing methods, e.g., GraphGrep [16], gIndex [17], TreePi [18], Tree+ Δ [19], and FG-Index [20], the graph database consists of a set of small graphs. The goal of graph indexing is to find all graphs that contain a given query graph. On the other hand, subgraph indexing finds all occurrences of a given query graph in a very large database graph. GADDI [5], NOVA [6], SUMMA [7], the approach proposed in [8], and SAPPER [10] fall into this category. Our approach, SASUM, also belongs to this category.

Recently, a number of methods have been proposed to support approximate subgraph matching [10], [21], [22].

Among them, similarity search methods, e.g., TALE [21] and G-Hash [22], are not designed to find all approximate matches of a given query graph. SAPPER [10] is the state-of-the-art method that finds all approximate matches of a given query graph from a large database graph. SAPPER uses the fact that subgraphs of a given query are highly overlapped. It finds the matches of a larger graph from those of a smaller graph. However, SAPPER still needs to perform subgraph matching for a large number of graphs.

8. Conclusions

In this paper we have investigated how to find all approximate matches of a given graph from a large database graph, allowing missing edges. A straightforward method is to generate a set of query subgraphs, which have no more missing edges than a user-specified threshold, and perform subgraph matching for each query subgraph independently. However, this simple method is infeasible in many cases since the number of query subgraphs could still be too large, and subgraph matching itself is a very difficult problem.

To address this problem, we have proposed a sharing-based approach to fast approximate subgraph matching called SASUM. We aim at reducing the number of graphs that need subgraph matching, which decides the overall performance of an approximate subgraph matching algorithm. To this end, SASUM exploits the fact that query subgraphs are highly overlapped. Due to this property, the matches of a query subgraph can be computed from those of another query subgraph without costly subgraph matching. SASUM uses a lattice framework to identify sharing opportunities between query subgraphs. SASUM goes one step further and produces small base graphs that are shared by query subgraphs by edge pruning. It then chooses a small number of seed graphs from them, and performs subgraph matching only for the seed graphs. SASUM computes the matches of all query subgraphs from those of seed graphs systematically. We have proven that the number of graphs that need subgraph matching required by SASUM is less than or equal to the number required by the state-of-the-art method, and it is much smaller in many cases as shown through the experimental evaluation. A comprehensive set of experiments has shown that SASUM outperforms the state-of-the-art method by orders of magnitude in terms of query execution time due to great reduction in the number of graphs that need subgraph matching.

Acknowledgements

This work was partly supported by NRF grant funded by Korea Government (No. 2011-0016282).

References

- [1] T. Keshava Prasad, R. Goel, K. Kandasamy, S. Keerthikumar, S. Kumar, S. Mathivanan, D. Telikicherla, R. Raju, B. Shafreen, A. Venugopal, L. Balakrishnan, A. Marimuthu, S. Banerjee, D. Somanathan, A. Sebastian, S. Rani, S. Ray, C. Harrys, S. Kishore,

- M. Kanth, M. Ahmed, R. Kashyap, Y. Mohmood, V. Ramachandra, B. Krishna, A. Rahiman, S.S. Mohan, P. Ranganathan, S. Ramabadran, R. Chaerkady, and A. Pandey, "Human protein reference database—2009 update," *Nucleic Acids Res.*, vol.37, no.suppl 1, pp.D767–D772, 2009.
- [2] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, WH Freeman & Co. NY, USA, 1979.
- [3] J. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol.23, no.1, pp.31–42, 1976.
- [4] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol.26, no.10, pp.1367–1372, 2004.
- [5] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," *Proc. EDBT*, pp.192–203, 2009.
- [6] K. Zhu, Y. Zhang, X. Lin, G. Zhu, and W. Wang, "Nova: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs," *Proc. DASFAA*, pp.140–154, 2010.
- [7] S. Zhang, S. Li, and J. Yang, "Summa: subgraph matching in massive graphs," *Proc. CIKM*, pp.1285–1288, 2010.
- [8] S. Kim, I. Song, and Y. Lee, "An edge-based framework for fast subgraph matching in a large graph," *Proc. DASFAA*, pp.404–417, 2011.
- [9] S. Suthram, T. Shlomi, E. Ruppin, R. Sharan, and T. Ideker, "A direct comparison of protein interaction confidence assignment schemes," *BMC Bioinf.*, vol.7, no.1, p.360, 2006.
- [10] S. Zhang, J. Yang, and W. Jin, "Sapper: subgraph indexing and approximate matching in large graphs," *Proc. VLDB Endow.*, vol.3, no.1-2, pp.1185–1194, 2010.
- [11] D.B. West, *Introduction to Graph Theory*, Prentice Hall, 2001.
- [12] T. Cormen, *Introduction to algorithms*, MIT electrical engineering and computer science series, MIT Press, 2001.
- [13] F. Viger and M. Latapy, "Efficient and simple generation of random simple connected graphs with prescribed degree sequence," *Computing and Combinatorics*, pp.440–449, 2005.
- [14] C. Stark, B. Breitkreutz, A. Chatr-aryamontri, L. Boucher, R. Oughtred, M. Livstone, J. Nixon, K. Van Auken, X. Wang, X. Shi, T. Regulj, J. Rust, A. Winter, K. Dolinski, and M. Tyers, "The biogrid interaction database: 2011 update," *Nucleic acids research*, vol.39, no.suppl 1, pp.D698–D704, 2011.
- [15] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densefication and shrinking diameters," *ACM Trans. Knowledge Discovery from Data*, vol.1, no.1, pp.2:1–2:41, 2007.
- [16] D. Shasha, J.T.L. Wang, and R. Giugno, "Algorithmics and applications of tree and graph searching," *Proc. PODS*, pp.39–52, 2002.
- [17] X. Yan, P. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," *Proc. SIGMOD*, pp.335–346, 2004.
- [18] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," *Proc. ICDE*, pp.966–975, 2007.
- [19] P. Zhao, J. Yu, and P. Yu, "Graph indexing: tree + delta $\geq$ graph," *Proc. VLDB*, pp.938–949, 2007.
- [20] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases," *Proc. SIGMOD*, pp.857–872, 2007.
- [21] Y. Tian and J. Patel, "Tale: A tool for approximate large graph matching," *Proc. ICDE*, pp.963–972, 2008.
- [22] X. Wang, A. Smalter, J. Huan, and G. Lushington, "G-hash: towards fast kernel-based similarity search in large graph databases," *Proc. EDBT*, pp.472–480, 2009.

Song-Hyon Kim received his B.S. degree in Computer Science from Korea Air Force Academy, Korea, and M.S. degree in Computer Science from KAIST, Daejeon, Korea, in 2000 and 2005, respectively. He is currently a Ph.D. candidate of the Department of Computer Science at KAIST. His research interests include database systems, graph databases, and distributed/parallel processing of large-scale data.

Inchul Song received his B.S. degree in Information and Computer Engineering from Ajou University, Suwon, Korea, in 2004, and his M.S. and Ph.D. degrees in Computer Science from KAIST, Daejeon, Korea, in 2006 and 2012, respectively. He is currently working for Samsung Electronics, Yongin, Korea, as a research staff member. His research interests include database systems, graph databases, lifelog management systems, and sensor networks.

Kyong-Ha Lee received the BS and MS degrees in Information Communications Engineering and the PhD degree in Computer Engineering from Chungnam National University, Korea, in 1998, 2000, and 2006, respectively. He is currently a research professor of computer science at KAIST, Korea. He has served as reviewers for numerous conference and journals including TKDE and DKE. He is also a member of ACM and KIISE computing society. His current research areas of interest include Massively

Parallel Processing (MPP) system, XML indexing and query processing, and database techniques for modern H/W.

Yoon-Joon Lee received his B.S. degree in Computer Engineering from Seoul National University, Seoul, Korea, and M.S. degree in Computer Science from KAIST, Daejeon, Korea, and Ph.D. degree in Computer Science from INPG-ENIMAG, France, in 1977, 1979, and 1983, respectively. He joined the faculty of the Department of Computer Science at KAIST, Daejeon, Korea, in 1984, where currently he is a full professor. His research interests include database systems, OLAP, and distributed/parallel processing of large-scale data.

distributed/parallel processing of large-scale data.