On Graph Query Optimization in Large Networks

Peixiang Zhao Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.
pzhao4@illinois.edu hanj@cs.uiuc.edu

ABSTRACT

The dramatic proliferation of sophisticated networks has resulted in a growing need for supporting effective querying and mining methods over such large-scale graph-structured data. At the core of many advanced network operations lies a common and critical graph query primitive: *how to search graph structures efficiently within a large network?* Unfortunately, the graph query is hard due to the NP-complete nature of subgraph isomorphism. It becomes even challenging when the network examined is large and diverse. In this paper, we present a high performance graph indexing mechanism, SPath, to address the graph query problem on large networks. SPath leverages decomposed shortest paths around vertex neighborhood as basic indexing units, which prove to be both effective in graph search space pruning and highly scalable in index construction and deployment. Via SPath, a graph query is processed and optimized beyond the traditional vertex-at-a-time fashion to a more efficient *path-at-a-time* way: the query is first decomposed to a set of shortest paths, among which a subset of candidates with good selectivity is picked by a query plan optimizer; Candidate paths are further joined together to help recover the query graph to finalize the graph query processing. We evaluate SPath with the state-of-the-art GraphQL on both real and synthetic data sets. Our experimental studies demonstrate the effectiveness and scalability of SPath, which proves to be a more practical and efficient indexing method in addressing graph queries on large networks.

1. INTRODUCTION

Recent years have witnessed a rapid proliferation of networks, such as communication networks, biological networks, social networks and the Web, most of which can be naturally modeled as large graphs [5]. The burgeoning size and heterogeneity of networks have inspired extensive interest in supporting effective querying and mining methods in real applications that are centered on massive graph data. At the core of many advanced network operations, lies a common and critical *graph query* primitive: given a network modeled as a large graph G, and a user-specified query graph Q, we want to retrieve as output the set of subgraphs of G, each of which is isomorphic to Q. As a key ingredient of many advanced applications in large networks, the graph query is frequently issued in various domains: (1) in a large protein-protein interaction network, it is desirable to find all protein substructures that contain an α-β-barrel motif, which is specified as a cycle of β strands embraced by another cycle of α helices [12]; (2) in a large software program which is modeled as large static or dynamic call graphs, software engineers want to locate a suspicious bug which arises as a distortion in the control flow and can be represented as a graph as well [7]; (3) in a bibliographic information network, such as DBLP, users are always eager to extract coauthor information in a specified set of conference proceedings [19]. A co-authorship graph is therefore reported as the graph query result.

Unfortunately, the graph query problem is hard in that (1) it requires subgraph isomorphism checking of Q against G, which has proven to be NP-complete [10]; (2) the heterogeneity and sheer size of networks hinder a direct application of well-known graph matching methods [9, 18, 20, 21, 27], most of which are designed on special graphs with no or limited guarantee on query performance and scalability support. Due to the lack of scalable graph indexing mechanisms and cost-effective graph query optimizers, it becomes hard, if not impossible, to search and analyze any reasonably large networks. For example, browsing and crosschecking biological networks depicted simultaneously in multiple windows is by no means an inspiring experience for scientists. Therefore, there is a growing need and strong motivation to take advantage of well-studied database indexing and query optimization techniques to address the graph query problem on the large network scenario.

In this paper, we present SPath, a new graph indexing technique towards resolving the graph query problem efficiently on large networks. SPath maintains for each vertex of the network a *neighborhood signature*, a compact indexing structure comprising decomposed shortest path information within the vertex's vicinity. As a basic graph indexing unit, neighborhood signature demonstrates considerable merits in that (1) neighborhood signature is very space-efficient ($O(1)$ for each vertex), which makes it possible to scale SPath up in large networks; (2) neighborhood signature preserves local structural information surrounding vertices, which is especially useful for search space pruning before costly subgraph matching; (3) neighborhood signature based graph index-

Proceedings of the VLDB Endowment, Vol. 3, No. 1

ing, SPath, revolutionizes the way of graph query processing from *vertex-at-a-time* to *path-at-a-time*, which proves to be more cost-effective than traditional graph matching methods.

With the aid of SPath, we decompose a query graph into a set of shortest paths, among which a subset of candidate paths with high selectivity is picked by a graph query optimizer. These candidate paths are required to properly cover the original query graph, i.e., for each edge in Q, it should belong to at least one candidate path selected. The query is further processed by joining candidate shortest paths in order to reconstruct the original query graph. Here the graph matching is performed in a path-at-a-time fashion and SPath plays a key role in shortest path reconstruction and orientation in the large network. To the best of our knowledge, SPath is the first scalable graph indexing mechanism which supports effective path-at-a-time graph query processing on large networks, and thus achieves far better query performance, compared with other traditional vertex-at-a-time graph matching methods. Our main contributions can be summarized as follows:

1. We propose a pattern based graph indexing framework to address the graph query problem on large networks. A query cost model is formulated to help evaluate different structural patterns for graph indexing in a qualitative way. As a result, decomposed shortest paths are considered as feasible indexing features in the large network scenario (Section 3);

2. We propose a new graph indexing technique, SPath, which makes use of neighborhood signatures of vertices as the basic indexing structure. SPath has demonstrated an effective search space pruning ability and high scalability in large networks (Section 4);

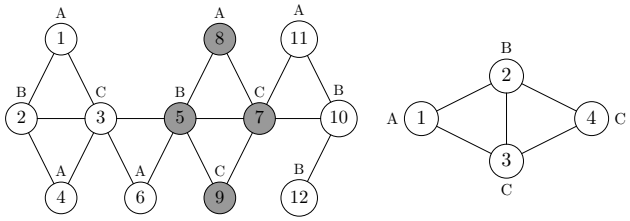
3. We design a graph query optimizer to help address graph queries in a path-at-a-time manner. With the aid of SPath, the graph query processing is facilitated by joining a set of shortest paths with good selectivity (Section 5);

4. We present comprehensive experimental studies on both real and synthetic networks. Our experimental results demonstrate that SPath outperforms a state-of-the-art graph query method, GraphQL [12]. Moreover, SPath exhibits excellent scalability and practicability in large networks (Section 6).

2. PROBLEM DEFINITION

A network can be modeled as a graph $G = \{V, E, \Sigma, l\}$ where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. Σ is a vertex label set and $l : V \rightarrow \Sigma$ denotes the vertex labeling function. For ease of notation, the vertex set of G is denoted as $V(G)$ and its edge set is denoted as $E(G)$. The *size* of G is defined as $|V(G)|$, the size of its vertex set. Analogously, the graph queries posed upon the network can be modeled as graphs as well. In this paper, we focus our study on the case of connected, undirected simple graphs with no weights assigned on edges. Without loss of generality, our methods can be easily extended to other kinds of graphs.

A graph G' is a *subgraph* of G, denoted as $G' \subseteq G$, if $V(G') \subseteq V(G)$, $E(G') \subseteq E(G)$ and $\forall(u, v) \in E(G'), u, v \in V(G')$. We alternatively say that G is a *supergraph* of G' and G *contains* G'. *Subgraph isomorphism* is an injective function f defined from G' to G such that (1) $\forall v \in V(G'), l'(v) = l(f(v))$; and (2) $\forall(u, v) \in E(G'), (f(u), f(v)) \in E(G)$, where l' and l are the labeling functions of G' and G, respectively. Under these conditions, f is often referred to as a *matching* of G' in G.



(a) G (b) Q

Figure 1: A Network G and a Query Graph Q

DEFINITION 1. *(**GRAPH QUERY**) Given a network G and a query graph Q, the graph query problem is to find as output all distinct matchings of Q in G.*

EXAMPLE 1. *Figure 1(a) and Figure 1(b) illustrate a network sample G and a query graph sample Q, respectively. Here we use numeric identifiers to distinguish different vertices in a graph. A subgraph G' of G with $V(G') = \{8, 5, 7, 9\}$ colored in grey is isomorphic to Q and hence returned as an answer to the graph query.*

Note there may exist multiple matchings of Q in G. For example, given a triangle graph Q with A, B, C as the label for each vertex, respectively. All the matchings of Q in G, as shown in Figure 1(a), are $\{1, 2, 3\}$, $\{4, 2, 3\}$, $\{6, 5, 3\}$, $\{8, 5, 7\}$ and $\{11, 10, 7\}$. Subgraph isomorphism is known to be NP-complete [10].

3. THE PATTERN-BASED GRAPH INDEXING FRAMEWORK

In this section, we discuss the feasibility and principle of graph indexing toward addressing the graph query problem on large networks. We first introduce a baseline algorithmic framework with no indexing techniques exploited (Section 3.1). In order to improve the query performance, we extend the framework by leveraging structural patterns for graph indexing (Section 3.2). A cost-sensitive model is then proposed to help evaluate different structural patterns qualitatively (Section 3.3). As a result, path-based graph indexing mechanism is selected as a feasible solution in large networks.

3.1 The Baseline Algorithmic Framework

A straightforward approach to answering the graph query Q against a network G is to explore a tree-structured search space considering all possible vertex-to-vertex correspondences from Q to G. The search space traversal is halted until the structure of Q implied by the vertex mapping does not correspond in G. While reaching a leaf node of the search space means successfully mapping all vertices of Q upon G without violating the structure and label constraints of subgraph isomorphism, and is therefore equivalent to having found a matching Q in G. Algorithm 1 outlines the baseline algorithm in detail (See Appendix 9.2).

DEFINITION 2 (**MATCHING CANDIDATE**). *$\forall v \in V(Q)$, the matching candidates of v is a set $C(v)$ of vertices in G sharing the same vertex label with v, i.e., $C(v) = \{u | l(u) = l'(v), u \in V(G)\}$, where l and l' are vertex labeling functions for G and Q, respectively.*

In the baseline algorithm, for each vertex $v \in V(Q)$, an exhaustive search of possible one-on-one correspondences to $u \in C(v)$ is required. Therefore, the total *search space* of the algorithm equals $\prod_{i=1}^{N} C(v_i)$, where $N = |V(Q)|$. The worst-case time complexity of the algorithm is $O(M^N)$ where M and N are the sizes of G and Q, respectively. This is a consequence of subgraph isomorphism that is known to be NP-complete. In practice, the running time of graph query processing depends tightly on the size of the search space, $\prod_{i=1}^{N} |C(v_i)|$.

3.2 Structural Pattern Based Graph Indexing

It has been shown that answering graph queries is very costly, and it becomes even challenging when the network examined is large and diverse. In order to alleviate the time-consuming exhaustive search in graph query processing, we consider reducing the search space size, $\prod_{i=1}^{N} |C(v_i)|$, in the following two aspects:

1. Minimize the number of one-on-one correspondence checkings, i.e, **min** N;
2. Minimize for each vertex in the query graph its matching candidates, i.e., **min** $|C(v_i)|, \forall v_i \in V(Q)$.

The two objectives motivate us to explore the possibility of leveraging structural patterns for graph indexing such that the search space size can be ultimately minimized. For structural patterns, we mean any kind of substructures of a graph, such as paths, trees, and general subgraphs.

We begin considering the first objective to reduce N. Note in the baseline algorithm, $N = |V(Q)|$ because we need to consider one-on-one correspondence for each vertex of the query, i.e., the graph query is performed in a *vertex-at-a-time* manner. However, if we have indexed a set of structural patterns $p_1, p_2, \ldots, p_k \subseteq Q$ where $\forall e \in E(Q), \exists p_i, s.t., e \in p_i$ $(1 \leq i \leq k)$, the graph query can be answered *pattern-at-a-time* by checking one-on-one correspondence on p_i instead $(1 \leq i \leq k)$, such that $N = k$. If $k < |V(Q)|$, we successfully reduce N to achieve our goal. Extremely, if we've indexed the query Q in advance, N is minimized to 1 and we can answer the graph query in one shot. Usually we have $1 \leq N \leq |V(Q)|$.

We then examine how to achieve the second objective by reducing $|C(v_i)|$ for all $v_i \in V(Q)$. In the baseline algorithm, every $u \in C(v_i)$ is a potential matching vertex of v_i and therefore needs to be matched temporarily for further inspection. However, a great many vertices in $C(v_i)$ have proven to be false positives eventually if the global structural constraints of subgraph isomorphism are cross-checked. So it is unnecessary to examine every vertex in $C(v_i)$ and it will be desirable if we can make use of structural patterns to help pre-prune false positives in $C(v_i)$, such that $|C(v_i)|$ can be reduced. Given a graph G and $u \in V(G)$, we consider a neighborhood induced subgraph, G_u^k, which contains all vertices within k hops away from u. This subgraph $G_u^k \subseteq G$ is referred as the *k-neighborhood subgraph* of u. We then pick structural patterns in G_u^k based on the following theorem:

THEOREM 1. *If $Q \subseteq G$ w.r.t. a subgraph isomorphism matching f, for any structural pattern $p \subseteq Q_{v_i}^k, v_i \in V(Q)$, there must be a matching pattern, denoted as $f(p) \subseteq G$, s.t. $f(p) \subseteq G_{f(v_i)}^k, f(v_i) \in V(G)$.* □

PROOF. See Appendix 9.3. □

Intuitively, if there exists a structural pattern p in the k-neighborhood subgraph $Q_{v_i}^k$ of $v_i \in V(Q)$, whereas there is no such $f(p)$ in the k-neighborhood subgraph G_u^k of $u \in C(v_i)$, we can safely prune the false positive u from $C(v_i)$, based on Theorem 1. It will be advantageous if we can index structural patterns from the k-neighborhood subgraphs of vertices in the network G before hand, such that false positives in $C(v_i)$ can be eliminated before real graph matchings. Therefore we can achieve our second objective to reduce $|C(v_i)|$. It is worth mentioning that the baseline algorithm does not consider any structural patterns but vertex labels only for indexing. It is just a special case of our pattern based indexing mechanism if we set $k = 0$.

Interestingly, the two objectives are neither independent nor conflicting with each other. By extracting and indexing structural patterns from the k-neighborhood subgraphs of vertices in the network, can we achieve both objectives effectively during graph query processing. Actually, the indexed patterns capture the local structural information within vertices' vicinity and it will be extremely useful in search space reduction.

A natural question may arise here: *Among different kinds of structural patterns, which one (or ones) are most suitable for graph indexing on large networks*? It is evident that by explicitly indexing all structural patterns within the neighborhood scope k for all vertices is of little practical use due to the exponential number of possible patterns, even when k is not set high. As a result, we need a careful selection such that our graph indexing solution lies right between the two extremes of *indexing-nothing* and *indexing-everything*. More importantly, the graph indexing structure should scale well in large networks and can achieve effective graph query performance, simultaneously.

3.3 Structural Pattern Evaluation Model

In this section, we propose a cost-sensitive model to help select the best structural patterns specifically used in large networks. Three different patterns are considered, i.e., paths, trees and graphs. For each structural pattern, we focus on two cost-sensitive aspects: 1. feature selection cost, and 2. feature pruning cost.

For a vertex $u \in V(G)$ (or $v \in V(Q)$), the *feature selection cost*, C_s, is to identify a pattern from the k-neighborhood subgraph of u (or v). The number of such patterns is denoted as n (or n'). Given a pattern p in the k-neighborhood subgraph Q_v^k of $v \in V(Q)$, the *feature pruning cost*, C_p, is to check whether there exists a pattern p' in the k-neighborhood subgraph G_u^k of $u \in C(v)$, such that $p \subseteq p'$. We further assume the vertex labels of the network G are evenly distributed, such that $|C(v)| = |V(G)|/|\Sigma|, v \in V(Q)$. Therefore the *total graph indexing cost*, C, can be formulated as a combination of (1) the total feature selection cost in G; (2) the total feature selection cost in Q; and (3) the feature pruning cost of Q, i.e.,

$$C = (|V(G)| * n + |V(Q)| * n') * C_s + \frac{|V(Q)| * |V(G)| * n' * C_p}{|\Sigma|} \quad (1)$$

Cost	$n(n')$	C_s	C_p
Path	exponential	linear time	linear time
Tree	exponential	linear time	polynomial time
Graph	exponential	linear time	NP-complete

Table 1: Qualitative Costs for Different Patterns

Given a network G, both $|V(G)|$ and $|\Sigma|$ are constant ($|V(G)|$ can be very large, though). As Q is always much smaller than G, so $|V(Q)|$ can be regarded as a small constant as well. The graph indexing cost C is therefore relevant to n, n', C_s and C_p. Table 1 shows the qualitative costs w.r.t. these parameters for different structural patterns. First, the number of patterns (n or n') can be exponentially large in the k-neighborhood subgraphs, even when k is not set high. However, the number of path patterns is usually much less than that for trees and graphs. For the feature selection cost, C_s, we can choose either BFS or DFS traversal method to identify one specific pattern in the k-neighborhood subgraph. As to the pattern pruning cost, C_p, the path containment testing takes linear time only. While for trees, a costly polynomial algorithm is required [22], and GraphQL [12] took a even more expensive semi-perfect matching method in cubic time. For graphs, though, C_p is still NP-complete because we are trying to use a set of subgraph isomorphism testings on small graphs to substitute the costly subgraph isomorphism testing on one large graph.

Based on the above analysis, paths excel trees and graphs as good indexing patterns in large networks. Although more structural information can be preserved by trees and graphs, their potentially massive size and expensive pruning cost even outweigh the advantage for search space pruning. Although theoretically the number of paths is still exponentially large in the worst case, in the remainder of the paper, we selectively use *shortest paths* for graph indexing. Shortest paths are further decomposed into a distance-wise structure, which makes our graph indexing technique, SPath, highly scalable. During graph query processing, shortest paths can be easily reconstructed and their joint pruning power proves to be very impressive.

4. SPATH

In this section, we present SPath, a path-based graph indexing technique on large networks. The principle of SPath is to use shortest paths within the k-neighborhood subgraph of each vertex of the graph to capture the local structural information around the vertex. To tackle a potentially polynomial number of shortest paths within k-neighborhood subgraphs, we further decompose shortest paths in a distance-wise structure, *neighborhood signature*, which reduces the space complexity of SPath to be linear w.r.t. the size of the network. Therefore SPath lends itself well to large networks.

4.1 Neighborhood Signature

DEFINITION 3 (**(k, l)-SET**). *Given a graph G, a vertex $u \in V(G)$, a nonnegative distance k and a vertex label $l \in \Sigma$, the (k,l)-set of u, $S_k^l(u)$, is defined as*

$$S_k^l(u) = \{v | d(u,v) = k, l(v) = l, v \in V(G)\}$$

where $d(u,v)$ is the shortest distance from u to v in G.

$S_k^l(u)$ is the set of vertices k hops away from u and having the vertex label l.

DEFINITION 4 (**k-DISTANCE SET**). *Given $u \in V(G)$, and a nonnegative distance k, the k-distance set of u, $S_k(u)$, is defined as*

$$S_k(u) = \{S_k^l(u) | l \in \Sigma\} \backslash \{\emptyset\}$$

DEFINITION 5 (**NEIGHBORHOOD SIGNATURE**). *Given $u \in V(G)$, and a nonnegative neighborhood scope k_0, the* neighborhood signature *of u, denoted as $NS(u)$, is defined as*

$$NS(u) = \{S_k(u) | k \leq k_0\}$$

$NS(u)$ maintains all k-distance sets of u from $k = 0$ (a singleton set with element $\{u\}$ only) up to the neighborhood scope $k = k_0$. Therefore, all shortest path information in the k_0-neighborhood subgraph $G_u^{k_0}$ of u is encoded in the neighborhood signature, $NS(u)$. Note we do not maintain shortest paths explicitly. Instead all paths are decomposed into the distance-wise neighborhood signature. Although extra costs have to be paid to reconstruct the exact shortest paths during graph query processing, the time spent is marginal because of the simplicity of path structures.

EXAMPLE 2. *For vertex u_1 in the network G shown in Figure 1(a), the 0-distance set $S_0(u)$ contains a unique (0, A)-set $A : \{1\}$, which contains u_1 itself. The 1-distance set $S_1(u)$ is $\{B : \{2\}, C : \{3\}\}$, and the 2-distance set $S_2(u)$ is $\{A : \{4, 6\}, B : \{5\}\}$. If the neighborhood scope k_0 is set 2, the neighborhood signature of u_1, $NS(u_1) = \{\{A : \{1\}\}, \{B : \{2\}, C : \{3\}\}, \{A : \{4, 6\}, B : \{5\}\}\}$. Similarly, for vertex v_1 in the graph query Q shown in Figure 1(b), the neighborhood signature of v_1, $NS(v_1) = \{\{A : \{1\}\}, \{B : \{2\}, C : \{3\}\}, \{C : \{4\}\}\}$*

As shortest path information within the k-neighborhood subgraph of a vertex is well preserved into its neighborhood signature, it can be used in search space pruning, i.e., the false positives in the matching candidates $C(v)$ can be eliminated before the real graph query processing, where $v \in V(Q)$. We define *neighborhood signature containment* (*NS containment* for short), which will be used for search space pruning.

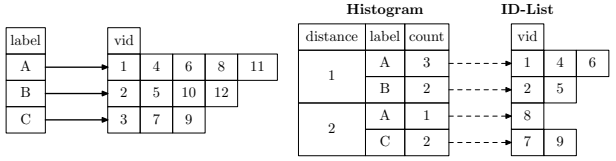
DEFINITION 6 (**NS CONTAINMENT**). *Given $u \in V(G)$ and $v \in V(Q)$, $NS(v)$ is contained in $NS(u)$, denoted as $NS(v) \sqsubseteq NS(u)$, if $\forall k \leq k_0$, $\forall l \in \Sigma$, $|\bigcup_{k \leq k_0} S_k^l(v)| \leq |\bigcup_{k \leq k_0} S_k^l(u)|$.*

THEOREM 2. *Given a network G and a query graph Q, if Q is subgraph-isomorphic to G w.r.t. f, i.e., $Q \subseteq G$, then $\forall v \in V(Q), NS(v) \sqsubseteq NS(f(v))$, where $f(v) \in V(G)$.* □

PROOF. See Appendix 9.4. □

Based on Theorem 2, for a vertex $v \in V(Q)$ and a vertex $u \in V(G)$, where $u \in C(v)$, if $NS(v)$ is not contained in $NS(u)$, denoted as $NS(v) \not\sqsubseteq NS(u)$, u is a false positive and can be safely pruned from v's matching candidates $C(v)$. Therefore, the search space is reduced.

EXAMPLE 3. *For $u_1 \in V(G)$ shown in Figure 1(a) and $v_1 \in V(Q)$ shown in Figure 1(b), their neighborhood signatures are presented in Example 2. Although $u_1 \in C(v_1)$ because u_1 and v_1 have the same label A, $NS(v_1) \not\sqsubseteq NS(u_1)$. In particular, when $l = C$, $|\bigcup_{k \leq 2} S_k^C(v_1)| = 2$ as $v_3 \in$*



(a) Global Lookup Table (b) Histogram and ID-List

Figure 2: The Global Lookup Table $\mathcal{H}$ and the Histogram and ID-List of $NS(u_3)$, $u_3 \in V(G)$ ($k_0 = 2$)

$S_1^C(v_1)$ and $v_4 \in S_2^C(v_1)$. However, $|\bigcup_{k\leq 2} S_k^C(u_1)| = 1$ as $u_3 \in S_1^C(u_1)$ only, such that $|\bigcup_{k\leq 2} S_k^C(u_1)| < |\bigcup_{k\leq 2} S_k^C(v_1)|$. So u_1 is a false positive and can be safely pruned from $C(v_1)$. By taking advantage of neighborhood signatures, we can prune the search space for $C(v_1)$ from $\{u_1, u_4, u_6, u_8, u_{11}\}$ to $\{u_6, u_8, u_{11}\}$, for $C(v_2)$ from $\{u_2, u_5, u_{10}, u_{12}\}$ to $\{u_5\}$, for $C(v_3)$ from $\{u_3, u_7, u_9\}$ to $\{u_7\}$, and for $C(v_4)$ from $\{u_3, u_7, u_9\}$ to $\{u_7, u_9\}$. The total search space size, $\prod_{i=1}^{4} |C(v_i)|$, has been reduced from 180 *to* 6.

Algorithm 2 (see Appendix 9.5) outlines the neighborhood signature containment algorithm for $v \in V(Q)$ and $u \in V(G)$, where $u \in C(v)$. Note we don't need to maintain the exact elements in $S_k^l(u)$ or $S_k^l(v)$ for NS containment testing. Instead only the cardinality information of the two sets are enough during the computation. The time complexity of Algorithm 2 is $O(k_0|\Sigma|)$, so it is a constant-time algorithm. In practice, the neighborhood signature containment testing can be executed efficiently.

4.2 SPath Implementation

Our graph indexing structure, SPath, maintains the neighborhood signature of each vertex in the network G. In practice, we further decompose neighborhood signatures into different components:

1. **Lookup Table:** We separately maintain $S_0^l(u)$, $u \in V(G)$ as a global lookup table (Note here $k = 0$): $\mathcal{H} : l^* \rightarrow \{u|l(u) = l^*\}, l^* \in \Sigma$, such that given a vertex v in the query graph, we can easily figure out its matching candidates, $C(v)$, which exactly equals $\mathcal{H}(l(v))$;

2. **Histogram:** we maintain a succinct distance-wise histogram $|S_k^l(u)|$ for $0 < k \leq k_0$ in the neighborhood signature. Based on Algorithm 2, we need not maintain the exact elements in the (k, l)-set of u, $S_k^l(u)$. Instead only the cardinality information, $|S_k^l(u)|$, is required in search space pruning. A numeric value, *count*, keeps track of $|S_k^l(u)|$ in the neighborhood signature histogram.

3. **ID-List:** We separately maintain the (k, l)-set of u, $S_k^l(u), u \in V(G)$, in an auxiliary data structure, *ID-list*, which keeps track of the exact vertex identifiers in $S_k^l(u)$.

The principle to decompose neighborhood signatures into a global lookup table, histograms and ID-lists is that both the lookup table and histograms can be maintained as a space-efficient data structure, upon which the NS containment testing can be performed without referring to the exact vertex information stored in ID-lists. Note ID-lists may be very large and only in the graph query processing phase, will ID-lists be visited to reconstruct real paths.

EXAMPLE 4. *Figure 2(a) presents the global lookup table of the network G in Figure 1(a). And Figure 2(b) illustrates the histogram and ID-list structure of the neighborhood signature of u_3 in the network G.*

To construct our graph indexing structure SPath for a network G, we need to build for each vertex $u \in V(G)$, its neighborhood signature $NS(u)$. If the neighborhood scope value k_0 is specified, a BFS traversal from u up to k_0 steps is required to collect shortest path information in the k_0-neighborhood subgraph of u. Suppose the average degree of vertices in G is d, the time complexity of building $NS(u)$ is $\sum_{i=0}^{k_0} d^i$ and the worst-case time complexity is $O(|V(G)| + |E(G)|)$. Therefore the worst-case time complexity for index construction is $O(|V(G)| * |E(G)|)$.

As to the space complexity, the global lookup table $\mathcal{H}$ takes $O(|V(G)| + |\Sigma|)$ space. Given a vertex $u \in V(G)$, the space for the histogram structure is $O(k_0|\Sigma|)$. So the total space complexity of SPath is $O(|V(G)| + |\Sigma| + k_0|\Sigma||V(G)|)$, i.e., the size of SPath grows linearly w.r.t. the network size, $|V(G)|$. Note the ID-List structure is located on the disk in that its size can be very large (the worst space complexity can be $O(|V(G)|^2)$) and it will not be used until the real path reconstruction. In practice, however, if the network is of medium size, e.g., for biological networks, we can maintain both histograms and ID-Lists in main memory to facilitate the graph query processing.

5. GRAPH QUERY PROCESSING

In this section, we will examine how graph queries are processed and optimized on large networks with the aid of SPath. Given a query graph Q, we first study how Q can be decomposed to a set of shortest paths, among which a subset of paths with good selectivity is then selected as candidates by our query plan optimizer. Q is then reconstructed and instantiated by joining the selected candidate paths until every edge in Q has been examined at least once. The major advantage of our method is its path-at-a-time philosophy in query processing and optimization, which proves to be more cost-effective and efficient than traditional vertex-at-a-time methods.

5.1 Query Decomposition

Given a query graph Q, we first compute the neighborhood signature $NS(v)$ for each $v \in V(Q)$. We then examine the matching candidates $C(v)$ by calling Algorithm 2 for NS containment testing. For $\forall u \in C(v)$, if $NS(v) \not\sqsubseteq NS(u)$, u is pruned from $C(v)$ as a false positive and the resulting matching candidates after pruning is called the *reduced matching candidates* of v, denoted as $C'(v)$.

During the NS containment testing of v w.r.t. $u \in C'(v)$, the shortest paths originated from v are generated as by-products of the neighborhood signature, $NS(v)$. Note if a path p connecting two vertices is shortest in Q, its mapping counterpart p' in the network G is not necessarily shortest between the mapping vertex-pair. We need to select the shortest paths from Q that are shortest in G as well, because only shortest paths have been indexed properly in SPath.

THEOREM 3. *For $v \in V(Q)$, a shortest path originated from v with length bounded up by k^* is guaranteed to be*

shortest as well in the k_0-neighborhood subgraph $G_u^{k_0}$ of u, where $u \in C'(v)$, if

$$k^* = \arg\min_k \Big\{ |\bigcup_{k \le k_0} S_k^l(u)| - |\bigcup_{k \le k_0} S_k^l(v)| > 0 \Big\}, \forall l \in \Sigma$$

PROOF. See Appendix 9.6. □

Based on Theorem 3, we select the shortest paths originated from v with length no greater than k^*. These paths are guaranteed to be shortest in the k_0-neighborhood subgraph of u, where $u \in C'(v)$. In the extreme case when $k^* = 0$, the shortest path is degenerated to a vertex and our graph query processing algorithm boils down to the baseline vertex-at-a-time algorithm (Algorithm 1).

EXAMPLE 5. *Consider $v_1 \in V(Q)$ in Figure 1(b) and $u_8 \in V(G)$ in Figure 1(a), because $NS(v_1) \sqsubseteq NS(u_8)$, $u_8 \in C'(v_1)$, the reduced matching candidates of v_1. When $k = 1$, $|S_1^B(v_1)| = |S_1^B(u_8)| = 1$, and $|S_1^C(v_1)| = |S_1^C(u_8)| = 1$. However, when $k = 2$, $|S_2^C(v_1)| = 1$ but$|S_2^C(u_8)| = 2$. So $k^* = 2$, and the shortest paths originated from v_1 w.r.t. u_8 are (v_1, v_2), (v_1, v_3), (v_1, v_2, v_4) and (v_1, v_3, v_4).*

5.2 Path Selection and Join

After the query graph Q has been decomposed, for each $u \in C'(v)$, it is attached with a set of shortest paths, denoted as $\mathcal{P}_u$, which can be easily looked up in SPath and will be used jointly to reconstruct Q. However, a natural question may arise: *which shortest paths should we choose in order to reconstruct Q?* To reconstruct the graph query means for every edge in Q, it should be examined at least once during the subgraph isomorphism testing such that the correctness of the query processing algorithm can be secured. So our selected shortest paths should properly "*cover*" the query, i.e., $\forall e \in E(Q)$, there should exist at least one selected shortest path p, such that $e \in p$. Furthermore, the subset of selected shortest paths should be cost-effective and help reconstruct the query Q in an efficient way. We consider two objectives in our query plan optimizer to address the path selection problem:

1. We need to choose the smallest set of shortest paths which can cover the query. This problem can be reduced to the *set-cover* problem, if every edge in $E(Q)$ is regarded as an element and every path is a subset of elements. Set-cover has proven to be NP-complete and a greedy $\log(n)$-approximation algorithm was proposed [6];

2. We need to choose shortest paths with good selectivity, such that the total search space can be minimized during real graph matching.

Let's first assume we have obtained such a subset of shortest paths which suffices for the above-mentioned objectives. Our graph query processing is then performed by joining shortest paths from among the set of selected paths.

DEFINITION 7. *Given a path $p = (v_{p_1}, v_{p_2}, \ldots, v_{p_k})$ and a path $q = (v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}})$, the join of p and q, denoted as $p \bowtie q$, is defined as an induced graph on the vertex set $\{v_{p_1}, v_{p_2}, \ldots, v_{p_k}\} \cup \{v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}}\}$, where $\{v_{p_1}, v_{p_2}, \ldots, v_{p_k}\} \cap \{v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}}\} \neq \emptyset$. The join-predicates are defined on the vertices $\{v_{p_1}, v_{p_2}, \ldots, v_{p_k}\} \cap \{v_{q_1}, v_{q_2}, \ldots, v_{q_{k'}}\}$. i.e., p and q are joinable if they share at least one common vertex.*

It is reasonable to suppose that the join cost of $p \bowtie q$ is proportional to $|C'(p)| * |C'(q)|$, where $|C'(p)| = \prod_{i=1}^{k} |C'(v_{p_i})|$ and $|C'(q)| = \prod_{j=1}^{k'} |C'(v_{q_j})|$, the multiplicity of sizes of the reduced matching candidates for each vertex in the path. If the number of join-predicates (i.e., the number of common vertices shared by both p and q) for $p \bowtie q$ is N_{pq}, and suppose N_{pq} join-predicates are mutually independent, all of which are associated with a selectivity factor θ, the remaining estimated size of $p \bowtie q$ will be $|C'(p)| * |C'(q)| * \theta^{N_{pq}}$. Given a join path $\mathcal{JP} = (((p_1 \bowtie p_2) \bowtie p_3) \bowtie \cdots \bowtie)p_t$ which covers the query Q, the total join cost can be formulated as

$$\begin{aligned} C(\mathcal{JP}) &= |C'(p_1)| * |C'(p_2)| \\ &+ |C'(p_1)| * |C'(p_2)| * \theta^{N_{p_1 p_2}} * |C'(p_3)| \\ &+ \cdots + |C'(p_1)|(\prod_{i=2}^{t-1} |C'(p_i)| \theta^{N_{p_{(i-1)} p_i}}) |C'(p_t)| \end{aligned} \quad (2)$$

In order to minimize the join cost $C(\mathcal{JP})$, we can either (1) minimize the number of join operators: $(t-1)$, which can be achieved by empirically choosing long non-repetitive paths first; Or (2) minimize the estimate size for each join operation, which can be obtained by always choosing the paths with good selectivity; Or minimize both. Note our objectives to minimize $C(\mathcal{JP})$ are almost the same as the objectives for path selection, as mentioned above. More interestingly, these objectives share the same philosophy as dictated in Section 3.3 for structural feature evaluation.

Keeping the aforementioned objectives in mind, we define *selectivity* of a path p, denoted as $sel(p)$, as follows

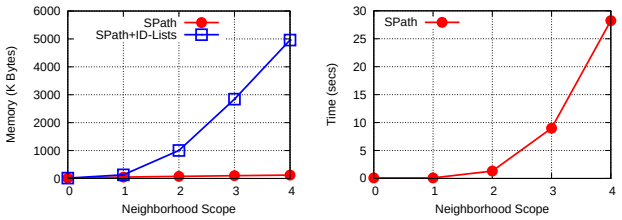
$$sel(p) = \frac{\psi(l)}{\prod_{v \in V(p)} |C'(v)|} \quad (3)$$

where $\psi(\cdot)$ is a function of the path length l, e.g., $\psi(l) = 2^l$. Intuitively, $sel(p)$ in Equation(3) tries to take both objectives into consideration. The larger the selectivity $sel(p)$ of p, the better chance p will be chosen from among the subset of shortest paths and be joined first to recover the graph query Q. In practice, our query optimizer takes a greedy approach to always pick the edge-disjoint path with highest selectivity first, and it achieves very effective query performance.

5.3 Path Instantiation

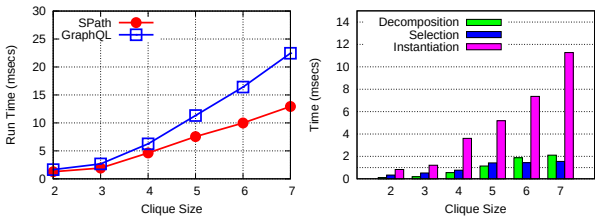
After a path has been selected, it needs to be instantiated in the network G, such that its exact matching can be determined and the join predicates can be cross-checked when path joins are executed between selected paths. We again make use of neighborhood signatures for path instantiation. Given a path $p = (v_1, v_2, \ldots, v_t)$ in the query graph Q, a straightforward way to instantiate p on G is an edge-by-edge verification for each edge $(v_i, v_{i+1}) \in p$. Specifically, for each v_i, we examine its matching candidate $u \in C'(v_i)$. If $C'(v_{i+1}) \cap S_1^{l(v_{i+1})}(u) \neq \emptyset$, it means there exist counterpart edges for (v_i, v_{i+1}) in the network G. It is worth noting that for each verification, we need to retrieve the ID-List of $S_1^{l(v_{i+1})}(u)$. If the ID-Lists reside on disk, the verification leads to expensive disk accesses, which is the most time-consuming part in graph query processing.

When selected paths are instantiated and joined with no join-predicates violation, we find one matching of Q against G successfully. The algorithm will not terminate until all matchings are detected from G. Algorithm 3 (Appendix 9.7) summarizes the whole procedure for graph query processing.



(a) Index Size (b) Construction Time

Figure 3: Index Construction Cost for SPath



(a) Average Response Time (b) Individual Time

Figure 4: Query Response Time for Clique Queries

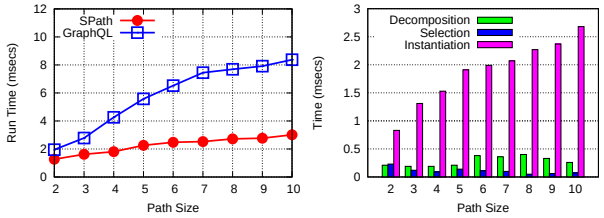
6. EXPERIMENTAL EVALUATION

In this section, we report our experimental studies to illustrate the effectiveness of SPath in graph query processing on large networks. We compare SPath with GraphQL and evaluate our algorithm SPath on both real and synthetic data sets. In the real data set, SPath proves to be a high-performance graph indexing scheme, and it achieves up to 4 times speedup in graph query processing, compared with GraphQL. In the synthetic data set which contains a set of disk-resident graphs, SPath demonstrates its scalability and effectiveness in answering graph queries in excessively large networks, whereas other proposed methods may fail in this scenario. The experimental studies on the synthetic data set are elaborated in Appendix 9.9. All our experiments were tested on an AMD Phenom 8400 Triple-Core 2.10GHz machine with 3GB memory running Ubuntu 9.04. SPath is implemented with C++ and compiled with gcc 4.3.3. We set all parameters of GraphQL as default values specified and recommended in [12]. The only parameter of SPath, i.e., the neighborhood scope k_0, is set 4, if not specified explicitly.

6.1 A Yeast Protein Interaction Network

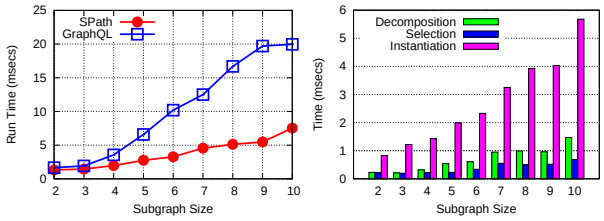
We adopt the same real data set used in GraphQL, which is a yeast protein interaction network [8]. The experimental settings are explained in Appendix 9.8.

We first consider the index construction cost for SPath on this biological network. Figure 3(a) illustrates the memory usage of SPath (in kilobytes) with the variation of the neighborhood scope, k_0. With an increase of k_0 from 0 to 4, SPath grows linearly and it takes less than 1M memory usage even when $k_0 = 4$. Note when $k_0 = 0$, only the global lookup table is built and it is the only data structure required in the baseline algorithm. When $k_0 > 0$, the histograms of neighborhood signatures in SPath are constructed in main memory as well. Figure 3(a) also presents that even the ID-Lists can be loaded in main memory and the total memory cost is less than 6M bytes. SPath proves to be very space-efficient and in the following experiments, we explicitly store ID-Lists into main memory. Figure 3(b) illustrates the runtime of index construction for SPath. Even when $k_0 = 4$, SPath can be constructed within 30 seconds.



(a) Average Response Time (b) Individual Time

Figure 5: Query Response Time for Path Queries



(a) Average Response Time (b) Individual Time

Figure 6: Query Response Time for Subgraph Queries

We then focus on the graph query processing performance in the network. We first test clique queries on the biological network. The query set contains 1,000 cliques with sizes spanning from 2 to 7. If a query has too many matchings in the network (more than 1,000), we early terminate the algorithm and show the first 1,000 matchings as output (GraphQL has the same setting to report at most 1,000 matchings of Q against G). Figure 4(a) shows the average query processing time for both SPath and GraphQL. For queries with small size (≤ 4), both methods achieve similar query processing performance. As the query size grows larger, SPath outperforms GraphQL for up to 50% improvement in query response time. Note for clique queries, the neighborhood signature of every vertex contains 1-distance set only which subsumes all other vertices. In the mean time, every clique query is decomposed to a set of edges (length-1 paths) for query processing. So the improvement mainly accounts for the optimal edge selection and join-predicate cross-checking. Figure 4(b) shows the average processing time for individual steps by varying clique sizes. The individual steps include query decomposition, abbreviated as decomposition; path selection, abbreviated as selection and path instantiation, abbreviated as instantiation. As shown in the figure, instantiation takes up the majority time during query processing.

We then test path queries on the biological network. Compared with clique queries, paths are at the other extreme of connectivity. Path queries have different sizes ranging from 2 to 10. For each size, we choose 1,000 paths and the average query processing time is examined. As illustrated in Figure 5(a), with the increase of query size, SPath achieves a speedup in graph query processing up to 4 times, compared with GraphQL. In this scenario, the neighborhood signature containment pruning takes into effect beyond the direct neighborhood scope, and the path-at-a-time matching method has proven much more effective than the traditional vertex-at-a-time approach, adopted by GraphQL. Figure 5(b) shows the average processing time for individual steps by varying path sizes. Each of the individual steps takes less time than that for clique queries, while selection spends even less because the number of possible paths selected from the

path queries is much less than that for the clique queries. Similarly, instantiation still takes up the majority time for query processing because we have to enumerate and instantiate the paths in the network.

We finally test general subgraph queries extracted from the biological network. Subgraphs are generated with sizes ranging from 2 to 10 and for each specific size, 1,000 queries are tested and the average query processing time is measured. As shown in Figure 6(a), SPath still outperforms GraphQL with a speedup for almost 4 times, especially when the query size becomes large. Figure 6(b) illustrates the individual time spent for subgraph queries and instantiation still dominates the whole graph query process for path instantiation.

7. CONCLUSIONS

In this paper, we consider the graph query problem on large networks. Existing data models, query languages and access methods no longer fit well in the large network scenario and we have presented SPath, a new graph indexing method to answer and optimize graph queries effectively on large networks. We evaluated different structural patterns based on our cost-sensitive model and shortest path information were chosen as good indexing features in large networks. Both index construction and query processing issues of SPath were discussed in detail. We performed our experimental evaluations on both real data sets and synthetic ones. The experimental results demonstrated that SPath is a scalable graph indexing technique and it outperforms the state-of-the-art GraphQL in addressing graph queries on large networks.

There are still several interesting problems left for further exploration. First of all, many large networks change rapidly over time, such that incremental update of graph indexing structures becomes important. Second, to accommodate noise and failure in the networks, we need to extend our method to support approximate graph queries as well. These interesting issues will be our new research directions in near future.

8. REFERENCES

[1] *Oracle Database 10g: Oracle Spatial Network Data Model*. Oracle Technical White Paper, 2005.

[2] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. DOGMA: A disk-oriented graph matching algorithm for RDF databases. In *Proceedings of ISWC '09*, pages 97–113, 2009.

[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of SDM '04*, 2004.

[4] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *Proceedings of SIGMOD'07*, pages 857–872, 2007.

[5] D. J. Cook and L. B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[7] F. Eichinger, K. Böhm, and M. Huber. Mining edge-weighted call graphs to localise software bugs. In *Proceedings of ECML PKDD'08*, pages 333–348, 2008.

[8] S. A. et al. Predicting protein complex membership using probabilistic network reliability. *Genome Research*, 2004.

[9] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *Proceedings of AAAI FS'06*, pages 45–53, 2006.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[11] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Proceedings of ICDE'06*, page 38, 2006.

[12] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of SIGMOD'08*, pages 405–418, 2008.

[13] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *Proceedings of SIGMOD '07*, pages 305–316, 2007.

[14] A. Hulgeri and C. Nakhe. Keyword searching and browsing in databases using BANKS. In *Proceedings of ICDE'02*, page 431, 2002.

[15] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of SIGMOD'09*, pages 813–826, 2009.

[16] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of SIGMOD'08*, pages 595–608, 2008.

[17] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of VLDB '05*, pages 505–516, 2005.

[18] B. Mckay. Practical graph isomorphism, 1981. `http://cs.anu.edu.au/~bdm/nauty/`.

[19] M. A. Nascimento, J. Sander, and J. Pound. Analysis of SIGMOD's co-authorship graph. *SIGMOD Rec.*, 32(3):8–10, 2003.

[20] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[21] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[22] R. Shamir and D. Tsur. Faster subtree isomorphism. In *Proceedings of ISTCS '97*, page 126, 1997.

[23] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.

[24] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of PODS'02*, pages 39–52, 2002.

[25] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2007.

[26] S. Trissl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of SIGMOD '07*, pages 845–856, 2007.

[27] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[28] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of SIGMOD'04*, pages 335–346, 2004.

[29] S. Zhang, M. Hu, and J. Yang. TreePi: A novel graph indexing method. In *Proceedings of ICDE'07*, pages 966–975, 2007.

[30] S. Zhang, S. Li, and J. Yang. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of EDBT'09*, pages 192–203, 2009.

[31] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta $\geq$ graph. In *Proceedings of VLDB'07*, pages 938–949, 2007.

[32] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.

9. APPENDIX

In this section, we provide related work, algorithm pseudo codes and explanations, proof sketches of the theorems, experimental setting descriptions and additional experimental results on synthetic data sets.

9.1 Related Work

The field of graph data management has seen an explosive spread in recent years because of new applications in bioinformatics, social and technological networks, communication networks, software engineering and the Web. It becomes increasingly important to manage graphs, especially large graphs, in DBMSs. However, existing database models, query languages and access methods, such as the relational model and SQL, lack native support for large graphs. The wave of graph-based applications calls for new models, languages and systems for large graph-structured networks.

Recent research has embraced the challenges of designing special-purpose graph databases. Generally, there are two distinct categories that are often referred to as the *graph-transaction setting* and the *single-graph setting*, or *network setting*. In the graph-transaction setting, a graph database consists of a set of relatively small graphs as transactions, whereas in the single-graph setting, the data of interest is a single large graph. In both settings lies a common and critical graph query problem, which can be formulated as a selection operator on graph databases and has been studied first in the theoretical literature as the subgraph isomorphism problem [9, 18, 20, 27], Subgraph isomorphism has proven to be NP-complete [10].

In a graph-transaction database, the graph query problem is to select all graphs in the database which contain the query graph as subgraph(s). The major challenge in this scenario is to reduce the number of pairwise subgraph isomorphism checkings. A number of graph indexing techniques have been proposed to address this challenge [24, 28, 11, 29, 31, 4, 23]. Different structural patterns are examined to help prune the candidate search space at the first step. Costly subgraph isomorphism checkings are verified in the second step on the pruned search space, instead of on all transactions of the graph database.

Although the graph query problem has been studied extensively in the graph-transaction setting, little attention [25, 12, 30] has been paid to improve the effectiveness of graph query processing in the single-graph setting. In this scenario, a graph query retrieves as output the complete set of occurrences of the query graph in the large network. Note the graph query problem in this setting is more general, in that a set of small graphs can be regarded as a large graph with different disconnected components. So an efficient solution in the single-graph setting will definitely help solve the graph query problem in the transaction setting. The challenge in this scenario is to accelerate the subgraph isomorphism testing itself. To develop effective and scalable techniques that address the graph query problem in the single graph setting is the focus of this paper.

A straightforward approach to managing large networks is to store the underlying graph structure in general-purpose relational tables and make use of built-in SQL queries to address the graph query problem. Oracle is currently the only commercial DBMS that provides internal support for graph data [1]. However, the relational model and SQL are fundamentally inadequate to support graph queries on large networks. Queries are translated into a large number of costly join operations and the structural knowledge of graphs is broken down and flattened during the course of database normalization. This allows little opportunity for graph specific optimizations and prevents effective pruning on the search space. The number of intermediate false positives can grow excessively large, especially when the network examined is large and diverse.

In SAGA [25], the authors proposed an approximate graph matching method, which employed a flexible graph distance model to measure similarities between graphs. However, not all exact matchings, but a subset of approximate matchings were returned as answers. In GraphQL [12], the authors made use of neighborhood subgraphs for global pruning and vertex profiles for local pruning. A search order optimizer was designed to jointly reduce the search space for each vertex in the query graph. Their experiments demonstrated that the graph-specific optimizations proposed by GraphQL outperformed an SQL-based implementation by orders of magnitude in graph query processing on large networks. GADDI [30] proposed a distance index-based matching method which were specifically used for biological networks and small social networks. The basic indexing unit is the NDS (neighboring discriminating substructure) distance for every pair of vertices in the graph. The costly frequent graph mining algorithm was adopted to help mine the discriminative subgraphs. All of the aforementioned methods have common problems: 1. they all target on pruning the search space of each vertex, such that the whole search space can be jointly reduced as much as possible. However, the query processing is still performed in a vertex-at-a-time way, which is extremely inefficient. 2. the methods proposed can only support graph queries in small networks or networks in specialized areas, whereas they cannot generalize and scale up to real large networks.

Similar graph queries were proposed on large RDF graphs as well [2]. A large RDF databases contains millions of RDF tuples (s, p, v) where s is a subject, p is a property and v is a value. Every RDF database has an associated RDF graph where vertices correspond to subjects and values, and the edges linking them are labeled with a property. A graph query expressed in SPARQL language may contain some variable vertices which can be substituted by either subject or value vertices in the RDF graph. Note a RDF graph contains vertices in two different categories: subject and value, and each vertex bears a distinct label. So RDF graphs are a special kind of networks in our study and our graph indexing method is more general and can be easily extended to answer graph queries on large RDF graphs.

Besides the graph query problem studied in this paper, other kinds of queries were proposed on large graph databases as well. Graph reachability queries [26, 16, 15] test whether there exist path connections from a vertex u to another vertex v in a large directed graph. Keyword search [14, 17, 13] over large graphs explores the graph structure and finds subgraphs that contain all the keywords in the query. Connection-preserving pattern matching queries [32] relax the subgraph isomorphism constraints by allowing two adjacent vertices in the query graph to be mapped to two vertices within distance δ in the network. It is believed that more queries of practical use, together with the corresponding query processing techniques, will be proposed and studied toward a better understanding of large networks.

Algorithm 1: Baseline Algorithm

Input: Query graph Q, Network G
Output: All subgraph isomorphism mappings f of Q against G

```
1  begin
2      for v ∈ V(Q) do
3          C(v) ← {u|u ∈ V(G), l'(v) = l(u)};
4      Recursive_Search(v_1);
5  end
6  Procedure Recursive_Search(v_i)
7  begin
8      for (u ∈ C(v_i)) and (u is unmatched) do
9          if notMatchable(v_i, u) then
10             continue;
11         f(v_i) ← u; u ← matched
12         if i < |V(Q)| then
13             Recursive_Search(v_{i+1});
14         else
15             Output a mapping f;
16         f(v_i) ← NULL; u ← unmatched;
17 end
18 Function boolean Matchable(v_i, u)
19 begin
20     for ∀ edge (v_i, v_j) ∈ E(Q), j < i do
21         if edge (u, f(v_j)) ∉ E(G) then
22             return false;
23     return true;
24 end
```

9.2 A Baseline Algorithm for Graph Query Processing

Algorithm 1 presents the baseline algorithm for graph query processing on large networks [27]. We start with finding matching candidates $C(v)$ for each vertex v in the query graph Q (line $2-3$). The matching candidates $C(v)$ is the set of vertices in the network each of which bears the same label with v, and the resulting product $\prod_{i=1}^{|V(Q)|} C(v_i)$ forms the total search space of the algorithm. The core procedure, Recursive_Search, matches v_i over $C(v_i)$ (line 11) and proceeds step-by-step by recursively matching the subsequent vertex v_{i+1} over $C(v_{i+1})$ (line $12-13$), or outputs a matching f if every vertex of Q has a counterpart in G (line 15). If v_i exhausts all vertices in $C(v_i)$ and still cannot find a feasible matching, Recursive_Search backtracks to the previous state for further exploration (line 16). Function Matchable examines the feasibility of mapping v_i to $u \in V(G)$ by considering the preservation of structural connectivity (line $18-24$). If there exist edges connecting v_i with previously explored vertices of Q but there are no counterpart edges in G, the Matchable test simply fails.

9.3 Theorem 1 Proof

Proof. Without loss of generality, we consider a subgraph g in the k-neighborhood subgraph of v, Q_v^k, $v \in V(Q)$. We first prove that for any vertex $w \in V(g)$, its mapping $f(w) \in G_{f(v)}^k$, i.e., the vertex $f(w)$ is in the k-neighborhood subgraph of $f(v)$. Because $w \in V(Q_v^k)$, there exists a path $p = v, \ldots, w$ with length $k' \leq k$ connecting v and w. Correspondingly, there exists a mapping path $f(p) = f(v), \ldots, f(w)$ with length k' connecting $f(v)$ and $f(w)$ in $G_{f(v)}^k$. So $f(w)$ is at most k' hops away from $f(v)$ (note $f(w)$ is not necessarily the shortest path between $f(v)$ and $f(w)$), i.e., $\forall w \in V(g)$, its counterpart $f(w)$ is in the k-neighborhood subgraph of $f(v)$, $G_{f(v)}^k$.

Then $\forall e = (w, x) \in E(g)$, there exists a counterpart mapping edge $e' = (f(w), f(x))$ in the k-neighborhood subgraph of $f(v)$ because $Q \subseteq G$ w.r.t. f. Therefore, $f(g) \subseteq G_{f(v)}^k$, i.e., the mapping graph $f(g)$ is in the k-neighborhood subgraph of $f(v)$. □

Algorithm 2: Neighborhood Signature Containment

Input: $NS(v), v \in V(Q)$, $NS(u), u \in V(G)$
Output: If $NS(v) \sqsubseteq NS(u)$, return **true**; Otherwise, return **false**

```
1  begin
2      for l ∈ Σ do
3          Count[l] ← 0;
4      for k ← 1 to k_0 do
5          for l ∈ Σ do
6              Count[l] ← Count[l] + |S_k^l(u)|
7              if |S_k^l(v)| > Count[l] then
8                  return false;
9              Count[l] ← Count[l] − |S_k^l(v)|
10     return true;
11 end
```

9.4 Theorem 2 (NS Containment) Proof

Proof. For $\forall v \in V(Q)$, we consider an arbitrary vertex $v' \in S_k^l(v)$, where $0 \leq k \leq k_0$ and $l \in \Sigma$. If $k_0 = 0$, then $k = 0$ and $l = l'(v)$. However, $l'(v) = l(f(v))$ because $Q \subseteq G$ w.r.t. f, where $l'(\cdot)$ and $l(\cdot)$ are labeling functions of Q and G, respectively. So $|S_0^l(v)| = |S_0^l(f(v))| = 1$.

We then consider the situations when $0 < k \leq k_0$. In Q, there must be a shortest path $p = v \ldots v'$ of length k with v and v' as its endpoints. because $Q \subseteq G$ w.r.t. f, the counterpart mapping path $f(p) = f(v) \ldots f(v')$, where $f(p) \subseteq G$, can be either (1) the shortest path between $f(v)$ and $f(v')$, such that $f(v') \in S_k^l(f(v))$; or (2) a non-shortest path between $f(v)$ and $f(v')$, because there must be another path p', $|p'| < k$, connecting $f(v)$ and $f(v')$ in G. This is true if some edges in p' cannot be mapped from any edge in Q. If so, $\exists \bar{k}, 0 < \bar{k} < k$ such that $f(v') \in S_{\bar{k}}^l(f(v))$, i.e., $f(v')$ appears in the $(\bar{k}, l)$-set of $f(v)$. Based on the two aforementioned situations, $\forall v' \in S_k^l(v)$, $\exists \bar{k}, 0 < \bar{k} \leq k$, such that $f(v') \in S_{\bar{k}}^l(f(v))$. So $|\bigcup_{k \leq k_0} S_k^l(v)| \leq |\bigcup_{k \leq k_0} S_k^l(f(v))|$ satisfies. □

9.5 NS Containment Algorithm

Algorithm 2 outlines the neighborhood signature containment testing for $v \in V(Q)$ and $u \in V(G)$, where $u \in C(v)$. We maintain a hash table, *Count*, to keep track of the value of $(|\bigcup_{k \leq k_0} C_k^l(u)| - |\bigcup_{k \leq k_0} C_k^l(v)|)$ for all $l \in \Sigma$. In real implementation, only the labels either in k_0-neighborhood subgraph of v or in k_0-neighborhood subgraph of u (or both) are examined. The time complexity of Algorithm 2 is $O(k_0|\Sigma|)$, so Algorithm 2 is a constant-time algorithm.

Algorithm 3: SPath Based Graph Query Processing

Input: Graph Query Q, Network G
Output: All matchings f of Q against G, s.t. $Q \subseteq G$

```
1  begin
2      for v ∈ V(Q) do
3          C'(v) ← {u | NS(u) ⊒ NS(v), u ∈ C(v)};
4          for u ∈ C'(v) do
5              P_u ← {p | |p| ≤ k*} based on Theorem 3;
6      v* ← arg min_v |C'(v)|
7      for u ∈ C'(v*) do
8          p_u ← arg min_p sel(p), p ∈ P_u;
9          I ← ∅;
10         Recursive_Search(p_u, I);
11 end
12 Procedure Recursive_Search(p_u, I)
13 begin
14     while i_{p_u} ← Next_Instantiation(p_u) do
15         if not Joinable(I, (p_u, i_{p_u})) then
16             continue;
17         I ← I ∪ {(p_u, i_{p_u})};
18         if ∀e ∈ E(Q) has been covered by I then
19             Output a matching f ← ⋃_{p_u} {i_{p_u} | p_u ∈ I};
20         else
21             p_{u'} ← arg min_p sel(p), p ∈ P_{u'}, where u' is
               any vertex covered by the paths in I so far;
22             Recursive_Search(p_{u'}, I);
23     I ← I − {(p_u, i_{p_u})} ;
24 end
25 Function boolean Joinable(I, (p_u, i_{p_u}))
26 begin
27     for ∀ path p_i ∈ I do
28         if p_i and p_u are joinable w.r.t. the join predicate
           set J in Q, but i_{p_i} and i_{p_u} fails the
           corresponding join predictates from J in G then
29             return false;
30     return true;
31 end
```

9.6 Theorem 3 Proof

PROOF. Note k^* is the minimum distance with which the (k,l)-set of u begins to differ from the (k,l)-set of v. We prove the theorem by contradiction. Assume there exists a shortest path $p = v \dots v_k$ in the neighborhood subgraph of v, and the length of p equals $k \leq k^*$. Assume the vertex label of v_k is l, so $v_k \in C^l_k(v)$. However, the counterpart path $p' = u \dots u_k$ in the neighborhood subgraph of u is no longer a shortest path, $u \in C'(v)$. Equivalently, $\exists k', 0 \leq k' < k$, s.t., $u_k \in C^l_{k'}(u)$. However, for $0 \leq k' < k \leq k^*$, we have $|\bigcup_{i \leq k'} S^l_i(u)| - |\bigcup_{i \leq k'} S^l_i(v)| = 0$. So we must have another vertex $\hat{v}$ in the neighborhood subgraph of v, s.t., $\hat{v} \in C^l_{k'}(v)$. It means that both v_k and $\hat{v}$ in the neighborhood subgraph of v have the counterpart vertex u_k in the neighborhood subgraph of u. So $NS(v) \not\sqsubseteq NS(u)$. It contracts with the fact that $u \in C'(v)$, i.e., $NS(v) \sqsubseteq NS(u)$. □

9.7 Graph Query Processing Algorithm

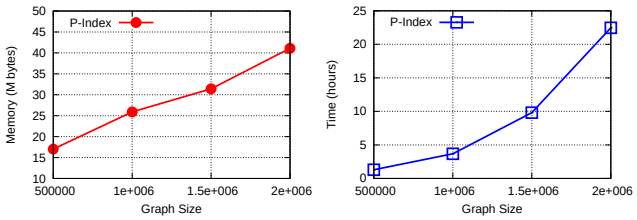
Our SPath based graph query processing is presented in Algorithm 3. It starts with a preprocessing step by pruning the search space for each vertex v in the graph query Q with neighborhood signature containment testings illustrated in Theorem 2 (Line 3). For each $u \in C(v)$, if $NS(u) \not\supseteq NS(v)$, it will be eliminated from $C(v)$ as a false positive. As a result, we get the reduced matching candidates, $C'(v)$. For each matching candidate $u \in C'(v)$, the set of shortest paths, $\mathcal{P}_u$, is generated simultaneously (Line $4-5$). Based on Theorem 3, all the paths in $\mathcal{P}_u$ with length no greater than k^* are guaranteed to be shortest as well in the k_0-neighborhood subgraph of u. We choose a vertex v^* with a minimal size of the reduced matching candidates as a starting point for graph query processing (Line 6). When all possible matching candidates $u \in C'(v^*)$ have been explored, our graph query processing algorithm terminates and all matchings of Q against G will be figured out. Among the set of shortest paths in $\mathcal{P}_u$, an optimal path is selected based on Equation 3 to initiate the recursive search (Line $7-10$). I is a data structure which maintains the pairs of (*path, instantiated_path*) discovered so far.

In Recursive_Search, we first instantiate the path p_u in the network G by calling the function Next_Instantiation(p_u) (Line 14). In practice, for each vertex in the path p_u to be instantiated, we maintain an iterator to keep track of the vertex oriented in the network G. Function Next_Instantiation(p) is called to manipulate iterators such that a new instantiation of p_u is enumerated in a pipelining manner. We then test *joinability* between the newly instantiated path with all the previously instantiated paths in I by calling the Joinable function (Line 15). Joinable function checks the join predicates between the path p_u and every path $p_i \in I$ in the query Q (Line $27-30$). If their corresponding matching paths i_{p_u} and i_{p_i} fail in any join-predicate verification in the network G, we have to explore the next instantiation of p_u (Line 16), or backtrack to the previously examined path (Line 23). Otherwise, if p_u and every path $p_i \in I$ are joinable, p_u is coupled with the instantiated path i_{p_u} in the network G. If every edge in the query graph Q has been covered by some paths in I, a matching f is found out as an output (Line $18-19$). Otherwise, we proceed by picking another path with the best selectivity for further inspection (Line $21-22$). The optimal path selected is from among the set of shortest paths $\mathcal{P}_{u'}$ where u' is any vertex having been explored so far in I. In practice, a maximum priority queue is maintained to get the path with highest selectivity. If two (or more) paths have the same highest selectivity, ties are broken by always picking the one which overlaps least with previously selected paths. When a new path p is selected, for any vertex $u \in p$, all shortest paths $\mathcal{P}_u$ pertaining to u are added in the priority queue for further selection.

9.8 Experimental Settings for the Real Data Set

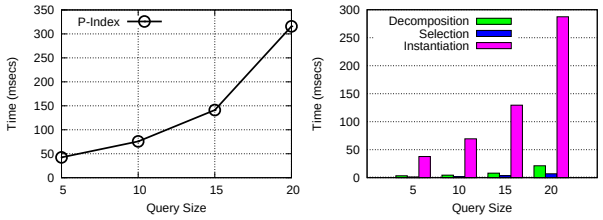
The yeast protein interaction network consists of $3,112$ vertices and $12,519$ edges. Each vertex represents a unique protein and each edge represents an interaction between proteins. It is worth mentioning that the traditional RDBMS based query processing method is extremely inefficient to support graph queries on this biological network [12].

We further add Gene Ontology (GO) information as vertex labels to the proteins. The GO is a hierarchy of categories that describes cellular components, biological processes, and molecular functions of genes and their products (proteins). Each GO term is a node in the hierarchy and



(a) Index Size (b) Index Construction Time

Figure 7: Index Construction for SPath



(a) Average Response Time (b) Individual Time

Figure 8: Query Response Time for Subgraph Queries in the Synthetic Graph

has one or more parent GO Terms, and each protein may have one or more GO terms. The original GO terms in the yeast protein interaction network consist of $2,205$ distinct labels. We relax these GO terms by using their highest level ancestors. There are 183 such highest level GO terms in total, which constitutes our vertex label set Σ.

As to the graph queries, GraphQL suggests two extreme kinds of graphs with totally different structures: *cliques* and *paths*. For biological networks, the clique structure corresponds to protein complexes, while the path structure corresponds to transcriptional or signaling pathways. We further extract general induced subgraphs by randomly choosing seeds in the network and traversing the network in a DFS fashion. These generated graphs can be thought of as general queries with arbitrary structures lying in the middle of the two extremes: path and clique.

9.9 Synthetic Disk-resident Graphs

We further evaluate our algorithm, SPath, on a series of disk-resident synthetic graphs based on the *Recursive Matrix* (R-MAT) model [3]. The graphs generated naturally follows the power-law in- and out-degree distributions. All the parameters of the graph generator are specified with default values suggested by the authors. For SPath, we maintain both the global lookup table and the histogram structures of neighbor signatures in main memory, while keeping all the ID-Lists on disk.

We first examine the index construction cost of SPath on different large networks. We generate four large networks with $|V(G)| = 500,000, 1,000,000, 1,500,000$ and $2,000,000$, and $|E(G)| = 5*|V(G)|$, respectively. For each graph generated, the vertex labels are drawn randomly from the label set Σ, where $|\Sigma| = 1\% * |V(G)|$. Note in a typical modern PC, the number of potential tree or graph indexing structures can be excessively large, which requires a storage in the terabyte or even peta-byte order. In this scenario, GraphQL fails simply because it cannot scale up on these large networks. However, as shown in Figure 7(a), SPath scales linearly with an increase of the network size, which makes SPath a feasible graph indexing solution applicable in large networks. Figure 7(b) illustrates the index construction time for SPath. Note building SPath from the network is a pre-processing step and executes only once before the real graph query processing, so the cost is still affordable for large networks.

We then test the query processing performance of SPath on one synthetic graph G with size $|V(G)| = 1,000,000$ and $|E(G)| = 5,000,000$. We further generate subgraph queries with different sizes 5, 10, 15, and 20 by randomly extracting induced subgraphs from G by DFS traversal. For each specific query size, we generate $1,000$ queries and measure the average query response time. As shown in Figure 8(a), SPath can achieve satisfactory response time even when the query size is large. However, all previously proposed methods, including GraphQL, cannot answer graph queries on this massive network. Figure 8(b) shows the individual time spent by different query processing components of SPath. As both decomposition and selection are performed in main memory, they take up little time during query processing. However, instantiation needs to retrieve ID-lists from disk, so it becomes the leading factor and potential bottleneck for graph queries on large networks.

9.10 Acknowledgement

The authors would like to thank Dr. Huahai He and Prof. Ambuj Singh for giving us access to GraphQL; and all the anonymous reviewers for their helpful comments and suggestions.

The work was supported in part by the NSF IIS-09-05215, and by the U.S. Army Research Laboratory under Cooperative Agreement No. W911NF-09-2-0053 (NS-CTA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.