An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases

Jinsoo Lee*
School of Computer Science and Engineering
Kyungpook National University, Korea
jslee@www-db.knu.ac.kr

Wook-Shin Han*†
School of Computer Science and Engineering
Kyungpook National University, Korea
wshan@knu.ac.kr

Romans Kasperovics*
School of Computer Science and Engineering
Kyungpook National University, Korea
romans.kasperovics@gmail.com

Jeong-Hoon Lee
School of Computer Science and Engineering
Kyungpook National University, Korea
jhlee@www-db.knu.ac.kr

ABSTRACT

Finding subgraph isomorphisms is an important problem in many applications which deal with data modeled as graphs. While this problem is NP-hard, in recent years, many algorithms have been proposed to solve it in a reasonable time for real datasets using different join orders, pruning rules, and auxiliary neighborhood information. However, since they have not been empirically compared one another in most research work, it is not clear whether the later work outperforms the earlier work. Another problem is that reported comparisons were often done using the original authors' binaries which were written in different programming environments. In this paper, we address these serious problems by re-implementing five state-of-the-art subgraph isomorphism algorithms in a common code base and by comparing them using many real-world datasets and their query loads. Through our in-depth analysis of experimental results, we report surprising empirical findings.

1. INTRODUCTION

Many complex objects, such as chemical compounds, social networks, and biological structures are modeled as graphs. Many real applications in bioinformatics, chemistry, and software engineering require efficient and effective management of graph structured data.

One of most important graph queries in graph databases is the subgraph isomorphism query. That is, given a query q and a data graph g, find all embeddings of q in g. This problem belongs to NP-hard [10] and has many important applications, such as searching chemical compound databases, querying biological pathways, and finding protein complexes in protein interaction networks.

Ullmann [14] proposes the first practical algorithm for subgraph isomorphism search for graphs. It is a backtracking algorithm which finds solutions by incrementing partial solutions or abandoning them when it determines they cannot be completed. In recent years, many algorithms such as VF2 [2], QuickSI [11], GraphQL [5], GADDI [17], and SPath [18] have been proposed to enhance the Ullmann algorithm. These algorithms exploit different join orders, pruning rules, and auxiliary information to prune out false-positive candidates as early as possible, thereby increasing performance.

Figure 1 shows the reported comparisons of the state-of-the-art subgraph isomorphism algorithms. Here, a directed edge depicts reported superiority. For example, according to [18], SPath is superior to GraphQL. Note that, in [17], GADDI is compared with TALE [13] which finds only *approximate* embeddings.

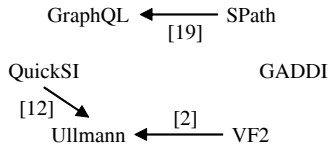


Figure 1: Comparisons of state-of-the-art graph isomorphism algorithms.

We observe serious problems in the current practice of experiments for these algorithms: 1) It is difficult to compare each algorithm since they have not been described in a common framework; 2) They have not been compared empirically in most research work. Only four comparisons as depicted in Figure 1 were reported. Thus, it is not clear whether the later work outperforms the earlier work; 3) The reported comparisons were done by comparing the binary executables provided by the original authors, ignoring a number of factors that heavily influence the performance (e.g., programming languages, implementer's programming skills, main memory vs. disk, and buffer size, etc.).

In order to address these problems, we re-implement all representative subgraph isomorphism algorithms in a common framework using C++. For this purpose, we use best-effort re-implementations based on the original papers and

*The first three authors contributed equally to this work.
†Corresponding author

Proceedings of the VLDB Endowment, Vol. 6, No. 2

on email communications with the original authors, since we were unable to acquire the source code of any technique except VF2 from its authors. However, since GraphQL is implemented in Java, we exploit a java bytecode analyzer to fully understand the original implementation. We also perform extensive experiments using many real datasets and their query workload and provide an in-depth analysis. We note that a similar experience has been reported in iGraph [3], which compares only graph *indexing* techniques rather than the subgraph isomorphism algorithms themselves.

Our contributions can be summarized as follows.

- We clearly explain the differences of existing algorithms using a common framework in Section 3.
- We re-implement five state-of-the-art algorithms (VF2, QuickSI, GraphQL, GADDI, and SPath) in the common code base.
- We fairly and empirically compare these algorithms using many real and synthetic datasets in Section 4.
- We analyze experiments in depth in order to understand why one algorithm outperforms another for specific query and data graphs in Section 4
- We report surprising findings through our analysis: 1) QuickSI designed for handling small graphs often outperforms the more recent algorithms GraphQL, GADDI, and SPath which are designed for handling large graphs. 2) QuickSI, VF2, and GADDI fail to find embeddings in trees in a reasonable time, showing exponential behavior. 3) GraphQL is the only method to process all query sets tested but shows slower performance than QuickSI in many query sets and datasets. 4) It should be noted that in this paper, unlike in [18], SPath is almost consistently slower than GraphQL. This is mainly because a) they are implemented in different programming languages, and b) the cost of reading signatures from *disk* is not taken into account. Note that SPath is implemented in C++, while GraphQL is implemented in Java. This strongly indicates that, unless both methods are implemented in a common framework, empirical comparisons would be useless. 5) We find that all existing algorithms have problems in their join order selections for some datasets, although GraphQL processes all queries we test in reasonable times. The blind computation of signatures of GraphQL regardless of queries and data sets incurs significant performance overhead compared with QuickSI. This calls for new subgraph algorithms combining the strengths of both algorithms.

The remainder of this paper is organized as follows. Section 2 reviews the background information as well as existing work on subgraph isomorphism. Section 3 presents the details of our implementations. Section 4 presents the results of performance evaluation. Section 5 summarizes and concludes our paper.

2. BACKGROUND

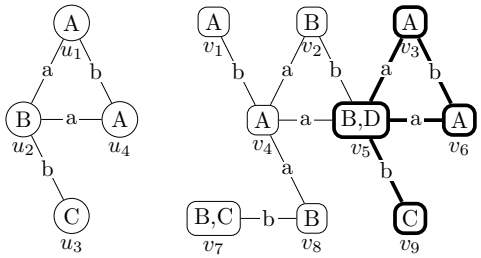
2.1 Problem Definition

All datasets and query sets used in [2, 5, 11, 17, 18] are modeled as *undirected labeled graphs*. An undirected labeled graph g is defined as a triple (V, E, L) where V is the set of vertices, E ($\subseteq V \times V$) is the set of undirected edges, and L is a labeling function which maps a vertex or an edge to a set of labels or a label, respectively. Without loss of generality, all subgraph isomorphism algorithms can be easily extended to handle graphs whose edges have a set of labels. Unless otherwise specified, we use symbols q, g, u, and v to denote a query graph, a data graph, a query vertex, and a data vertex, respectively.

Given a query graph $q = (V, E, L)$, a data graph $g = (V', E', L')$, a *subgraph isomorphism* (or an *embedding*) is an injective function $M : V \rightarrow V'$ such that (1) $\forall u \in V$, $L(u) \subseteq L'(M(u))$, and (2) $\forall (u_i, u_j) \in E$, $(M(u_i), M(u_j)) \in E'$, and $L(u_i, u_j) = L'(M(u_i), M(u_j))$.

Problem Definition 1. [5,14,18] *Given a query graph q and a data graph g, the subgraph isomorphism problem is to find all distinct embeddings of q in g.*

Figure 2 shows an example of query q and data graph g. We have two embeddings M_1 and M_2 for this subgraph isomorphism query.



(a) query graph q. (b) data graph g.

$M_1 = \{(u_1, v_3), (u_2, v_5), (u_3, v_9), (u_4, v_6)\}$
$M_2 = \{(u_1, v_6), (u_2, v_5), (u_3, v_9), (u_4, v_3)\}$

(c) resulting subgraph isomorphisms.

Figure 2: Example of query and data graphs.

In practice, we may stop the subgraph isomorphism search after the first k embeddings are found. In [5,18], k is set to 1000.

2.2 Basic Concepts

We explain the concept of the induced subgraph, the partial subgraph isomorphism, the adjacency set, and the k-neighborhood.

A graph q' is an *induced subgraph* of a graph q iff $\mathrm{V}(q') \subseteq \mathrm{V}(q)$, and $\mathrm{E}(q')$ contains only the edges present in q, i.e., $\mathrm{E}(q') = \mathrm{E}(q) \cap (\mathrm{V}(q') \times \mathrm{V}(q'))$. Note that an induced subgraph q' of q can be defined solely with its vertex set $\mathrm{V}(q')$. For example, the induced graph of the vertices v_3, v_5, v_6, v_9 is marked with bold lines in Figure 2.

Let g' be an induced subgraph of g, and let M' be a subgraph isomorphism from g' to g. We call M' a *partial solution* when searching all subgraph isomorphisms from g' to g if $\mathrm{V}(g') \subset \mathrm{V}(g)$ (as opposed to a *complete solution* such that $\mathrm{V}(g') = \mathrm{V}(g)$).

The *adjacency set* of a vertex v of a graph g, denoted as $\mathrm{adj}(v)$, is a set of vertices directly connected (adjacent) to v. The *k-neighborhood* of a vertex v of a graph g, denoted as $\mathrm{N}_k(v)$, is a set of vertices of g where for each vertex v' in $\mathrm{N}_k(v)$, the shortest distance between v' and v is less than or equal to k. That is, $\mathrm{N}_k(v)$ includes v itself.

Consider the vertex v_4 in the data graph g from Figure 2(b). The adjacency set $\mathrm{adj}(v_4)$ is $\{v_1, v_2, v_5, v_8\}$. The 1-neighborhood $\mathrm{N}_1(v_4)$ is $\{v_1, v_2, v_4, v_5, v_8\}$. The 2-neighborhood $\mathrm{N}_2(v_4)$ contains all vertices in the data graph g.

2.3 Related Work

We can classify existing algorithms into two categories depending on whether or not they use exact search: (1) *exact* subgraph matching and (2) *approximate* subgraph matching. Each matching has its own important target applications.

Exact subgraph matching algorithms can also be classified into two subcategories depending on the usage of graph indexing techniques. The first subcategory includes GraphGrep [12], gIndex [16], FG-Index [1], Tree+Δ [19], gCode [20], SwiftIndex [11], and C-Tree [4]. These indexing algorithms are based on a two-step filter-and-refine strategy where the filtering step uses graph indexes to minimize the number of candidate graphs, and the refinement step checks if there exists *one* subgraph isomorphism for each candidate. The second subcategory includes Ullmann [14], VF2 [2], QuickSI [11], GraphQL [5], GADDI [17], and SPath [18]. These algorithms find *all* embeddings for a given query graph and a data graph. We will detail each algorithm in the following section. We note that Ullmann, VF2, and QuickSI have been originally designed for handling *small* graphs while GraphQL, GADDI, and SPath have been originally designed for handling *large* graphs.

Approximate subgraph matching algorithms find approximate embeddings with their own similarity measures. Representative algorithms in this area include TALE [13], SIGMA [9], and Ness [6].

3. IMPLEMENTATION

In this section, we describe how we implement the five state-of-the-art algorithms in a common framework. For this purpose, we introduce a generic subgraph isomorphism algorithm so that each algorithm can be implemented by extending this generic algorithm according to its specifics.

3.1 Generic Subgraph Isomorphism Algorithm

The generic subgraph isomorphism algorithm is implemented as a backtracking algorithm [7] which finds solutions by incrementing partial solutions or abandoning them when it determines they cannot be completed.

Algorithm 1 shows a generic subgraph isomorphism algorithm, GenericQueryProc. Its inputs are a query graph q and a data graph g, and its output is a set of subgraph isomorphisms (or embeddings) of q in g. Here, to represent an embedding, we use a list M of pairs of a query vertex and a corresponding data vertex.

For each vertex u in q, GenericQueryProc first invokes FilterCandidates to find a set of candidate vertices $C(u)$ ($\subseteq$ V(g)) such that L(u) $\subseteq$ L(v) (Line 3). Note that we place logical expressions in double square brackets to show the *necessary* post-conditions for each subroutine. If $C(u)$ is empty, we can safely exit, making early termination possible (Line 5). After that, GenericQueryProc invokes a recursive subroutine, SubgraphSearch, to find mapping pairs of a query vertex and matching data vertices at a time (Line 8). Note that SubgraphSearch of SPath matches one query path at a time for each recursive call.

SubgraphSearch Subroutine

SubgraphSearch takes as parameters a query graph q, a data graph g, and a partial embedding M and reports all embeddings of q in g.

The recursion stops when the algorithm finds the complete solution (i.e., when $|M| = |V(q)|$) (Line 1). Otherwise, the algorithm calls NextQueryVertex to select a query vertex $u \in$ V(q) which is not yet matched (Line 4). After that, it calls RefineCandidates to obtain a refined candidate vertex set C_R from $C(u)$ by using algorithm-specific pruning rules (Line 5). Next, for each candidate data vertex $v \in C_R$ such that v is not matched yet, the IsJoinable subroutine checks whether the edges between u and already matched query vertices of q have corresponding edges between v and already matched data vertices of g (Line 7). If v is qualified, it is matched to u, and SubgraphSearch updates status information by calling UpdateState (Line 9), and the algorithm proceeds to match the remaining query vertices of q by recursively calling SubgraphSearch (Line 10). Next, all changes done by UpdateState are restored by calling RestoreState (Line 11). The algorithm terminates when all possible embeddings are found.

Algorithm 1 GenericQueryProc

Input: query graph q
Input: data graph g
Output: all subgraph isomorphisms of q in g

```
 1: M := ∅;
 2: for each u ∈ V(q) do
 3:    C(u) := FilterCandidates (q, g, u, ...);
       [[ ∀v ∈ C(u)((v ∈ V(g)) ∧ (L(u) ⊆ L(v))) ]]
 4:    if C(u) = ∅ then
 5:       return;
 6:    end if
 7: end for
 8: SubgraphSearch (q, g, M, ...);
Subroutine SubgraphSearch (q, g, M, ...)
 1: if |M| = |V(q)| then
 2:    report M;
 3: else
 4:    u := NextQueryVertex (...);
       [[ u ∈ V(q) ∧ ∀(u', v) ∈ M(u' ≠ u) ]]
 5:    C_R := RefineCandidates (M, u, C(u), ...);
       [[ C_R ⊆ C(u) ]]
 6:    for each v ∈ C_R such that v is not yet matched do
 7:       if IsJoinable (q, g, M, u, v, ...) then
 8:          [[ ∀(u', v') ∈ M((u, u') ∈ E(q) ⟹
                    (v, v') ∈ E(g) ∧ L(u, u') = L(v, v')) ]]
 9:          UpdateState (M, u, v, ...);
             [[ (u, v) ∈ M ]]
10:          SubgraphSearch (q, g, M, ...);
11:          RestoreState (M, u, v, ...);
             [[ (u, v) ∉ M ]]
12:       end if
13:    end for
14: end if
```

The SPath algorithm grows partial solutions with one path at a time rather than a vertex at a time. Thus, although our generic recursive algorithm accommodates the characteristics of SPath, we will explain SPath separately in Section 3.7 for ease of understanding.

Common Graph Storage

Depending on the size of a data graph, we store the graph as a tuple in a heap file or a large object in a BLOB file as in iGraph. We also use a B+-tree to efficiently find a data graph using a graph ID.

For each subgraph isomorphism algorithm, we tune the disk representation of a data graph in order to support fast retrieval and construction of its main memory data structures. In subsequent subsections, we describe data structures for each method.

3.2 Ullmann Algorithm

FilterCandidates: FilterCandidates returns a set of data graph vertices with a matching label u.

NextQueryVertex: NextQueryVertex returns one vertex at a time from the vertices in the order they appear in the input. It is clear that the performance of the Ullmann algorithm highly depends on the input order of the query vertices. We will describe this issue in detail when we describe the NextQueryVertex function of VF2.

RefineCandidates: RefineCandidates prunes out all candidate vertices $v \in C(u)$ that have a smaller degree than u.

IsJoinable: IsJoinable iterates through *all* adjacent query vertices of u. If the adjacent query vertex u' is already matched, i.e., $(u', v') \in M$, then it checks whether there is a corresponding edge (v, v') in the data graph. Note that, since IsJoinable is called in a most inner loop, we must carefully design this function. If there is no edge between u and already matched query vertices, we can optimize this process by skipping this checking process. Such optimization will be explained in IsJoinable of QuickSI.

UpdateState, RestoreState: UpdateState appends a pair (u, v) to M while RestoreState restores M by removing the pair (u, v) from M.

In the following algorithms, we describe only the subroutines which are different from those in Ullmann.

3.3 VF2 Algorithm

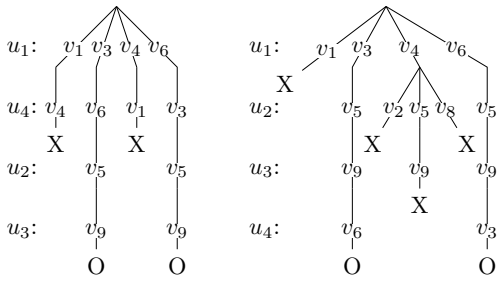
NextQueryVertex: Unlike Ullmann, VF2 starts with the first vertex and selects a vertex *connected* from the already matched query vertices. Note that the original VF2 algorithm does not define any order in which query vertices are selected.

RefineCandidates: VF2 uses the following three pruning rules to prune out data vertex candidates: (1) Prune out any vertex v in $C(u)$ such that v is not connected from already matched data vertices; (2) Let M_q and M_g be a set of matched query vertices and a set of matched data vertices, respectively. Let C_q and C_g be a set of adjacent and not-yet-matched query vertices connected from M_q and a set of adjacent and not-yet-matched data vertices connected from M_g, respectively. Let adj(u) be a set of adjacent vertices to a vertex u. Then, prune out any vertex v in $C(u)$ such that $|C_q \cap \text{adj}(u)| > |C_g \cap \text{adj}(v)|$; (3) prune out any vertex v in $C(u)$ such that $|\text{adj}(u) \setminus C_q \setminus M_q| > |\text{adj}(v) \setminus C_g \setminus M_g|$.

For example, consider again the query graph q and the data graph g from Figure 2. Suppose that the current partial solution $M = \{(u_1, v_4)\}$ and that u_2 is the next vertex returned by NextQueryVertex with $C(u_2) = \{v_2, v_5, v_7, v_8\}$. Then, $M_q = \{u_1\}$, $M_g = \{v_4\}$, $C_q = \{u_2, u_4\}$, $C_g = \{v_1, v_2, v_5, v_8\}$. The RefineCandidates subroutine prunes out v_7 using the pruning rule (1), because v_7 is not connected to any vertex in M_g. The subroutine also prunes out v_8 from $C(u_2)$ using the pruning rule (2), since $\text{adj}(u_2) \cap C_q = \{u_4\}$ and $\text{adj}(v_8) \cap C_g = \{\}$. The subroutine prunes out v_2 from $C(u_2)$ using the pruning rule (3) since $\text{adj}(u_2) \setminus C_q \setminus M_q = \{u_3\}$ and $\text{adj}(v_2) \setminus C_g \setminus M_g = \{\}$.

Improvements

We note that the matching order driven by NextQueryVertex significantly impacts the query performance by reducing the size of the recursive call tree. For instance, consider the query and data graphs from Figure 2. If we match query vertices in order (u_1, u_2, u_3, u_4), SubgraphSearch of the generic algorithm is called 14 times (Figure 3(b)). However, if we match query vertices in order (u_1, u_4, u_2, u_3), SubgraphSearch is called 12 times (see Figure 3(a)). Note that in both cases, at least eight recursive calls are necessary to output two complete solutions.



(a) matching order (u_1, u_4, u_3, u_4).

(b) matching order (u_1, u_2, u_3, u_4).

Figure 3: Recursion trees using the generic subgraph isomorphism algorithm for the query and data graphs in Figure 2.

The original VF2 version used in iGraph uses a reordering technique which sorts *query* vertices by the frequency of the query vertex label and then feeds these reordered query vertices as input to VF2. This technique could be effective when the reordered vertex sequence is similar to the one ordered by the frequency of the *data* vertex label. We also optimize the original VF2 version in several ways: 1) On comparing labels of two vertices, we directly compare the label IDs (i.e., integer comparison) of those vertices instead of calling expensive virtual function calls. 2) By exploiting the inverse vertex label list, we accelerate the search performance of finding vertices having a given vertex label. 3) When returning from each recursive call, C_q and C_g must be restored. By maintaining additional stacks, this process can also be accelerated efficiently. By putting these optimizations including the reordering technique all together, our VF2 version outperforms the original by up to 29.86 times.

Disk Representation

VF2 represents a graph using three structures: 1) *vertex label list* that allows access to the ordered vertex label list of a vertex by a given ID (see Figure 4(a)); 2) *inverse vertex label list* that allows access to the ordered vertex ID list by a given vertex label (see Figure 4(b)); and 3) *adjacency lists* (see Figure 4(c)) of each vertex which store adjacency information, i.e., a list of pairs (vertex ID, edge label) ordered by the vertex ID. Note that we materialize the inverse vertex label list in the graph database for speedup, although it can be constructed from the vertex label list.

3.4 QuickSI Algorithm

NextQueryVertex: QuickSI tries to access vertices having infrequent vertex labels and infrequent, adjacent edge labels as early as possible. Specifically, instead of using label frequency information from a query graph as in VF2, QuickSI pre-processes data graphs to compute the frequencies of vertex labels and the frequencies of a triple (source vertex label, edge label, target vertex label). By using the computed edge label frequencies, we assign a weight to each query edge and obtain a minimum spanning tree using a modified Prim algorithm. QuickSI creates a sequence by using the order in which the vertices are inserted into the

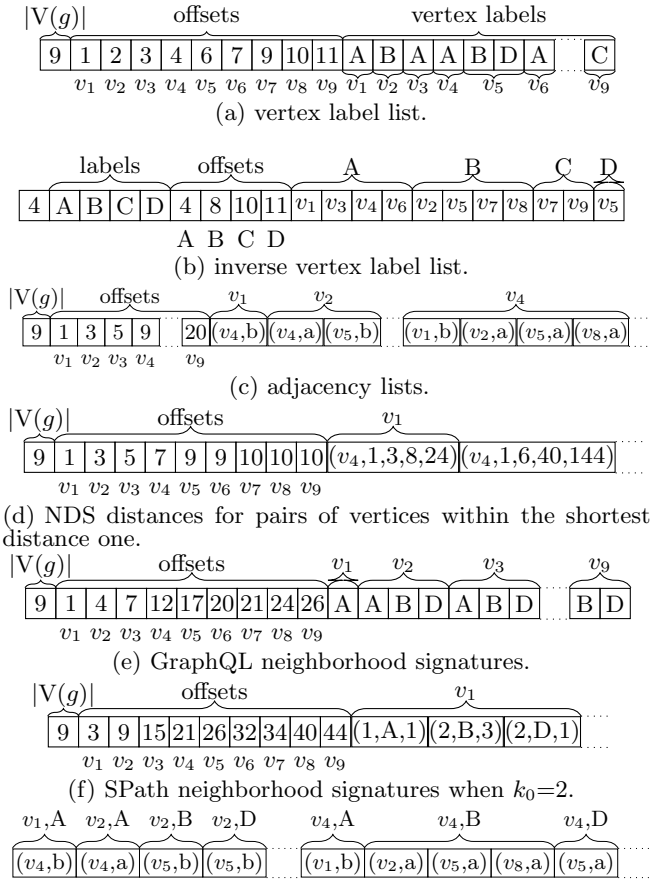


Figure 4: Disk representation of the data graph from Figure 2.

minimum spanning tree. When the algorithm selects a starting edge (u_1, u_2), the algorithm uses u_1 as the first vertex in the sequence if the vertex label frequency of u_1 is lower than that of u_2. Otherwise, u_2 is used as the first vertex. For detailed explanation, we refer readers to [11].

RefineCandidates: For the first query vertex u, QuickSI does not refine u. For the subsequent vertices u returned by NextQueryVertex, let u_{par} be the parent vertex of u in the minimum spanning tree and v' be the matching data vertex of u_{par}. Then, QuickSI prunes out v in $C(u)$, if there is no edge between v and v'.

IsJoinable: Unlike IsJoinable of Ullmann which blindly iterates through *all* adjacent query vertices of u, IsJoinable of QuickSI iterates through adjacent and already matched query vertices of u.

Although this important property is not elaborated in [11], our empirical analysis shows that this mechanism contributes to speedups of QuickSI, making the invocation cost of the SubgraphSearch of QuickSI the lowest among all five algorithms.

Disk Representation

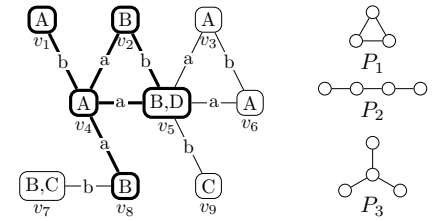
QuickSI stores a graph in the same format as the VF2 algorithm. In addition, QuickSI uses two B+-trees, one B+-tree for storing all distinct vertex labels along with their frequencies and the other for storing all distinct triples (source vertex label, edge label, target vertex label) along with their frequencies.

3.5 GADDI Algorithm

Before explaining each subroutine specific to GADDI, we first need to understand the concept of the *neighboring discriminating substructure* (NDS) distance.

The NDS distance between v_1 and v_2 using a subgraph P, denoted as $\Delta_{\text{NDS}}(v_1, v_2, P)$, is defined as the number of embeddings of P in an induced subgraph of $\text{N}_k(v_1) \cap \text{N}_k(v_2)$. Noth that k is a given parameter of GADDI.

Figure 5 displays the data graph g from Figure 2 with the induced subgraph of $\text{N}_2(v_1) \cap \text{N}_2(v_4)$ drawn with thicker lines. For the substructures P_1, P_2, and P_3 shown in the same figure, the NDS distances are calculated as follows: $\Delta_{\text{NDS}}(v_1, v_4, P_1) = 3$; $\Delta_{\text{NDS}}(v_1, v_4, P_2) = 8$; and $\Delta_{\text{NDS}}(v_1, v_4, P_3) = 24$.



$\Delta_{\text{NDS}}(v_1, v_4, P_1) = 3$, $\Delta_{\text{NDS}}(v_1, v_4, P_2) = 8$, $\Delta_{\text{NDS}}(v_1, v_4, P_3) = 24$

Figure 5: The NDS distances between v_1 and v_4 when $k = 2$.

Now, we explain how GADDI selects substructures for calculating NDS distances. For this purpose, GADDI samples 100 pairs of vertices from the data graph, and for each pair (v_1, v_2), we construct an induced *unlabeled* graph[1] of $\text{N}_k(v_1) \bigcap \text{N}_k(v_2)$. For those induced graphs, [17] suggests selecting top ten frequent subgraphs using a frequent subgraph mining algorithm such as gSpan [15]. We construct a matrix L where each row corresponds to an induced graph g, and each column corresponds to a subgraph selected P, and each entry in L corresponds to the number of embeddings of P in g. We select the three columns (i.e., subgraphs) that have the largest numbers of distinct values as substructures.

NextQueryVertex: GADDI first selects a query vertex appearing first in the input and then performs a depth-first search to find next query vertices.

RefineCandidates: RefineCandidates of the GADDI algorithm prunes out v in $C(u)$, if, for each query vertex $u' \in \text{N}_k(u)$, there is no data vertex $v' \in \text{N}_k(v)$ satisfying the following three conditions: 1) $\text{L}(u') \subseteq \text{L}(v')$; 2) for each P_i in a given set of substructures, $\Delta_{\text{NDS}}(u', u, P_i) \leq \Delta_{\text{NDS}}(v', v, P_i)$; and 3) the shortest distance between u' and u is greater than or equal to the shortest distance between v' and v.

UpdateState, RestoreState: GADDI uses an additional pruning technique that reduces the candidate sets of all query vertices. For each data vertex v' in $\text{N}_k(v)$, if there does not exist a query vertex u' in $\text{N}_k(u)$ satisfying the three conditions used in RefineCandidates, we can prune v' from all the current candidate sets except $C(u)$. Therefore, UpdateState identifies those prunable data vertices and additionally removes them from all $C(u_i)$s except $C(u)$ ($1 \leq i \leq |\text{V}(q)|$), and RestoreState additionally restores the removed vertices.

[1]We have communicated with one of the original authors to learn how to construct such induced graphs.

Improvements

We use the reordering technique which is used in the improved VF2 version to reduce the size of the recursive call tree of SubgraphSearch.

According to [17], subgraphs selected for computing NDS distances are connected, unlabeled graphs having three or four edges. Thus, there are only eight different connected graphs having three or four edges. By exploiting this important fact, we can safely skip the expensive frequent subgraph mining.

Disk Representation

GADDI stores a graph in the same representation as it does for the VF2 algorithm. In addition, for each pair of data vertices of a graph, we store the precomputed NDS and the shortest distances along with the graph. Specifically, for each pair (v_i, v_j) of vertices $(i < j)$ within a given shortest distance, we store an entry storing the shortest distance and three NDS distances. These entries are ordered by the IDs of the first and the second vertices, and we use binary search to locate the distances between the given pair of vertices. Figure 4(d) shows a fragment of the disk representation of the data graph g in Figure 2 storing the precomputed shortest distance and three NDS distances for the pairs of vertices within the distance of one. In this fragment, v_1 has just one vertex v_4 within distance one from v_1, and the corresponding entry stores the NDS distances from Figure 5.

3.6 GraphQL Algorithm

FilterCandidates: The GraphQL algorithm uses two additional pruning rules to reduce the size of the candidate sets: 1) neighborhood signature based pruning and 2) the pseudo subgraph isomorphism test based pruning.

We first explain the concept of the *GraphQL neighborhood signature* of a vertex v, denoted as $\text{sig}_{\text{GraphQL}}(v)$. $\text{sig}_{\text{GraphQL}}(v)$ is a multiset of labels of $\text{adj}(v)$. For example, $\text{sig}_{\text{GraphQL}}(u_2)$ in Figure 2 is $\{A, A, C\}$, and $\text{sig}_{\text{GraphQL}}(v_2)$ is $\{A, B, D\}$.

The neighborhood signature based pruning prunes out a candidate vertex v if $\text{sig}_{\text{GraphQL}}(u) \nsubseteq \text{sig}_{\text{GraphQL}}(v)$. For example, assume that u is u_2 (Line 3 of Algorithm 1). Then, v_2 is pruned since $\text{sig}_{\text{GraphQL}}(u_2) \nsubseteq \text{sig}_{\text{GraphQL}}(v_2)$.

Now, we explain the concept of the pseudo isomorphism test, which is an iterative algorithm using the depth d as a parameter. At first iteration, we obtain two breadth first search trees T_u and T_v for u and v respectively, where their depth is 1(i.e., $d = 1$). Then, we can prune out v if T_u is not contained in T_v. We can iterate this process by increasing d by one until $d = r$, where r is called the *refinement level*. For detailed explanation, we refer readers to [5].

NextQueryVertex: The GraphQL algorithm finds next query vertices by using a greedy strategy similar to the heuristic-based join order optimization based on the cardinality of intermediate joined results. NextQueryVertex of GraphQL first selects a query vertex u which has the smallest candidate set size $|C(u)|$. In the subsequent calls, NextQueryVertex returns a query vertex u that is connected already matched query vertices and that makes the smallest size of intermediate results.

Improvements

For graphs having labeled edges, we extend GraphQL neighborhood signature of v with labels of the adjacent edges to v to improve pruning power. For instance, $\text{sig}_{\text{GraphQL}}(v_2)$ from Figure 2 becomes {(a,A), (b,B), (b,D)}.

Disk Representation

GraphQL stores a graph using an inverse vertex label list (see Figure 4(b)), an adjacency list(Figure 4(c)), and a GraphQL neighborhood signature (Figure 4(e)). GraphQL does not use the vertex label list, since it can construct candidate vertex lists during the neighborhood signature based pruning. Note that GraphQL materializes GraphQL neighborhood signatures of all vertices for a data graph, which is more efficient than constructing such signatures on the fly during query processing. Figure 4(e) illustrates the disk representation of GraphQL signatures of the data graph from Figure 2.

3.7 SPath Algorithm

The SPath algorithm also uses the GenericQueryProc, though it invokes a different SubgraphSearch in order to match a query *path* rather than a query vertex per recursive call. Thus, it may minimize the depth of the recursion tree by matching a path per call [18]. However, as we will see in our experiments, the selection order of paths significantly impacts the query performance, and the selection order of SPath is far from optimal for most data sets.

FilterCandidates: Similar to the GraphQL algorithm, SPath uses neighborhood signatures to minimize candidate sets. However, it attempts to exploit more neighborhood information. The *SPath neighborhood signature* of a given vertex u, denoted as $\text{sig}_{\text{SPath}}(u)$, is a set of triples where each triple (d, l, c) is constructed from the vertices in $\text{N}_{k_0}(u)$. The triple (d, l, c) represents the fact that there are c vertices in $\text{N}_{k_0}(u)$ containing the label l such that the shortest distance from u is d. Here, k_0 is called the *neighborhood scope*. For instance, if $k_0 = 1$, the signature of vertex v_7 in Figure 2 is $\text{sig}_{\text{SPath}}(v_7) = \{(1, B, 1)\}$. If $k_0 = 2$, the signature of the same vertex v_7 is $\text{sig}_{\text{SPath}}(v_7) = \{(1, B, 1), (2, A, 1)\}$.

Now, we explain how to use the SPath neighborhood signature. We first define a function $S_d^l(v)$ to represent the containment relationship between two SPath neighborhood signatures. If there exists a triple (d, l, c) in $\text{sig}_{\text{SPath}}(v)$, $S_d^l(v) = c$. Otherwise, $S_d^l(v) = 0$. Next, we define a rule for pruning data vertices using the SPath neighborhood signature. For the given signatures $\text{sig}_{\text{SPath}}(u)$ and $\text{sig}_{\text{SPath}}(v)$ of query vertex u and data vertex v, we can prune out the data vertex v in $C(u)$, if it does not satisfy the following condition: for all $k \leq k_0$ and all possible labels l in $\text{sig}_{\text{SPath}}(u)$, $\sum_{i=1}^{k} S_i^l(u) \leq \sum_{i=1}^{k} S_i^l(v)$. For example, assume that u is u_3 (Line 3 of Algorithm 1) and that $k_0 = 2$. Here, the signature of u_3 is $((1, B, 1), (2, A, 2))$. We can prune v_7, since $\sum_{i=1}^{2} S_i^A(u_3)(= 2) > \sum_{i=1}^{2} S_i^A(v_7)(= 1)$.

Unlike [18] suggesting a large neighborhood scope ($k_0 = 4$), we observe that, if we increase the neighborhood scope of SPath, the performance can decrease. This phenomenon is explained as follows. A larger neighborhood scope increases filtering power, but also increases the size of SPath neighborhood signature and the filtering time. The optimal neighborhood scope, which is difficult to choose, lies in a balance between filtering power and filtering time.

The following subroutine shows SubgraphSearch of SPath. Note that the neighborhood scope k_0 is used as an additional parameter in order to limit the radius of the SPath neighborhood signature of a vertex.

The subroutine stops this recursion when the algorithm finds the complete solution (i.e., when $|M| = |V(q)|$) (Line 1). Otherwise, the subroutine calls NextQueryPath to se-

lect a next query *path* p_q whose length is shorter than or equal to k_0, and whose vertices except the first vertex are not yet matched. (Line 4). After that, it calls GetCandidatePaths to obtain all data paths matching the query path p_q in the data graph g (Line 5). Next, for each candidate data path $p_g \in P$, the IsJoinable subroutine checks whether the edges between the vertices in the p_q and already matched query vertices of q have corresponding edges between the vertices in the p_g and already matched data vertices of g (Line 7). Note that the first vertex of p_q is already matched, so all the resulting candidate data paths should start from the matched data vertex. If p_g is qualified, it is matched to p_q, and SubgraphSearch updates the partial solution M by calling UpdateState (Line 8), and the algorithm proceeds to match the remaining query paths of q by recursively calling SubgraphSearch (Line 9).

```
Subroutine SubgraphSearch (q, g, M, k0)
 1: if |M| = |V(q)| then
 2:    report M;
 3: else
 4:    p_q := NextQueryPath (q, g, k0);
 5:    P := GetCandidatePaths (p_q, M, C);
 6:    for each p_g ∈ P do
 7:       if IsJoinable (p_q, p_g, M) then
 8:          UpdateState (p_q, p_g, M);
 9:          SubgraphSearch (q, g, M, k0);
10:          RestoreState (p_q, p_g, M);
11:       end if
12:    end for
13: end if
```

Now, we explain the subroutine NextQueryPath further, which is most important in query performance of SPath.

NextQueryPath: The SPath algorithm first selects a query vertex u which has the smallest candidate set size $|C(u)|$. In the subsequent call, SPath returns the most selective path which starts from an already matched query vertex. Here, the selectivity function $sel(p)$ for a given path p is calculated as $\frac{2^{|\mathrm{V}(p)|}}{\prod_{u \in \mathrm{V}(p)} |C(u)|}$, where V($p$) denotes all vertices in a path p. Note that the denominator represents the join cardinality of the candidate sets for all query vertices in p. However, this overestimate leads to significant errors in estimating the join cardinality. Thus, in many query and datasets, SPath is bound to choose a suboptimal join order.

Disk Representation

Like the GraphQL algorithm, SPath stores only the inverse vertex label list for accessing vertices by label and does not store the vertex label list, which is constructed in memory from the inverse vertex label listduring the SPath neighborhood signature based pruning. Each SPath neighborhood signature for the data vertex v_i is stored as a list of triples (d, l, c) ordered by d and l (Fig. 4(f)). As a distinction from all other methods, SPath adjacency lists are grouped by labels of target vertices. For example, in Fig. 4(g), the adjacency list of the vertex v_4 is split into three sublists: 1) the adjacent vertices with label A ((v_1, b)), 2) the adjacent vertices with label B ($(v_2, a), (v_5, a), (v_8, a)$), and 3) the adjacent vertices with label D ((v_5, a)).

4. EXPERIMENTS

In this section, we evaluate the performance of the five representative subgraph isomorphism algorithms VF2 [2], QuickSI [11] (in short, QSI), GraphQL [5] (in short, GQL), GADDI [17] (in short, GAD), and SPath [18] (in short, SPA). We use best-effort reimplementations based on the original papers and on email communications with the original authors. As for GraphQL, we re-implement and optimize it in C++ by analyzing the original implementation using a java bytecode analyzer.

Datasets: We use four real datasets referred to here as AIDS, NASA, Yeast, and Human. The AIDS dataset was used in [11]. The Yeast dataset was used in [5, 18]. The NASA dataset was produced from a popular XML dataset [8] used in the XML research field. Since the Human dataset used in [17] could not be obtained from the original authors, we re-created it using the same process described in [17], which is straightforward. We also experimented with the synthetic datasets used in [5, 17]. However, there were no significant differences in overall performance trends of the algorithms, and thus, we omit them to save space.

Note that these datasets have different characteristics. The AIDS dataset contains a set of *sparse* graphs where the average graph size is small (= 27.4 in terms of the number of edges), and the number of unique labels is small (=51). The NASA dataset contains a set of *trees* where the average tree size is 32.2, and the number of unique labels is much larger (=117,302) than AIDS. The Yeast dataset contains only one large graph having 3112 vertices and 12519 edges. A vertex corresponds to a protein, and an edge corresponds to an interaction between two proteins. Since one protein can appear in several cellular components and biological processes, a vertex can have multiple labels. This graph is *denser* than graphs in AIDS and NASA. The Human dataset contains a large graph modeling a protein interaction network of 4675 proteins and 86282 interactions, which is larger and denser than Yeast. This graph has a fewer number of labels and a larger average degree than Yeast. Table 1 summarizes the properties of the four datasets.

Table 1: Summary of the real-world datasets.

Dataset	AIDS	NASA	Yeast	Human
# of graphs	10000	36790	1	1
# of vertices	2∼214	2∼889	3112	4675
# of edges	1∼217	1∼888	12519	86282
Avg. degree	1.95	1	8.05	36.82
Max. degree	11	245	168	771
# of distinct v. labels	51	117302	184	90
# of distinct e. labels	4	0	0	0
Avg. # of labels per vertex	1	1	7.55	4.63

Query sets: For AIDS, we use the existing query sets which are currently downloadable together with the iGraph framework[2]. There are six query sets. Each query set contains 1000 query graphs of the same size in terms of the number of edges. The query sizes are 4, 8, 12, 16, 20, and 24 edges. To generate queries of size 24, we randomly select connected subgraphs of size 24 from data graphs. We then generate the other query sets so that the following containment relationship is satisfied. That is, a small size query graph q of size s is constructed from a large size query graph q' of size $s + 4$ by removing edges until q' is still connected and contains s edges. For the other datasets, we generate query sets by using this query generation process. In addition, we use two types of query sets for Yeast and Human which were used in [5]. These query sets were generated with a tool provided

[2]http://www.igraph.or.kr/

by the GraphQL authors and do not satisfy the containment relationship.

Suppose that a large query graph q is a supergraph of a small graph q'. Then, the number of embeddings for q is typically smaller than or equal to the number of embeddings for q'. Thus, the performance of a *good* subgraph isomorphism algorithm would decrease as we increase the query size. In contrast, the search space for a graph q exponentially increases as we increase the query size. If a query q has n vertices, u_1, u_2, $\cdots$, u_n, then the size of its search space becomes $|C(u_1)|\times|C(u_2)|\times\cdots\times|C(u_n)|$. Thus, the performance of a good subgraph isomorphism algorithm is likely to decrease as we increase the query size by exploiting powerful pruning rules and optimal join orders.

Setup: For all experiments, we use a PC with Intel Xeon Quad Core 2.27GHz , 8 GB of main memory, and 1 TB 7200 RPM hard disk, running Windows Vista. We implement all algorithms in the iGraph framework using C++ and compile them using Microsoft Visual-C++ compiler. We run all algorithms with best possible parameter values for each dataset. We stop subgraph isomorphism search after the first 1000 subgraph isomorphisms are found, as it was done in [5, 18]. The size of the database buffer is set to 500 MB, which allows the once read data from the database to be kept in main memory.

We use the number of I/Os and the elapsed time as the performance metrics for database construction. We use the average elapsed time and the average number of SubgraphSearch calls (# of recursive calls) as the performance metrics for query execution. Specifically, if there is more than one data graph in a database, the elapsed time for a query means the accumulated elapsed time for processing all data graphs for that query. Thus, we average the total elapsed times for all queries to calculate the average elapsed time. Note that each algorithm has the significantly different cost of a SubgraphSearch call.

We divide our experimental section into four parts for each dataset and give more details on each dataset in the corresponding subsections.

4.1 AIDS Dataset

4.1.1 Database Construction

Table 2 shows the database sizes and database building times for all five algorithms using the AIDS dataset. Note that the number of read I/Os for all algorithms is 588. SPath was run with the best neighborhood scope $k_0 = 2$. The GADDI algorithm was run with the shortest distance $d = 1$ and the neighborhood scope $k_0 = 2$. Setting $d = 1$ and $k_0 = 2$ produced the best performance results as in [17].

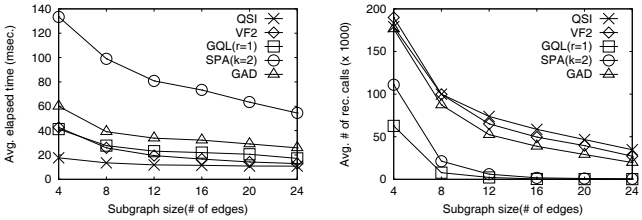
The VF2 database has the smallest disk space, and its building time is the fastest, since the VF2 algorithm does not use any auxiliary information. QuickSI has the same disk representation as VF2 along with two additional B+-trees to store the frequencies of 51 unique vertex labels and 240 unique edge features. However, the space of the frequency information of QuickSI for the AIDS dataset is negligible. Thus, the database sizes of both algorithms are almost the same. However, in QuickSI, for each data graph, we first count unique vertex labels and unique edge features and update the frequencies in the corresponding B+-trees, resulting in 33582 + 60214 updates to both B+-trees. Thus, the building time for QuickSI is 1.61 times slower than VF2. GraphQL and SPath use additional neighborhood signatures which are pre-computed and stored together with graphs, although they do not use the vertex label list as explained in Section 3.6. Since the AIDS dataset has a small average degree, the database of GraphQL is only 1.36 times larger than that of VF2. However, the database of SPath is 2.31 times larger than that of VF2 due to a larger signature size than GraphQL. GADDI stores pre-computed NDS and shortest distances between data vertices. The size of these distances is comparable to that of SPath signature.

Table 2: Database size and construction time for AIDS dataset.

Alg.	Size(MB)	# of total I/Os	Time(msec.)
VF2	9.25	1774	418
QuickSI	9.28	1780	671
GraphQL	12.59	2202	684
SPath($k_0 = 2$)	21.40	3330	1420
GADDI	21.47	3339	9064

4.1.2 Query Processing

Figure 6 shows the subgraph isomorphism search performance using the AIDS dataset. SPath was run with the best signature neighborhood scope $k_0 = 2$, which is contradictory to [18] suggesting a large neighborhood scope ($k_0 = 4$). However, in this experiment, GraphQL was run with the best refinement level $r = 1$. The GADDI was run with the shortest distance $d = 1$ and the neighborhood scope $k_0 = 2$.



(a) average elapsed time. (b) avg. # of recursive calls.

Figure 6: AIDS dataset.

The average number of recursive calls for every algorithm decreases as we increase the query size. Thus, all algorithms behave well despite their inherent exponential time complexity by choosing good join orders and by exploiting effective pruning rules. In terms of the average number of recursive calls, the ranking order is GraphQL, SPath, GADDI, VF2, and QuickSI. GraphQL and SPath exploit signature-based pruning before calling SubgraphSearch, and thus, the size of the candidates (i.e., the search space) is the smallest. Note that the number of recursive calls for GraphQL is 6.7 and 6.4 smaller than those for QuickSI and VF2, respectively.

However, in terms of the average elapsed time, the ranking order is completely different. QuickSI is the fastest algorithm. VF2, GraphQL, GADDI and, SPath are 1.73, 1.99, 2.88, and 6.62 times slower than QuickSI on average, respectively. We analyze this surprising phenomenon in depth as follows: Although the average number of recursive calls for VF2 is 1.05 times smaller than QuickSI, the average cost of recursive calls for VF2 is 2.82 times larger than that of QuickSI. This is due to QuickSI's optimized design of IsJoinable as we pointed out in Section 3.4. In GraphQL, the filtering time spent for GraphQL neighborhood signature based filtering and pseudo-isomorphism based pruning constitutes 63.16% of the total elapsed time while the filtering time for QuickSI is zero. Note that the filtering is executed before starting to call SubgraphSearch. In addition,

the average cost of recursive calls for GraphQL is 1.94 times larger than VF2. This is because 1) the query optimization time of GraphQL constitutes 10.6% of SubgraphSearch since the sizes of query graphs are relatively large compared with the sizes of data graphs, and 2) GraphQL does not use any pruning rule for RefineCandidates.

In SPath, the filtering time spent for SPath neighborhood signature based filtering constitutes 61.23% of the total elapsed time. The average cost of recursive calls for SPath is 5.18 times larger than that of GraphQL due to the overhead in path-based matching. Furthermore, the larger size of SPath neighborhood signature, compared to the GraphQL neighborhood signature leads to slower reading times for accessing data graphs. In GADDI, for each query, calculating NDS distances for every pair of vertices of the query graph is very expensive. The average cost of recursive calls for GADDI is 3.76 times larger than that of GraphQL since the data graphs are sparse, and thus the NDS distances are often equal to zero, which does not help to refine candidates. Note that, unlike GraphQL and SPath, there is no additional filtering step for GADDI.

4.2 NASA Dataset

4.2.1 Database Construction

Table 3 shows the database size and database building times for all five algorithms using the NASA dataset. Note that the number of read I/Os for all algorithms is 3,077. The size of the database for QuickSI is 1.34 times large than that of VF2, since the sizes of the two B+-trees storing the frequency information become large due to the large number of unique labels (= 117,302). The size of the database for SPath is 5.85 times larger than VF2 since the value of neighborhood scope k_0 is set to four instead of two. Note that setting k_0 to four shows the best query performance for NASA. The increasing rate of GADDI over VF2 in NASA in terms of the database size and the elapsed time is smaller than that in AIDS since the average degree of NASA is smaller than that of AIDS. The GADDI was run with the shortest distance $d = 1$ and the neighborhood scope $k_0 = 2$.

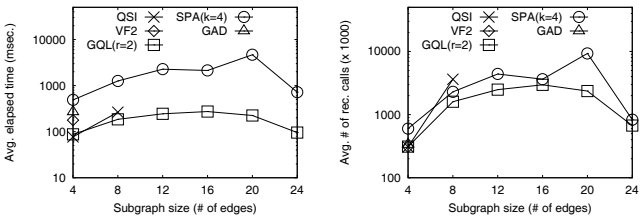
Table 3: Database size and construction time for NASA dataset.

Alg.	Size(MB)	# of total I/Os	Time(msec.)
VF2	44.04	8718	1697
QuickSI	58.99	10634	3769
GraphQL	57.65	10461	1950
SPath(k_0=4)	257.68	36089	21737
GADDI	95.33	15288	32167

4.2.2 Query Processing

Figure 7 shows the subgraph isomorphism search performance using the NASA dataset. SPath was run with the best signature neighborhood scope $k_0 = 4$, and GraphQL was run with the best refinement level $r = 2$. The GADDI was run with the shortest distance $d = 1$ and the neighborhood scope $k_0 = 2$.

The purpose of this experiment is to see if those five algorithms exploit this important characteristics of the dataset. VF2,QuickSI, and GADDI show exponential behaviors at the sizes of query sets 8, 12, and 8, respectively. Only GraphQL and SPath complete their query execution in a reasonable time. This striking phenomenon is due to the serious problems both in their join order selection and in the absence of signature-based pruning. To verify our claim,



(a) average elapsed time. (b) avg. # of recursive calls.

Figure 7: NASA dataset.

we first changed the join orders of QuickSI with those of GraphQL for some slow queries, and this modified version of QuickSI completed these queries in a reasonable time. We note that QuickSI uses statistics for label frequencies for *all* data graphs in the database while GraphQL and SPath use statistics for label frequencies for each data graph. Thus, GraphQL and SPath can generate a better join order than QuickSI for these queries. However, for the other slow queries, especially, those having star-shaped subgraphs with many same labeled vertices, although QuickSI uses the join orders of GraphQL, it still shows exponential behavior. Note that QuickSI compares one query vertex with one data vertex at a time, and thus, it tries to explore all combinations of query vertices and data vertices of the same label for such star-shaped subgraphs. However, GraphQL and SPath can efficiently prune such subgraphs if query vertex signatures are not contained in the corresponding data vertex signatures. Thus, both GraphQL and SPath efficiently process such star-shaped queries, significantly reducing the number of candidates. This indicates that we need to combine good join order strategies with signature-based pruning for the NASA query/data sets.

It should be noted that, at the query sizes 20, unlike GraphQL, SPath shows jumps both in the number of recursive calls and the elapsed time. This is due to serious problem in its join order selection. However, at the query size 24, we observe that signature-based pruning significantly prune candidate sets. For example, $\sum_{u \in V(q)} |C(u)|$ for the 61st query of size 24 is 71.01 times smaller than $\sum_{u \in V(q)} |C(u)|$ for the 61st query of size 20. Thus, in spite of the inherent problem in join order selection of SPath, the elapsed time of SPath significantly is reduced for the query size 24 by exploiting the signature-based pruning of SPath.

The average number of recursive calls for GraphQL increases as we increase the query size up to 16. This is due to search space increments. However, when we increase the query size further, the average number of recursive calls decreases. This phenomenon is explained as follows. As we increase the size of a query, the search space also increases. However, when a larger query has vertexes having infrequent labels, the signature-based pruning of GraphQL significantly prunes candidates, thereby significantly reduce the average number of recursive calls.

In terms of the average elapsed time, the performance of GraphQL is 10.38 times faster than SPath on average since 1) the loading time including in-memory signature construction time for GraphQL is 5.55 times faster than that for SPath, 2) the filtering time for GraphQL is 2.45 times faster than that for SPath since SPath uses $k_0 = 4$, and 3) the total cost of recursive calls for GraphQL is 14.33 times smaller than that for SPath. Note that setting r to two in GraphQL filters the number of candidates very effectively. In order to

compensate for the lack of the additional filtering step using pseudo-subgraph isomorphism of the GraphQL algorithm, we have to use a larger neighborhood scope (k_0=4) for the SPath algorithm. Thus, $\sum_{u \in V(q)} |C(u)|$ of SPath is 1.21 times smaller than that of GraphQL. However, due to the overhead of computing a larger SPath neighborhood signature, the filtering time of SPath is 2.45 times larger than that of GraphQL.

4.3 Yeast Dataset

In this experiment, we use three types of query sets: subgraph, clique, and path. The query sets of the first type are generated as is done for AIDS. The other types of query sets are provided by the original authors of GraphQL. The query sets of the second type contain clique queries (complete graphs) that correspond to protein complexes. The queries of the last type correspond to biological pathways. The vertices of the last two types are randomly assigned with the top 40 most frequent vertex labels, and each query graph vertex has only one label. Note that, unlike subgraph queries, path and clique queries do not satisfy the containment relationship since they are randomly generated.

4.3.1 Database Construction

Table 4 shows the database sizes and database building times for all five algorithms using the Yeast dataset. The SPath database is built with the best neighborhood scope $k_0 = 1$. That is, a larger neighborhood scope (i.e., $k_0 > 1$) decreases the query performance of SPath due to the overhead of larger SPath neighborhood signatures, which is contrary to [18] suggesting a larger neighborhood scope for better query performance. The GADDI was run with the shortest distance $d = 1$ and the neighborhood scope $k_0 = 2$.

Note that the number of read I/Os for all algorithms is 33. The size of the database for QuickSI is 2.58 times large than that of VF2, because of the large size of the B+-tree storing the frequencies. In the Yeast dataset, each vertex has 7.55 labels on average. Therefore the number of all distinct triples (source vertex label, edge label, target vertex label) which we need to store in a B+-tree can increase quadratically[3]. The size of the database for SPath is 9.02 times larger than that of VF2 since the number of triples (d, l, c) in the SPath neighborhood signature and the number of adjacency lists in the SPath adjacency list increase with the average number of vertex labels. The increasing rate of GADDI over VF2 in terms of the average elapsed time is much larger than that for the AIDS dataset since the number and the cost of NDS distance calculations increase due to the larger degree of Yeast. Note that the cost of calculating an NDS distance depends on the size of the neighborhood. The increasing rate of GraphQL over VF2 in Yeast in terms of the database size and elapsed time is larger than that of AIDS because of the larger average degree and multiple labels for each vertex.

4.3.2 Subgraph Queries

The purpose of this experiment is to analyze the trends in query performance for Yeast using the same query generation technique as for AIDS and NASA datasets. We used ten subgraph query sets with sizes ranging from one to ten. These query sets have the same containment property as for AIDS and NASA.

[3]i.e., $=O\left((\text{the average number of vertex labels})^2\right)$

Table 4: Database size and construction time for Yeast dataset.

Alg.	Size(MB)	# of total I/Os	Time(msec.)
VF2	0.41	89	46
QuickSI	1.07	175	378
GraphQL	2.03	296	284
SPath(k_0=1)	3.73	514	437
GADDI	0.91	152	25272

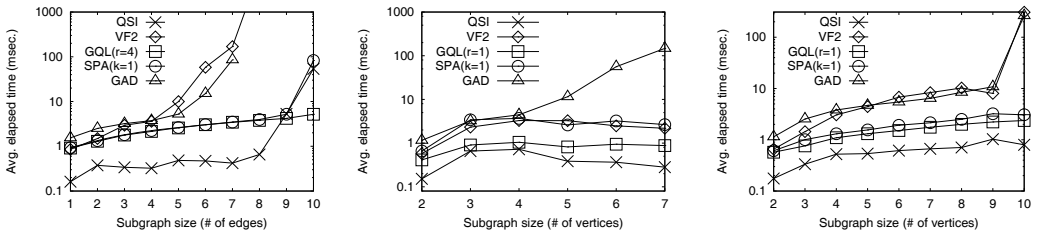
Figure 8(a) and Figure 9(a) show the subgraph isomorphism search performance using subgraph queries for Yeast dataset. SPath was run with $k_0 = 1$, and GraphQL was run with $r = 4$. Note that if we increase k_0 further, the query performance decreases due to the increased filtering cost. The GADDI was run with $d = 1$ and $k_0 = 2$. Only QuickSI, GraphQL, and SPath complete their query execution in a reasonable time. VF2 and GADDI show exponential behavior at query sizes nine and eight, respectively. This is due to the combined factors of increased search space and inefficient join ordering strategies of both algorithms. The average number of recursive calls increases with the size of queries. Note that such a trend is different from the results of the experiments on AIDS and NASA datasets. In terms of average elapsed time, QuickSI is the fastest algorithm until query size eight. However, on average, QuickSI, SPath, GADDI, and VF2 are 2.20, 3.78, 5.89, and 369.50 times slower than GraphQL, respectively. Although QuickSI is designed for handling small graphs, it often outperforms the more recent algorithms GraphQL and SPath which are designed for handling large graphs. This indicates that signature-based pruning might not be very effective for the Yeast dataset. To analyze this phenomenon, we calculate the size of $\sum_{u \in V(q)} |C(u)|$ for QuickSI, GraphQL, and SPath. The size of $\sum_{u \in V(q)} |C(u)|$ for QuickSI is only 2.01 and 2.78 times larger than those for SPath and GraphQL on average, respectively. However, increased filtering and loading times of SPath and GraphQL negate the advantage of signature-based pruning.

4.3.3 Clique Queries

Figure 8(b) and Figure 9(b) show the subgraph isomorphism search performance using clique queries for the Yeast dataset. We use the same parameter values as used in subgraph queries except for setting $r = 1$.

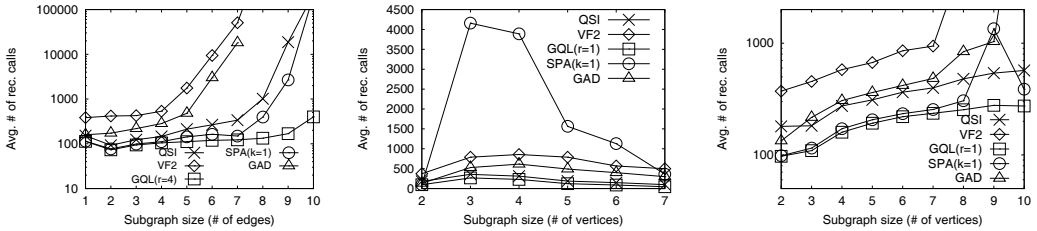
The average number of recursive calls for every algorithm increases as we increase the query size until three or four, then it decreases, if we increase the query size further. This trend is similar to that for NASA. Thus, all algorithms behave well despite their inherent exponential time complexity by choosing good join orders and by exploiting effective pruning rules. In terms of the average number of recursive calls, the ranking order is GraphQL, QuickSI, GADDI, VF2, and SPath.

In terms of the average elapsed time, the ranking order is QuickSI, GraphQL, VF2, SPath, and GADDI. This is due to the lower cost of a recursive call of QuickSI and VF2. Note that, although the numbers of recursive calls of GraphQL and SPath decrease as we increase the size of queries from five to seven, the elapsed times of GraphQL and SPath slightly increase. This is due to increasing overhead of filtering in GraphQL and SPath. As for GADDI, since the data graphs are dense, the comparison costs based on the NDS distance and the shortest path exponentially in-



(a) subgraph queries. (b) clique queries (Y-axis in log scale). (c) path queries.

Figure 8: Average elapsed time for Yeast PIN dataset.



(a) subgraph queries. (b) clique queries. (c) path queries.

[0] **Figure 9: Average number of recursive calls for Yeast PIN dataset.**

crease as the size of a query increases. Thus, GADDI shows the worst performance in terms of the average elapsed time.

4.3.4 Path Queries

Figure 8(c) and Figure 9(c) display the subgraph isomorphism search performance using path queries for the Yeast dataset. We use the same parameter values as used in subgraph queries except for setting $r = 1$.

The performance results of all the algorithms increase as the query size increases. In terms of the average elapsed time, the ranking order is QuickSI, GraphQL, SPath, GADDI, and VF2. This is because 1) QuickSI generates reasonably good join orders for path queries, and 2) the cost of a recursive call for QuickSI is the lowest. In terms of the average elapsed time, QuickSI is 2.53, 3.22, 56.96, and 65.12 times faster than GraphQL, SPath, GADDI, and VF2 on average. The VF2 and GADDI algorithms show exponential behaviors at query size ten because of the serious problems in their join order selection.

4.4 Human Dataset

4.4.1 Database Construction

Table 5 shows the database sizes and database building times for all five algorithms. Note that the number of read I/Os for all algorithms is 148. SPath was run with $k_0 = 3$ for subgraph queries. Otherwise, it was run with $k_0 = 1$. The GADDI was run with $d = 1$ and $k_0 = 2$. The ratio of the size of a QuickSI database over a VF2 database is 1.11, which is smaller than the ratio for the Yeast dataset. This is due to the smaller number of distinct vertex labels. Since this dataset has a larger average degree than Yeast, the database sizes of GraphQL and SPath are 5.27 and 13.58 times larger than that of VF2. Due to a larger degree than Yeast, the cost of calculating NDS distances sharply increases, and thus, the ratio of the building time of GADDI over VF2 accordingly increases compared with that for Yeast.

4.4.2 Subgraph queries

Figures 10(a) and 11(a) show the results of subgraph isomorphism search test using subgraph queries for the Human dataset. SPath was run with $k_0 = 3$, and GraphQL was run with $r = 4$. The GADDI was run with $d = 1$ and $k_0 = 2$.

Table 5: Database size and construction time for Human dataset.

Alg.	Size(MB)	# of total I/Os	Time(msec.)
VF2	1.55	349	93
QuickSI	1.72	373	964
GraphQL	8.13	1193	774
SPath(k_0=1)	11.28	1596	1669
SPath(k_0=3)	17.09	2340	3323
GADDI	4.88	775	624265

In terms of the average elapsed time and the average number of recursive calls, the performance of all algorithms increases as the query size increases. Although QuickSI shows the best performance in terms of the average elapsed time until the query size four, all algorithms except GraphQL fail to complete the subgraph queries due to their exponential behaviors at query sizes five, seven, eight, and eight for VF2, GADDI, QuickSI, and SPath, respectively. This is due to 1) the increasingly large search space size with increasing query sizes and 2) the join order selection problem. Note that this dataset has a higher density and fewer unique vertex labels than those of Yeast.

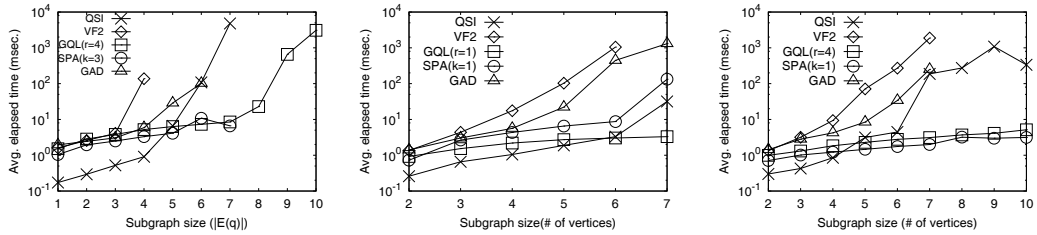
4.4.3 Clique and Path Queries

Figures 10(b) and 11(b) show the performance results for the clique queries. Figures 10(c) and 11(c) show the performance results for the path queries. We also use the same parameter values except for setting $r = 1$ and $k = 1$ for clique queries and setting $r = 4$ and $k = 1$ for path queries.

The performance trend of both types of queries for the Human dataset is similar to that of path queries for the Yeast dataset. The only notable difference is that, in path queries, the performance of QuickSI at query size ten is better than that at query size nine. This is because QuickSI fortunately selects better join orders at query size ten.

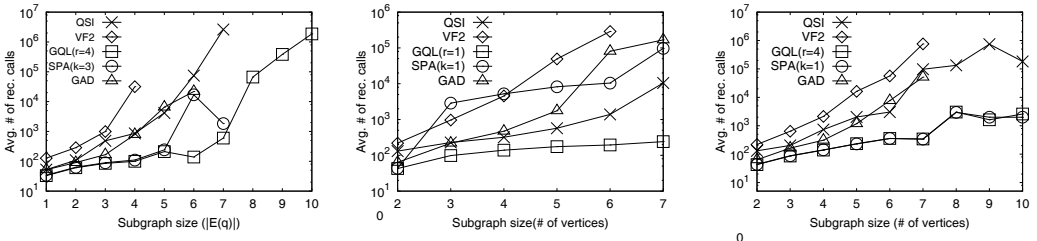
5. CONCLUSION

In this paper, we provide a fair comparison of subgraph isomorphism algorithms by using a common framework. We



(a) subgraph queries. (b) clique queries. (c) path queries.

Figure 10: Average elapsed time for the Human dataset.



0 (a) subgraph queries. (b) clique queries. (c) path queries.

Figure 11: Average number of recursive calls for the Human dataset.

performed extensive experiments with small and large real datasets and analyzed performance in depth.

Although there is no single winner for all experiments, to our surprise, QuickSI, the algorithm designed for handling small graphs, performs the best for many queries for both small and large data graphs (the AIDS and YEAST datasets) since the cost of its recursive call is the lowest. QuickSI, VF2, and GADDI failed to find embeddings in trees (the NASA set) in a reasonable time, showing exponential behavior due to serious problems in their join order selection. GADDI shows very bad performance for many queries tested due to expensive NDS distance calculation and lowest pruning power. GraphQL is the only algorithm that completed all queries tested, although it is slower than QuickSI for most queries. SPath almost performed worse than GraphQL due to its large SPath neighborhood signature overhead and the serious problem in its join order selection. We also note that all existing algorithms had problems in their join order selections, and signature-based pruning is only effective for some datasets. This calls for new subgraph algorithms which exploit both *good* join order selection and *selective* signature-based pruning.

We believe that our community will benefit greatly from our implementations and new findings. The source code of all algorithms that we implemented will be released along with publication of this paper.

6. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (No. 2012033242) and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/ National Research Foundation of Korea(NRF) (Grant 2012-0000474).

7. REFERENCES

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

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

[3] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1):449–459, 2010.

[4] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, pages 38–49, 2006.

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

[6] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.

[7] D. E. Knuth. Estimating the efficiency of backtrack programs. Technical report, Stanford, CA, USA, 1974.

[8] G. Miklau. Xml data repository. http://www.cs.washington.edu/research/xmldatasets.

[9] M. Mongiovi, R. D. Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. Sigma: a set-cover-based inexact graph matching algorithm. *J. Bioinformatics and Computational Biology*, 8(2):199–218, 2010.

[10] R. Shamir and D. Tsur. Faster subtree isomorphism. *Theory of Computing Systems, Israel Symposium on the*, 0:126, 1997.

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

[12] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[13] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[14] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, January 1976.

[15] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[16] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

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

[18] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

[19] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, pages 938–949, 2007.

[20] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.