PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing

Changjiu Jin[§] Sourav S Bhowmick[§] Byron Choi[†] Shuigeng Zhou[‡]

[§]School of Computer Engineering, Nanyang Technological University, Singapore
[†]Hong Kong Baptist University, Hong Kong
[‡]Fudan University, China

cjjin|assourav@ntu.edu.sg, choi@hkbu.edu.hk, sgzhou@fudan.edu.cn

August 27, 2013

Abstract

In a previous paper, we laid out the vision of a novel graph query processing paradigm where instead of processing a visual query graph *after* its construction, it *interleaves* visual query formulation and processing by exploiting the latency offered by the GUI to filter irrelevant matches and prefetch partial query results [10]. Our first attempt at implementing this vision, called GBLENDER [10], shows significant improvement in *system response time* (SRT) for subgraph containment queries. However, GBLENDER suffers from two key drawbacks, namely inability to handle visual subgraph similarity queries and inefficient support for visual query modification, limiting its usage in practical environment. In this paper, we propose a novel algorithm called PRAGUE (**PR**actical visu**A**l **G**raph **QU**ery bl**E**nder), that addresses these limitations by exploiting a novel data structure called *spindle-shaped graphs* (SPIG). A SPIG succinctly records various information related to the set of supergraphs of a newly added edge in the visual query fragment. Specifically, PRAGUE realizes a unified visual framework to support SPIG-based processing of *modification-efficient* subgraph containment *and* similarity queries. Extensive experiments on real-world and synthetic datasets demonstrate effectiveness of PRAGUE.

1 Introduction

Graph is an extensively studied subject in mathematics and many areas of computer science as it provides a natural way of modeling data in a wide variety of domains. For example, in chem-informatics graphs are used to represent atoms and bonds in chemical compounds. In bioinformatics, protein interaction networks are graphs where nodes represent molecules and edges represent interactions between them. Due to explosive growth of such graph-structured data in recent years, it is paramount to develop user-friendly, efficient, and scalable tools to process search queries on the graph databases.

A wide variety of graph queries in many applications (e.g., drug design, computer vision and pattern recognition) involve the core *substructure search* problem (also called *subgraph containment query*). In this problem, given a graph database $\mathcal{D}$ and a query graph q, the aim is to find all data graphs in $\mathcal{D}$ in which q is a subgraph. Note that q is a subgraph of a data graph $g \in \mathcal{D}$ if there exist a subgraph isomorphism from q to g [22]. A common problem for this type of query is that in many occasions there may not exists any $g \in \mathcal{D}$ that matches the query. For example, consider the substructure search query in Figure 1(a) and the data graphs in Figures 1(b) and (c). Observe that the query is not a subgraph of any of these data graphs. In this case, it is often useful to find out data graphs that "nearly" contain the query graph, which is called the *substructure similarity search* problem [20] (also called *subgraph similarity query*). For example, if we are allowed to miss at most two edges from the query in Figure 1(a), then both the data graphs match it as they contain subgraphs that nearly (or approximately) contain the query graph (shown by dotted box). Designing efficient strategy for evaluating such graph queries is a challenging problem due to its inherent computational hardness. Consequently in recent times, the database community has focused on proposing several innovative solutions to subgraph query processing [3, 6, 9, 10, 12, 17, 18, 20, 22–24, 26].

A number of graph query languages (e.g., SPARQL, PQL [11], GraphQL [5]) have been proposed that can be used to formulate subgraph queries. However, formulating a graph query using these languages often demands considerable cognitive effort from the end user and requires "programming" skill that is at least comparable to SQL [1, 8]. A user must be familiar with the syntax of the language to correctly formulate a query. Unfortunately, in many real life domains it is unrealistic to assume that users are proficient in expressing such queries [8, 10]. For instance, biologists cannot be expected to learn the complex syntax of PQL to be able to formulate meaningful queries over biological networks.

1.1 Motivation

The traditional approach to address the query formulation challenge is to build a user-friendly visual framework on top of a state-of-the-art graph query processing technique (e.g., [3]). Figure 2 depicts an example of such a visual interface. A

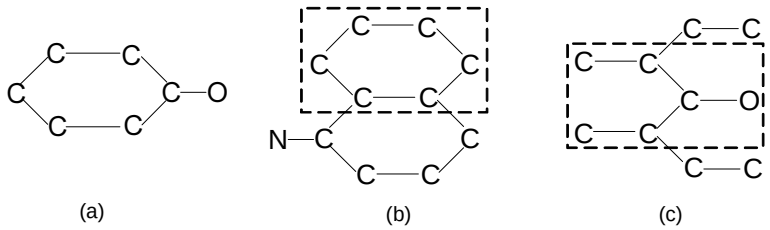


Figure 1: A query graph (a) and data graphs ((b) and (c)).

user begins formulating a query by choosing a database as the query target and creating a new query canvas using Panel 1. The left panel (Panel 2) displays the unique labels of nodes that appear in the dataset in lexicographic order. In the query formulation process, the user chooses labels from Panel 2 for creating the nodes in the query graph. Then, she drags a node that is part of the query from Panel 2 and drops it in Panel 3. Next, she adds another node in the same way and creates an edge between the added nodes by left and right clicking on them. Additional nodes and edges are added to the query graph by repeating these steps[1]. Finally, the user can execute the query by clicking on the `Run` icon in the *Query Toolbar*. Panel 4 displays the query results.

In traditional visual query processing paradigm, although the final query that a user intends to pose is revealed gradually in a step-by-step manner during query construction, it is not exploited by the query processor prior to clicking of the `Run` icon to execute the query. That is, query processing is initiated only *after* the user has finished drawing the query. This often results in slower *system response time* (SRT)[2] as the query processor remains idle during query formulation.

In [10], we laid out the vision of a novel graph query processing paradigm where we *blend* the two traditionally orthogonal steps, namely visual query formulation and query processing. Specifically, we proposed a visual subgraph containment querying system called GBLENDER (**G**raph **blender**) which was our first attempt at implementing this vision. Let us illustrate it with an example. Consider a graph-structured chemical compounds dataset. GBLENDER first mines and extracts the *frequent* and *infrequent graph fragments* from this dataset using an existing frequent graph mining algorithm [21]. These fragments are then used to construct the *action-aware frequent index* (A^2F) and *action-aware infrequent index* (A^2I) to support efficient matching of frequent and infrequent query fragments, respectively, while formulating a visual query.

Suppose now a user formulates a visual subgraph containment query over this dataset using the GUI in Figure 2. The sequence of steps taken by the user to formulate this query is shown in Figure 3 (*Sequence 1*). After every visual step taken by

[1] In this paper, we assume an "edge-at-a-time" visual query formulation interface. A more advanced and domain-dependent GUI may support drag and drop of *canned patterns* or subgraphs (e.g., benzene ring) for composing visual queries. Such visual query composition interface is beyond the scope of this work.

[2] Duration between the time a user presses the `Run` icon to the time when the user gets the query results [10].

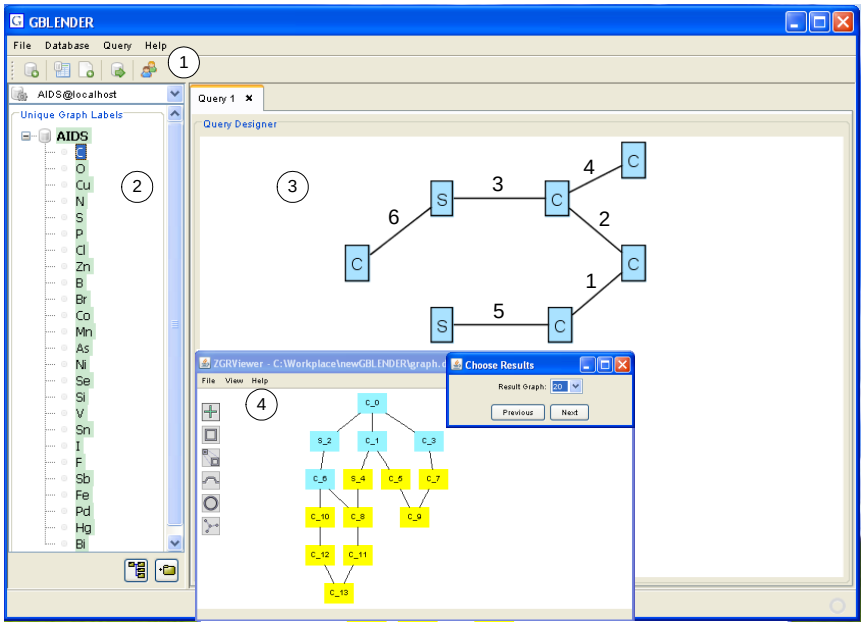


Figure 2: Visual interface for formulating graph queries.

the user, the current query fragment is evaluated by exploiting the latency offered by the GUI. For instance, after Step 1 the query fragment is a frequent fragment (see the *Status* column) and is efficiently evaluated using the A^{2}F-index and a set of identifiers of data graphs containing this fragment (denoted by R_q) is retrieved. Next, when the user draws Step 2, R_q is refined by filtering irrelevant matches using the index structure (the query fragment is still frequent). Observe that at Step 4, the query fragment evolves from frequent to an infrequent one. Consequently, the A^{2}I-index is probed and R_q is refined accordingly. This continues until the user click on the `Run` icon, when the final query results are computed by performing subgraph isomorphism test *if necessary*.

The key benefits of the aforementioned paradigm are two-fold. First, it ensures that the query processor does not remain idle during visual query formulation. Second, it significantly improves the SRT. In traditional graph processing paradigm, SRT is identical to the time taken to evaluate the entire query. In contrast, in this new paradigm SRT is the time taken to process a part of the query that is yet to be evaluated (if any).

Despite these appealing benefits of the new paradigm, GBLENDER has the following limitations. Firstly, it was designed to handle subgraph containment queries. Hence, if a query fragment does not have any match in the underlying database then it returns empty result set. For instance, in Figure 3 (*Sequence 1*) the query fragment after Step 5 does not have any match (the value of *Status* is set to "Similar" indicating that no exact match exists.). Hence, GBLENDER returns empty result set from this step onward. As mentioned earlier, this may not be desirable in many practical occasions. In fact, it should support substructure similarity

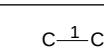
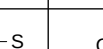
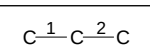
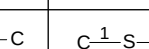
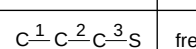
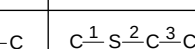
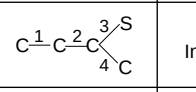
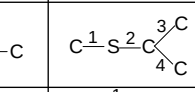
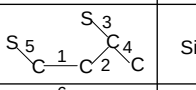
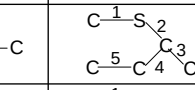
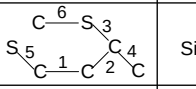
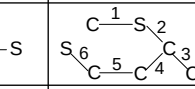
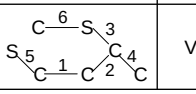
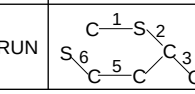
Steps	Sequence 1			Sequence 2		
	Action	Current Graph	Status	Action	Current Graph	Status
Step 1	C—C		frequent	C—S		frequent
Step 2	C—C		frequent	S—C		Infreq
Step 3	C—S		frequent	C—C		Infreq
Step 4	C—C		Infreq	C—C		Infreq
Step 5	C—S		Similar	C—C		Infreq
Step 6	C—S		Similar	C—S		Similar
Step 7	Click RUN		Verify	Click RUN		Verify

Figure 3: Query formulation steps.

search by retrieving graphs that are *similar* to the query fragment. Secondly, as GBLENDER utilizes the R_q computed in the *preceding* step to update the candidate data graphs, it is expensive to update it if a user modifies the formulated query fragment (e.g., delete an edge) at any time during query construction (see Section 2). In this paper, we propose a novel framework that provides a unified solution to the aforementioned limitations.

1.2 Overview and Contributions

We present a novel algorithm called PRAGUE (**PR**actical visu**A**l **G**raph Q**U**ery bl**E**nder) (see Section 4) that seamlessly supports evaluation of subgraph containment and similarity queries as well as efficient visual query modification[3]. During visual query formulation, it creates a novel data structure called ***spi**ndle-shaped **g**raph* (SPIG) for each new edge e_ℓ added by the user. A SPIG succinctly records information related to the set of supergraphs of e_ℓ in the query fragment q and their containment relationships (see Section 5 for details). The *source* (vertex with no incoming edge) and *target* (vertex with no outgoing edge) vertexes of a SPIG represent e_ℓ and q, respectively. The intermediate vertexes represent various supergraphs of e_ℓ in q.

The algorithm monitors the status of R_q at each step. If R_q remains non-empty at each step then SPIG-based subgraph containment search is invoked as q has ex-

[3]Note that there are two different research streams processing graph queries [14]. One stream handles a large number of small graphs. The other stream handles a small number of large graphs using approximate graph search. The former is the focus of our study.

act matches in the database (see Section 6). However, if R_q becomes empty then it again exploits the SPIG set to efficiently support the following two steps. (a) If the user chooses to modify the query fragment (e.g., the user may wish to retrieve only exact matches) then it automatically *suggests* the edge e_d that needs to be deleted to make R_q non-empty. Specifically, it exploits the SPIG set to efficiently compute e_d whose deletion would maximize the size of the candidate graph set of the modified query fragment. Note that the user may ignore this suggestion and is free to delete any edge (at any time during query formulation) that has been previously constructed by her (see Section 7). (b) Otherwise, it invokes substructure similarity search to retrieve approximate matches to q (see Section 6).

In Section 8, our experimental study demonstrates that PRAGUE has excellent performance as the system response time (SRT) and query modification cost grow gracefully with increasing number of data graphs. Importantly, our results show that the latency offered by the GUI at every step during visual query formulation is sufficient to efficiently support practical subgraph query processing in the new paradigm. We also show that PRAGUE has significantly smaller candidate size compared to several traditional substructure similarity search techniques [12, 17, 20]. Consequently, in spite of adopting a simple subgraph similarity verification technique [4], its SRT is often significantly smaller than these techniques. In summary, the main contributions of this paper are as follows.

- We present a novel data structure called *spindle-shaped graph* (SPIG) which facilitates efficient pruning and retrieval of partial results satisfying subgraph containment and similarity query fragments. It also provides an efficient framework to support modification to the visual query at any time during its construction.
- We present a novel algorithm that unifies subgraph containment and similarity search in the new paradigm by efficiently exploiting the SPIGs and latency offered by the visual querying environment.
- We present an efficient and novel SPIG-based framework to support modifications to the visual query fragment during query construction by exploiting the GUI latency.
- By applying PRAGUE to real-world and synthetic datasets, we show its effectiveness, significant improvement of candidate graph size and SRT over existing methods based on traditional paradigm, and ability to handle increasing number of graphs for exact and approximate subgraph matches as well as query modification.

2 Related Work

Recently, there have been a number of studies to speed up evaluation of subgraph containment [3, 6, 15, 22, 24–27] and subgraph similarity queries [6, 9, 12, 16–18,

20, 23] over large graph databases. All these efforts follow the conventional query processing paradigm where the formulation of a query graph is independent of its evaluation against the database. Typically, the *complete* query is first specified before it is processed. Consequently, the indexing schemes and query processing strategies are effective only when the complete query is known [10]. In contrast, in PRAGUE query processing is initiated when the *entire* query is not known and leverages on the GUI latency and users' interaction behaviors for efficient pruning and retrieval. Hence, our proposed method for substructure similarity search is orthogonal to these existing techniques.

More germane to this work is our previous effort in [10] called GBLENDER. The main idea behind GBLENDER is to compute efficiently the *identifier* of data graphs containing unique *discriminative infrequent fragments* (DIFs) (for infrequent queries) or *frequent fragments* (for frequent queries) with the addition of each new edge by utilizing the candidate matches of the preceding step. The candidate space for final verification is generated by intersecting the identifier sets of the data graphs (R_q) containing these fragments.

Our work differs from GBLENDER in the following ways. Firstly, we focus on a practical querying environment where we assume that a user is oblivious to the nature of the query fragment match (exact or approximate) at different formulation steps. Our proposed query evaluation technique automatically responds to the evolving nature of the query fragment type by invoking exact or substructure similarity search. In contrast, in GBLENDER the visual query framework assumes that the formulated query fragment must have *exact* matches to the data graphs. Otherwise, it returns empty results set. Secondly, we present an efficient framework to support modifications to a visual query any time during query formulation. Query modification was not discussed in [10].

Thirdly and more importantly, although GBLENDER and PRAGUE exploit the same action-aware indexing schemes they have *very distinct* query processing strategies. GBLENDER is based on the assumption that as the size of a query graph increases the size of candidate data graphs decreases. Consequently, it only records the *most recent* R_q. Although this assumption is sufficient to support efficient subgraph containment query processing, it is not conducive for subgraph similarity queries as the candidates set size may not decrease after each formulation step. Furthermore, it also makes update of candidate data graphs expensive when a user modifies the visual query graph during formulation. For instance, suppose at Step i a user deletes an edge that was formulated at Step k ($k < i$). Then, GBLENDER needs to recompute R_q for each step again starting from the earliest step which obviously involves unnecessary processing. In contrast, PRAGUE exploits a novel data structure called *spindle-shaped graph* (SPIG) which efficiently records the DIFs and frequent fragments extracted during *all* (not only the most recent) query formulation steps for future processing. It exploits these information effectively to support *both* exact and approximate matches to users' queries as well as query modifications. Specifically, for each query formulation step, a new SPIG is generated and recorded. Each vertex of a SPIG represents a partial query graph for-

Table 1: Key symbols.

Symbol	*Definition*
$\mathcal{D}$	A graph database
g, G	A (sub)graph
q, Q	A query graph (fragment)
$\mathcal{D}_g$	A set of FSGs of g
$fsgIds(g)$	Set of identifiers of the data graphs in $\mathcal{D}_g$
$delId(g)$	A subset of $fsgIds(g)$ used in indexes
dif_i	A discriminative infrequent fragment (DIF)
inf_i	A non-discriminative infrequent fragment (NIF)
$\mathcal{I}_d$	A set of DIFs in $\mathcal{D}$
$a2fId(\cdot)$	Identifier of each node in A^{2}F-index
$a2iId(\cdot)$	Identifier of a DIF in A^{2}I-index
α	Minimum support threshold
σ	Subgraph distance threshold
β	Fragment size threshold
e_ℓ	A new edge added by user
$S_\ell = (V_\ell, E_\ell)$	A SPIG
$\mathcal{L}_E(g)$	*Edge List* associated with the vertex $v \in V_\ell$ containing a list of labels of edges in g
$\mathcal{L}_{frag}(g)$	The *Fragment List* of a vertex $v \in V_\ell$ representing g
$freqId(g)$	frequent id attribute of $\mathcal{L}_{frag}(g)$
$difId(g)$	DIF id attribute of $\mathcal{L}_{frag}(g)$
$\Phi(g)$	frequent subgraph id set attribute of $\mathcal{L}_{frag}(g)$
$\Upsilon(g)$	DIF subgraph id set attribute of $\mathcal{L}_{frag}(g)$
$\mathcal{S}$	A set of SPIGs
R_q	Identifiers of data graphs containing q
R_{free}	Identifiers of verification-free candidate graphs
R_{ver}	Candidate graphs that need verification

mulated in previous steps. A novel and efficient SPIG management strategy is also proposed to build, update, and remove SPIGs during query formulation in order to support practical subgraph query processing.

3 Background

For the sake of completeness, in this section we briefly describe the *action-aware* indexing schemes of GBLENDER [10], which we shall be exploiting in the sequel. Note that it is important for PRAGUE to seamlessly support subgraph similarity queries on top of existing indexing structure as it is inelegant to maintain two distinct index structures to support subgraph containment and similarity queries. We begin by introducing subgraph isomorphism that is fundamental to the understand-

ing of graph query processing. First, we briefly describe *frequent* and *infrequent* graph fragments [10]. Then, we discuss the indexing schemes to index these fragments. The key notations used in this paper are summarized in Table 1.

3.1 Subgraph Isomorphism

A graph G is denoted as (V, E), where V is the set of nodes and $E \subseteq V \times V$ is the set of (directed or undirected) edges in the graph. Nodes and edges can have labels as attributes specified by mappings $\phi : V \rightarrow \sum_{V_\ell}$ and $\psi : E \rightarrow \sum_{E_\ell}$ respectively, where $\sum_{V_\ell}$ is the set of node labels and $\sum_{E_\ell}$ is the set of edge labels. In this paper, we assume that G (data or query graph) has at least one edge, and all nodes in G are connected (no dangling edges or nodes). The *size* of G is defined as $|G| = |E|$. For ease of presentation, we present our method using undirected graphs with labeled nodes. It is straightforward to extend our method to process edge-labeled and/or directed graphs.

A graph $G_1 = (V_1, E_1)$ is a *subgraph* of another graph $G_2 = (V_2, E_2)$ (or G_2 is a *supergraph* of G_1) if there exists a subgraph isomorphism from G_1 to G_2, denoted by $G_1 \subseteq G_2$ (or $G_2 \supseteq G_1$). We may also simply say that G_2 contains G_1. The graph G_1 is called a *proper subgraph* of G_2, denoted as $G_1 \subset G_2$, if $G_1 \subseteq G_2$ and $G_1 \not\supseteq G_2$.

Definition 1 *(**Subgraph Isomorphism**) A **subgraph isomorphism** is an injective function $f : V_1 \rightarrow V_2$, such that (1) $\forall u \in V_1$, $\phi_1(u) = \phi_2(f(u))$, and (2) $\forall (u, v) \in E_1$, $(f(u), f(v)) \in E_2$ and $\psi_1(u, v) = \psi_2(f(u), f(v))$.*

3.2 Frequent and Infrequent Fragments

Informally, we use the term *fragment* to refer to a small subgraph existing in graph databases or query graphs. Let $\mathcal{D}$ be a graph database containing a set of data graphs. We assign an unique identifier to each data graph in $\mathcal{D}$. Let g be a subgraph of $G_i \in \mathcal{D}$ $(0 < i \leq |\mathcal{D}|)$ and has at least one edge. Then, g is a *fragment* in $\mathcal{D}$. Given a fragment $g \subseteq G$ and $G \in \mathcal{D}$, G is referred to as the *fragment support graph* (FSG) of g. We denote the set of FSGs of g as $\mathcal{D}_g$. We refer to $|\mathcal{D}_g|$ as (*absolute*) *support*, denoted by $sup(g)$. We denote the set of identifiers of the data graphs in $\mathcal{D}_g$ as $fsgIds(g)$. Note that we shall refer to a fragment in a query graph as *query fragment* in order to distinguish it from a fragment in a data graph.

A fragment g is *frequent* if its support is no less than $\alpha|\mathcal{D}|$ where α is the minimum support threshold [10]. That is, if $g \in \mathcal{D}$ and $sup(g) \geq \alpha|\mathcal{D}|$ and $0 < \alpha < 1$ then g is a *frequent* fragment in $\mathcal{D}$. We denote the set of frequent fragments in $\mathcal{D}$ as $\mathcal{F}$. For example, let $|\mathcal{D}| = 10000$ and $\alpha = 0.1$. Then, all the fragments with support larger than or equal to 1000 are frequent fragments. The fragments $f_0 - f_6$ in Figure 4 are frequent fragments (support values shown in parenthesis). Note that a frequent fragment's subgraph must be a frequent fragment [22].

Given a fragment $g \in \mathcal{D}$, if $sup(g) < \alpha|\mathcal{D}|$ then g is an *infrequent* fragment [10]. For example, in Figure 4 dif_0-dif_2 and inf_0-inf_7 are infrequent fragments. We denote the set of infrequent fragments in $\mathcal{D}$ as $\mathcal{I}$. Specifically, only *discriminative infrequent fragments* (DIFs) are indexed in GBLENDER as it is computationally expensive to index all infrequent fragments in the database. Informally, a DIF is a smallest infrequent subgraph of an infrequent fragment. Given $g \in \mathcal{I}$, let $sub(g)$ be the set of all subgraphs of g. If $sub(g) \subset \mathcal{F}$ or $|g| = 1$, then g is a *discriminative infrequent fragment* (DIF) in $\mathcal{D}$. For example in Figure 4, dif_1 is a DIF as all its subgraphs are frequent fragments (f_0, f_1, f_2, f_3, and f_5). Similarly, dif_0 and dif_2 are DIFs. However, inf_0 is not a DIF as one of its subgraph (C−S−C) is infrequent. We denote a set of DIFs in $\mathcal{D}$ as $\mathcal{I}_d$.

A DIF satisfies the following properties (see [10]).

- Let $g' \in \mathcal{I}_d$ and $g \in \mathcal{D}$. If $g' \subset g$ then $g \in \mathcal{I}$.
- Given $g \in \mathcal{I}$, $\exists g' \in \mathcal{I}_d$ such that $g' \subseteq g$.
- Given $g \in \mathcal{I}$, if $\forall g_i \subset g$ and $g_i \in \mathcal{F}$, $g \in \mathcal{I}_d$.

For distinction, we refer to an infrequent fragment that is not a DIF as *non-discriminative infrequent fragment* (NIF). For example, inf_0-inf_7 are NIFs. Note that if one of the subgraphs of g is a DIF, then g is an infrequent fragment. Therefore, a DIF can be used in turn to identify an infrequent fragment. In practice, the number of DIFs is significantly smaller than the total number of infrequent fragments [10].

3.3 Action-Aware Indexes

The *action-aware frequent index* (A^2F) is a graph-structured index having a *memory-resident* and a *disk-resident* components called *memory-based frequent index* (MF-index) and *disk-based frequent index* (DF-index), respectively. Small-sized frequent fragments (frequently utilized) are stored in MF-index whereas larger frequent fragments (less frequently utilized) reside in DF-index. Informally, DF-index is an array of *fragment clusters*. A *fragment cluster* is a directed graph $\mathcal{C} = (V_C, E_C)$ where each *vertex*[4] $v \in V_C$ is a frequent fragment f where the size of f (denoted as $|f|$) is greater than the *fragment size threshold* β (i.e., $|f| > \beta$). There is an edge $(v', v) \in E_C$ *iff* f' is a proper subgraph of f (denoted as $f' \subset f$) and $|f| = |f'| + 1$. The root vertex (vertex with no incoming edge) of $\mathcal{C}$ is denoted by $root(\mathcal{C})$. Each fragment f of v is represented by its CAM code [7], denoted as $cam(g)$. That is, g is represented by an adjacency matrix M. Every diagonal entry of M is filled with the label of the corresponding node and every off diagonal entry is filled with 1 or 0 if there is no edge. The CAM code is formed by concatenating lower triangular entries of M, including the entries on the diagonal. The order is from top to bottom and from the leftmost entry to the rightmost entry. We choose

[4]For clarity, we distinguish between a node in a query graph fragment and a node in action-aware indexes and SPIGs by using the terms "node" and "vertex", respectively.

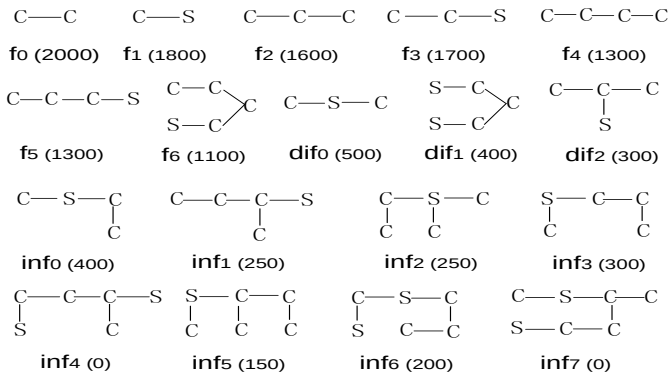


Figure 4: Frequent and infrequent fragments.

the maximal code among all possible codes of a graph by lexicographic order as this graph's canonical code. Details about CAM code can be found in [7]. Each vertex with fragment f in $\mathcal{C}$ points to a set of FSG identifiers of f ($fsgIds(f)$). Note that it is not space efficient to attach the complete list of $fsgIds(f)$ on each vertex as the size can be large. Fortunately, given the frequent fragments f and f', if $f' \subset f$ then $fsgIds(f) \cap fsgIds(f') = fsgIds(f)$ [3]. That is, node v' (representing f') and its child node v (representing f) share a large number of FSGs. GBLENDER exploits this to store only $delId(f) \subset fsgIds(f)$.

MF-index indexes all frequent fragments having size less than or equal to β. Similar to a fragment cluster, it is a directed graph $G_M = (V_M, E_M)$ where the vertexes and edges have same semantics as $\mathcal{C}$. In addition, by abusing notations for trees, vertexes representing frequent fragments of size β are *leaf* vertexes in G_M. Each leaf vertex $v \in V_M$ (representing f) is additionally associated with a *fragment cluster list* $\mathcal{L}$ where each entry $\mathcal{L}_i$ points to a fragment cluster $\mathcal{C}_j$ in the DF-index such that $f \subset root(\mathcal{C}_j)$. An example of MF-index is depicted in Figure 5(a) ($\beta = 4$) based on the frequent fragments in Figure 4. Note the distinction between $delId(f)$ and $fsgIds(f)$. For instance, $|delId(f_0)| = |fsgIds(f_0)| - |fsgIds(f_2)| - |fsgIds(f_3)|$. Also, each vertex v in A^{2}F-index is assigned an identifier, denoted by $a2fId(v)$ (e.g., $a2fId(v_0) = 0$ in Figure 5(a))

The *action-aware infrequent index* (A^{2}I) indexes DIFs to prune the candidate space for infrequent queries. It consists of an array of DIFs arranged in ascending order of their sizes. Each entry stores the CAM code of a DIF g and a list of FSG identifiers of g. Figure 5(b) depicts a A^{2}I-index based on the DIFs in Figure 4. The identifier of each DIF g in the index is denoted by $a2iId(g)$ (e.g., $a2iId(dif_1) = 1$ in Figure 5(b)).

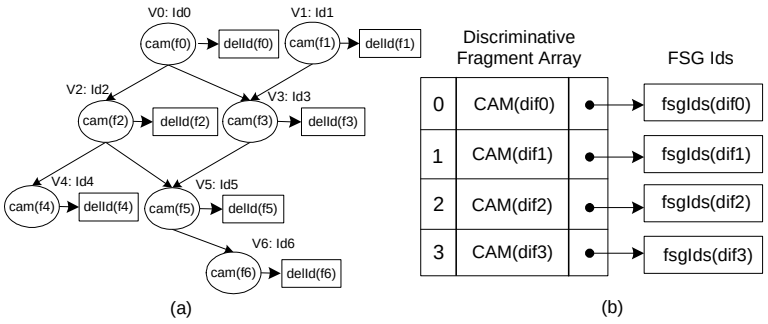


Figure 5: Examples of MF-index and A^{2}I-index.

4 Overview of PRAGUE

In this section we first present the substructure similarity search problem and then give an overview of the PRAGUE algorithm. In subsequent sections, we discuss the algorithm in detail.

4.1 Substructure Similarity Search Problem

While many techniques have been proposed in the literature for evaluating subgraph containment queries [3, 6, 9, 10, 22, 24–27], very few algorithms exist for processing subgraph similarity queries [6, 12, 17, 18, 20, 23]. Most of the existing subgraph similarity query processing techniques measure similarity between two graphs using distance measures that are either based on *graph edit distance* [6, 18, 23] or *maximum common subgraph* [17, 20]. In the former approach, the similarity of two graphs is defined by the least edit operations (insertion, deletion, and relabeling) used to transform one graph into another. Each of these operations relaxes the query graph by removing or relabeling one edge. The latter approach detects the *Maximum Common Subgraph* (MCS) [2] between the query graph and the data graphs, and measures the similarity based on the difference of the query graph and the MCS. Grafil [20] uses MCS to compute similarity between graphs. Since the maximum common subgraph is not necessarily connected, it may include many low-quality results in substructure similarity search [17]. This is because it is possible that different parts of a query are mapped to very different locations in a data graph which are far away from each other. To alleviate this problem, Shang et al. [17] adopted *maximum connected common subgraphs* (MCCS) for the substructure similarity search problem. Given two graphs Q and G, the *maximum common connected subgraph* of Q and G is the largest connected subgraph of Q that is subgraph-isomorphic to G, denoted as $mccs(G, Q)$.

In spite of the applicability of edit distance for any type of graphs and its superior quality of results over MCS for several cases [23], in this paper we use MCCS for similarity search for the following reasons. Firstly, as highlighted in [2] any edit distance measure critically depends on the costs of the underlying edit opera-

Algorithm 1: PRAGUE

Input: GUI $Action$, query q, candidate set R_q, subgraph distance threshold σ, graph database $\mathcal{D}$.
Output: Query results $Results$

if $Action$ *is* New **then**
  $q \leftarrow q + e_\ell$;
  $S_\ell \leftarrow$ **SpigConstruct**(q, Q, e_ℓ, $\mathcal{S}$) /*Algorithm 2*/;
  if $simFlag = false$ **then**
    $R_q \leftarrow$ **ExactSubCandidates**($S_\ell.v_{target}$) /*Algorithm 3*/;
    if $R_q = \emptyset$ **then**
      $Action \leftarrow$ **OptionDialogueDisplay**();
  else
    $(R_{free}, R_{ver}) \leftarrow$ **SimilarSubCandidates**(q, σ, $\mathcal{S}$) /*Algorithm 4*/;
else if $Action$ *is* $Modify$ **then**
  $q \leftarrow$**QueryModification**(q, R_q, $\mathcal{S}$, σ) /*Algorithm 7*/;
else if $Action$ *is* $SimQuery$ **then**
  Set $simFlag = true$;
  $(R_{free}, R_{ver}) \leftarrow$ **SimilarSubCandidates**(q, σ, $\mathcal{S}$);
else if $Action$ *is* Run **then**
  if $simFlag = false$ **then**
    $Results \leftarrow$ **ExactVerification**(R_q);
    if $Results = \emptyset$ **then**
      $(R_{free}, R_{ver}) \leftarrow$ **SimilarSubCandidates**(q, σ, $\mathcal{S}$);
      $Results \leftarrow$ **SimilarResultsGen**(q, R_{free}, R_{ver}, σ) /*Algorithm 5*/;
  else
    $Results \leftarrow$**SimilarResultsGen**(q, R_{free}, R_{ver}, σ);

tions. How these edit costs are obtained is still a challenging problem. The costs that are assigned to the edit operations have an important influence on the matching results. *Two graphs that are similar under one particular cost function may be no longer similar under another cost function.* Secondly and more importantly, in a visual querying system the choice of similarity measure needs to take into account the cognitive overhead associated with the end-users to interpret the similarity matches. Visually displaying edit operations on query results to highlight similarity between a pair of graphs *add significant cognitive overhead to end-users who may not have any knowledge about edit distance*. Comparatively, missing edges (used for MCCS) are more intuitive in a visual system and easier to interpret. It can easily be depicted in the results by highlighting the MCCS in the matched data graphs.

Definition 2 ***(Subgraph Similarity Degree)*** *Given two graphs Q and G, the subgraph similarity degree from G to Q is defined as: $\delta = \frac{|mccs(G,Q)|}{|Q|}$.*

The *subgraph distance* measures the maximum number of edges that are al-

lowed to be missed (deleted) in Q in order to match G.

Definition 3 *(**Subgraph Distance**) Given two graphs G and Q and their subgraph similarity degree δ, the subgraph distance, denoted as $dist(Q, G)$, is defined as follows: $dist(Q, G) = \lfloor (1 - \delta)|Q| \rfloor$.*

Observe that the subgraph similarity degree and subgraph distance are used to measure the similarity between two graphs. Two graphs G_1 and G_2 with a larger δ or smaller $dist$ are more similar to each other. If $\delta = 1$ or $dist(G_1, G_2) = 0$, then G_1 and G_2 are subgraph isomorphism to each other.

Definition 4 *(**Substructure Similarity Search**) Given a query graph Q, a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_n\}$, and subgraph distance threshold σ, the goal of substructure similarity search problem is to retrieve all the graphs $g_i \in \mathcal{D}$ with $dist(Q, g_i) \leq \sigma$.*

Example 1 Reconsider the query and data graphs in Figure 1. If we set σ as 1 (one edge miss), then Figure 1(b) is an approximate match with $\delta = 6/7$. If we relax σ to 2, then Figure 1(c) is also an approximate match with $\delta = 5/7$. ∎

4.2 Algorithm Overview

The PRAGUE algorithm is outlined in Algorithm 1. In the sequel, we assume that subgraph queries in PRAGUE are formulated using the GUI in Figure 2. Let q be the visual query being formulated by the user. Let $simFlag$ be a boolean variable to indicate if q is subgraph similarity or containment query (*true* or *false*, respectively). We monitor four visual actions on the GUI, namely New for new edge addition, $Modify$ for deletion of an existing edge, $SimQuery$ for invoking substructure similarity search, and Run for executing q. When the user adds a new edge e_ℓ to q, the algorithm first constructs the *spindle-shaped graph* (SPIG) S_ℓ (Line 3). If $simFlag$ is *false*, it retrieves the FSG identifiers of q (R_q) by invoking the *ExactSubCandidates* procedure (Line 5). Note that this step follows a different strategy from GBLENDER [10] as it exploits the SPIG to determine candidates. As mentioned earlier, SPIGs are not generated by GBLENDER.

If R_q is empty, then there is no exact match for q after the addition of e_ℓ. Consequently, PRAGUE gives the user options to either modify q ($Action$ is $Modify$) or enable retrieval of approximate matches ($Action$ is $SimQuery$) by popping out an option dialogue box (Line 7). If the user chooses to modify q, then it provides suggestion on which edge she should delete in order to ensure R_q is not empty. The user may select the suggested modification or perform a different modification to q. These steps are encapsulated in the procedure *QueryModification* (Line 11). On the other hand, if the user intends to continue formulating the query without modification ($Action$ is $SimQuery$), then the algorithm identifies q as a subgraph similarity query. The *SimilarSubCandidates* procedure retrieves the candidate data

graphs that match approximately with q by exploiting the SPIG set $\mathcal{S}$ (Line 14). These steps are repeated for each new edge until the user clicks the `Run` icon (Line 15). If $simFlag$ is *false*, then the exact results $Results$ are returned from the candidate graphs (Line 17). If $Results$ is empty after candidate verification (subgraph isomorphism test) then the substructure similarity search is invoked to retrieve approximate matches (Lines 19-20). Otherwise, if it is already a substructure similarity search ($simFlag$ is *true*), then a list of data graphs that match the query approximately is returned to the user. This step is encapsulated in the procedure *SimilarResultsGen* (Lines 22). We now elaborate on these procedures in detail.

Example 2 Consider the visual query in Figure 2 and the graph fragments in Figure 4. Figure 3 depicts two distinct sequences of visual actions (also referred to as steps) for formulating this query. In these steps, we assume $\sigma = 2$. In *Sequence 1*, the query remains as frequent in the first three steps. After Step 4, it evolves to an infrequent query. Note that in these steps, the candidates satisfying the query fragments are computed by the *ExactSubCandidates* procedure (Line 5). When the edge 5 is added to the query fragment in Step 5, the exact candidate set (R_q) becomes empty. Consequently, the *OptionDialogueDisplay* procedure pops out an option dialogue for the user to choose if she intends to modify the query fragment to retrieve exact matches or invoke substructure similarity search (Line 7). If the user chooses to modify the query, then the *QueryModification* procedure is invoked (Line 13). Otherwise, PRAGUE considers it to evolve to a subgraph similarity query (Steps 5 and 6) and *SimilarSubCandidates* procedure is invoked to retrieve candidate set for approximate match (Line 16). After the user clicks the `Run` icon (Step 7), the *SimilarResultsGen* is invoked to return the list of data graphs that match the query approximately (Line 25). ∎

5 Spindle-Shaped Graph (SPIG)

We now present in detail the concept of *spindle-shaped graph*.

5.1 Definition of SPIG

For each *new* edge e_ℓ created by the user, we create a ***spi**ndle-shaped **g**raph* (SPIG). We allocate each edge a unique identifier according to their formulation sequence. That is, the *ℓ-th* edge constructed by a user is denoted as e_ℓ where ℓ is the *label* of the edge. The edge with the *largest* ℓ is referred to as *new edge* (most recently added). For example, in Figure 3 (Sequence 1) after Step 4, four edges have been constructed and they are uniquely identified as e_1 to e_4. The new edge is e_4 (C−C) as $\ell = 4$ is largest in this step.

A SPIG is a directed graph $S_\ell = (V_\ell, E_\ell)$ where each vertex $v \in V_\ell$ represents a subgraph g of the query fragment containing the new edge e_ℓ. In the sequel, we

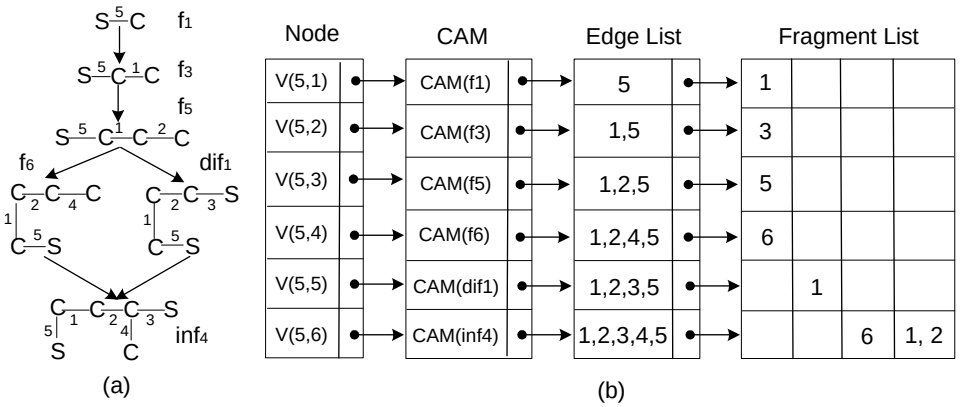


Figure 6: The vertices of the spindle-shaped graph in step 5.

refer to a vertex v and its associated query fragment g interchangeably. There is a directed edge from vertex v' to vertex v if $g' \subset g$ and $|g| = |g'| + 1$. Each v is associated with the CAM code of the corresponding g, a list of labels of edges of g, and a list of identifier set called *Fragment List* to capture information related to frequent or infrequent nature of g or its subgraphs. We now elaborate on the structure of a *Fragment List*.

A *Fragment List* contains four attributes, namely *frequent id*, DIF *id*, *frequent subgraph id set*, and DIF *subgraph id set*.

- If g is in A^{2}F-index or A^{2}I-index (see Section 3), then the identifier of the vertex or entry representing g in the corresponding index is stored in *frequent id* or DIF *id* attribute, respectively. Recall from Section 3, the identifier of a vertex or an entry in A^{2}F-index or A^{2}I-index is denoted by $a2fId(g)$ or $a2iId(g)$, respectively.
- If g is neither in A^{2}F-index nor in A^{2}I-index, then the *frequent subgraph id set* stores the frequent ids of all *largest* proper subgraphs of g that are in A^{2}F-index. Note that size of these subgraphs is $|g| - 1$. The DIF *subgraph id set* of g contains the DIF ids of all subgraphs of g that are indexed by A^{2}I-index.

The *source* vertex (vertex with no incoming edge) in the first level of S_ℓ, denoted by $S_\ell.v_{source}$, represents e_ℓ and the *target* vertex (vertex with no outgoing edge) in the last level, denoted by $S_\ell.v_{target}$, represents the entire query fragment at a specific step. Since there is only one vertex at the first and the last level and a set of vertices in the "middle" levels, the shape of S_ℓ is like a spindle.

Definition 5 *(**Spindle-shaped Graph** (**SPIG**)) Let e_ℓ be the new edge added to a visual graph query q during Step ℓ. Then, the **spindle-shaped graph** (*SPIG*) of e_ℓ is a directed graph $S_\ell = (V_\ell, E_\ell)$ that satisfies the following conditions.*

Algorithm 2: ***SpigConstruct***

Input: Query q, Vertex queue $\mathbb{Q}$, new edge e_ℓ, set of SPIGs $\mathcal{S}$
Output: Spindle-shaped graph S_ℓ

1 $v_{\ell,1} \leftarrow f(e_\ell)$;
2 **Enqueue**$(v_{\ell,1}, \mathbb{Q})$;
3 **Insert**$(v_{\ell,1}, S_\ell)$;
4 **while** $\mathbb{Q} \neq \emptyset$ **do**
5 $v_{\ell,i} \leftarrow$ **Dequeue**$(\mathbb{Q})$;
6 **foreach** $v_{\ell,j} \in S_\ell$ *is the parent of* $v_{\ell,i}$ **do**
7 Add $v_{\ell,j}$'s $FragmentList$ to $v_{\ell,i}$;
8 **if** $g_i \notin$ A^{2}F-*index or* A^{2}I-*index* **then**
9 $g'_i \leftarrow g_i - e_\ell$;
10 **if** g'_i *is connected* **then**
11 $v'_{\ell,i} \leftarrow$ Search $cam(g'_i)$ in the $|g'_i|$-th level of $S_{\ell'}$;
12 Attach $v'_{\ell,i}$'s $FragmentList$ to $v_{\ell,i}$;
13 **else**
14 Let g'_{i1} and g'_{i2} be the two connected components of g'_i;
15 Perform Lines 11-12 on both g'_{i1} and g'_{i2};
16 **else**
17 Attach $v_{\ell,i}$ with $difId(g_i)$ or $freqId(g_i)$;
18 **if** $|g_i|$=$|q|$ **then**
19 Add S_ℓ in $\mathcal{S}$;
20 **return** S_ℓ;
21 **else**
22 **foreach** $g_i \subset g_j \subset q$ *and* $|g_j|$=$|g_i|$+*1* **do**
23 **if** $v_{\ell,j} \notin \mathbb{Q}$ **then**
24 $v_{\ell,j} \leftarrow f(g_j)$;
25 **Enqueue**$(v_{\ell,j}, \mathbb{Q})$;
26 **Insert**$(v_{\ell,j}, S_\ell)$;
27 Connect $edge(v_{\ell,i}, v_{\ell,j})$;

- *For each $v \in V_\ell$, $\exists$ an injective function f: $v \rightarrow f(g)$ s.t. e_ℓ is contained in g and $g \subseteq q$.*
- *By abusing the notations of trees, each $(v', v) \in E_\ell$ represents the parent-child relationship between two vertices v' and v where v is the child of v' iff $g' \subset g$ and $|g| = |g'| + 1$.*
- *Each $v \in V_\ell$ is a 3-tuple $v = (cam(g), \mathcal{L}_E(g), \mathcal{L}_{frag}(g))$ where $cam(g)$ is the* CAM *code of g, $\mathcal{L}_E(g)$ is the Edge List containing a list of labels of edges in g, and $\mathcal{L}_{frag}(g) = (freqId(g), difId(g), \Phi(g), \Upsilon(g))$ is the Fragment List.*
  $freqId(g)$, $difId(g)$, $\Phi(g)$, and $\Upsilon(g)$ refer to frequent id, DIF *id, frequent subgraph id set, and* DIF *subgraph id set, respectively such that:*

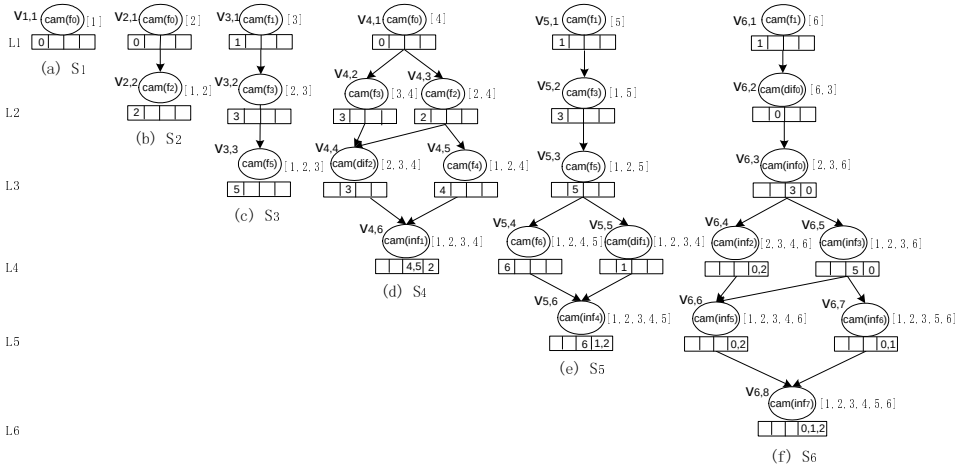


Figure 7: The SPIG set for Sequence 1 (Edge Lists are in square brackets and Fragment Lists are shown in rectangular boxes.)

1. *if* $g \in$ A^{2}F*-index, then* $freqId(g) = a2fId(g)$ *and* $difId(g) = \Phi(g) = \Upsilon(g) = \varnothing$.
2. *if* $g \in$ A^{2}I*-index, then* $difId(g) = a2iId(g)$ *and* $freqId(g) = \Phi(g) = \Upsilon(g) = \varnothing$.
3. *if* $g \notin$ A^{2}F*-index and* $g \notin$ A^{2}I*-index, then* $\forall g' \subset g$ *where* $|g'| = |g| - 1$, *if* $g' \in$ A^{2}F*-index, then* $\Phi(g)$ *contains* $a2fId(g')$, *and* $freqId(g) = difId(g) = \varnothing$. *Also,* $\forall g' \subset g$ *where* $g' \in$ A^{2}I*-index,* $\Upsilon(g)$ *contains* $a2iId(g')$.

- *Each* v *is uniquely identified by the pair* (ℓ, k) *where* k *is the position of* v *based on depth-first traversal order starting from* $S_\ell.v_{source}$.

Example 3 Consider Step 5 (*Sequence 1*) in Figure 3. Figure 6(a) depicts the SPIG S_5 after the addition of the new edge labeled 5 (e_5). Each vertex represents a subgraph of the query fragment containing e_5 and is identified by a pair of identifiers containing label of e_5 and its position. For instance, $v_{5,3}$ refers to the third vertex in S_5. Information associated with each vertex in S_5 is shown in Figure 6(b). Particularly, the entries from left to right in the Fragment List are $freqId$, $difId$, Φ, and Υ, respectively (we follow this sequence in all relevant figures). Note that $v_{5,1}, v_{5,2}, v_{5,3}$ and $v_{5,4}$ represent the frequent fragments f_1, f_3, f_5 and f_6 (Figure 4), respectively. Therefore, their $freqId$s are 1, 3, 5, and 6, respectively. Since $v_{5,5}$ represents dif_1, the $difId$ is 1 (Figure 5(b)). However, $v_{5,6}$ represents the NIF inf_4. Hence, it satisfies the Condition 3 in Definition 5 as inf_4 is neither indexed by A^{2}F-index nor by A^{2}I-index. Consequently, $freqId(v_{5,6}) = difId(v_{5,6}) = \varnothing$. Among all the largest proper subgraphs of inf_4 (size of these subgraphs is $|inf_4| - 1$), the subgraph f_6 (see Figure 4) is a

frequent fragment and hence stored in the A^{2}F-index (vertex id 6 in Figure 5(a)). Hence, $\Phi(v_{5,6}) = \{6\}$. Also, among all the subgraphs of inf_4, the subgraphs dif_1 and dif_2 (see Figure 4) are DIFs and are indexed by A^{2}I-index (having entry ids 1 and 2 in Figure 5(b)). Consequently, $\Upsilon(v_{5,6}) = \{1, 2\}$. ∎

5.2 Algorithm for SPIG Construction

The algorithm for building a spindle-shaped graph is shown in Algorithm 2. It takes as input the new edge e_ℓ added to the query fragment q, a set of SPIGs $\mathcal{S}$ from previous step, and a queue $\mathbb{Q}$ to temporarily store the vertexes of S_ℓ. The building process starts from the new edge (Lines 1-2). It first attaches the CAM code and edge label of e_ℓ to vertex $v_{\ell,1}$. Let $v_{\ell,i}$ be the vertex dequeued from $\mathbb{Q}$ (Line 5). For each $v_{\ell,j}$ in S_ℓ, if $v_{\ell,j}$ is the parent of $v_{\ell,i}$, then $v_{\ell,i}$ inherits the frequent and DIF ids of $v_{\ell,j}$. That is, it attaches $freqId(v_{\ell,j})$ to $\Phi(v_{\ell,i})$, $difId(v_{\ell,j})$ and $\Upsilon(v_{\ell,j})$ to $\Upsilon(v_{\ell,i})$ (Lines 6-7). If g_i is not a DIF or a frequent fragment (Line 8), then the algorithm first extracts the largest subgraph of g_i without e_ℓ (denoted by g'_i) (Line 9). g'_i can be either connected or unconnected. If g'_i is connected, let ℓ' be the new edge in g'_i where $\ell' < \ell$. Since $S_{\ell'}$ has already been constructed and stored in $\mathcal{S}$, the algorithm retrieves $v'_{\ell,i}$ from the$|g'_i|$-*th* level of $S_{\ell'}$ (Line 11). Then it attaches the relevant ids in $FragmentList$ of $v'_{\ell,i}$ to $v_{\ell,i}$ (Line 12). Note that as all the largest subgraphs of $v_{\ell,i}$ can be found in $\mathcal{S}$, the identifiers of frequent and infrequent fragments can be efficiently inherited *without* decomposing it to its subgraphs and retrieving them by probing action-aware-indices. When g'_i is not connected, since g_i is originally connected, e_ℓ must be a bridge of g_i and g'_i contains two connected components, g'_{i1} and g'_{i2}. Similar to g'_i in the previous case, both g'_{i1} and g'_{i2} are subgraphs of g_i and in their respective SPIGs in $\mathcal{S}$. Thus, the same process in Lines 11-12 are performed on both g'_{i1} and g'_{i2} to update the $FragmentList$ of $v_{\ell,i}$.

If g_i is a DIF or a frequent fragment, then it attaches frequent fragment id or DIF id of g_i on $v_{\ell,i}$'s $freqId$ or $difId$, respectively (Line 17). If $|g_i| = |q|$, then the SPIG construction process is terminated and S_ℓ is added to $\mathcal{S}$ (Lines 18-20). Otherwise, vertex $v_{\ell,j}$ is constructed as the child of $v_{\ell,i}$ in S_ℓ. For each $g_j \supset g_i$ in q, if $v_{\ell,j}$ does not exist in $\mathbb{Q}$ then it attaches the CAM code and edge labels of g_j to $v_{\ell,j}$ and inserts the vertex in $\mathbb{Q}$. Lastly, it adds $v_{\ell,j}$ in S_ℓ and connects $v_{\ell,i}$ and $v_{\ell,j}$ with a directed edge (Lines 22-27).

Observe that the aforementioned procedure *does not incrementally* build S_ℓ from $S_{\ell'}$ ($\ell' < \ell$) as e_ℓ is different in each formulation step. Consequently, the fragments represented by the vertices of S_ℓ are often different from those in $S_{\ell'}$. For instance, Figure 7 depicts a set of SPIGs constructed for Steps 1 to 6 in *Sequence 1* in Figure 3. Observe that the fragments in two consecutive SPIGs (e.g., S_5 and S_6) can be quite different.

Example 4 Reconsider the SPIG S_5 in Example 3. As the edge labeled 5 is the new edge, the SPIG construction starts off by creating the vertex $v_{5,1}$ representing fre-

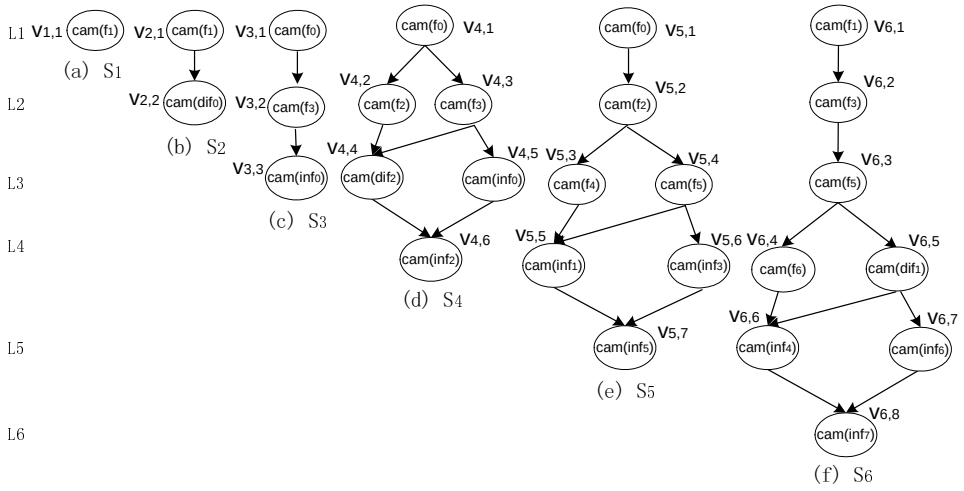


Figure 8: The spindle-shaped graph set for Sequence 2.

quent fragment f_1 (Line 1). Then the algorithm connects $v_{5,2}$ with $v_{5,1}$ as its child as f_3 is the child of f_1. Also, id 3 is attached to $freqId$ entry of the $fragmentList$ of $v_{5,2}$ (Line 17). Next, $v_{5,3}$ is constructed as the child of $v_{5,2}$. Since $v_{5,3}$ represents f_5, id 5 is attached to the $freqId$ entry of $v_{5,3}$. Then, the children of $v_{5,3}$ is constructed. As $v_{5,4}$ represents f_6 and $v_{5,5}$ represents dif_1, ids 6 and 1 are inserted in $freqId$ and $difId$ entries of $v_{5,4}$ and $v_{5,5}$, respectively. In the next step, the algorithm constructs $v_{5,6}$ to represent inf_4 (Lines 8-15). Observe that the largest subgraphs of inf_4 are f_6, dif_1, and inf_1. The new edges of f_6, dif_1 and inf_1 are edges labeled 5 and 4, respectively. Hence, it searches the vertices $v_{5,4}$, $v_{5,5}$ in S_5 and $v_{4,6}$ in $S_4 \in \mathcal{S}$ (Line 11). Note that all these vertices are in level 4 of their respective SPIGs. Consequently, $\Upsilon(v_{5,6})$ inherits the identifiers from $\Upsilon(v_{5,5})$ and $\Upsilon(v_{4,6})$, and $\Psi(v_{5,4})$ is assigned to $\Psi(v_{5,6})$. Finally, the construction of S_5 terminates as $v_{5,6}$ represents the current query fragment (Line 18) and S_5 is inserted into $\mathcal{S}$.

Figure 7 depicts a set of SPIGs constructed for Steps 1 to 6 in Sequence 1 in Figure 3. Observe that the fragments represented by vertices of two consecutive SPIGs (e.g., S_5 and S_6) can be quite different. ∎

5.3 Analysis of SPIG Construction

Size of SPIG set. The cost of SPIG construction depends on the number of edges in the query as it influences the number of levels and vertex set size of the SPIG. Let q be a visual query graph fragment with n *distinct* edges. That is, q has n edges with unique node label pairs (v_i, v_j). Then the maximum number of vertexes in the k-*th* level of S_ℓ is C_{n-1}^{k-1}. Consequently, the total number of vertices in S_ℓ is:

Algorithm 3: ***ExactSubCandidates***

Input: Target vertex v in S_ℓ, A^{2}F-index, A^{2}I-index
Output: Set of candidate identifiers R_q

```
if freqId(v) ≠ ∅ then
    i = freqId(v);
    R_q ← retrieve fsgIds(g_i) from A²F-index;
else if difId(v) ≠ ∅ then
    i = difId(v);
    R_q ← retrieve fsgIds(g_i) from A²I-index;
else
    foreach i ∈ Φ(v), j ∈ ϒ(v) do
        R_q ← R_q ∩ fsgIds(g_i) ∩ fsgIds(g_j);
```

$\sum_{k=1}^{n} \mathcal{C}_{n-1}^{k-1}$. However in practice, often some nodes in q share the same vertex labels. For example, in the query in Figure 2 there are only two distinct edges (`C-S` and `C-C`). Consequently, the number of unique vertexes in the k*-th* level of S_ℓ is much less than the worst-case scenario. For instance, only two vertexes are in the fourth level of S_6 (Figure 7(f)). We shall empirically study the cost of SPIG set construction in Section 8.

LEMMA 1 *The total number of vertexes in the k-th levels of* SPIG*s in $\mathcal{S}$ is: $N(k) \leq \mathcal{C}_n^k$.* □

Proof 1 *(Sketch)* $\forall g_i \subseteq q, v_i \in \mathcal{S}$. The number of g_i with k edges $\leq \mathcal{C}_n^k, n = |q|$, so $N(k) \leq \mathcal{C}_n^k$.

Effect of query formulation sequence. Different sequence of formulation steps for a query q (e.g., *Sequences 1* and *2* in Figure 3) will result in different SPIG sets. However, the total number of vertexes in the k*-th* level will remain identical in different SPIG sets. That is, given $\mathcal{S}_i$ and $\mathcal{S}_j$ generated by two distinct sequence of formulation steps for q, $N_i(k) = N_j(k)$.

For example, Figure 8 depicts the set of SPIGs that are created when we follow *Sequence 2* in Figure 3 to formulate the query. For brevity, in this figure we ignore displaying the *Edge List* and *Fragment List*. Now reconsider the SPIG set in Figure 7. Observe that S_5 in these two SPIG sets are different. Although the vertexes representing $inf_1, inf_2, inf_3, f_6, dif_1$ are in the fourth level of the SPIGs for *Sequences 1* and *2*, they are distributed in different SPIGs. For instance, in SPIGs generated by *Sequence 1*, inf_1 is in S_4, f_6 and dif_1 are in S_5, inf_2 and inf_3 are in S_6. However for *Sequence 2*, inf_2 is in S_4, inf_1 and inf_3 are in S_5, and f_6 and dif_1 are in S_6. Also observe that the total number of vertexes in the k*-th* level are identical in both SPIGs. That is, given two SPIG sets $\mathcal{S}_i$ and $\mathcal{S}_j$, generated by two distinct sequence of formulation steps for the query q, $N_i(k) = N_j(k)$. For instance, the total number of vertexes in the fourth level is 5 in Figures 7 and 8.

6 Substructure Similarity Search

In this section, we elaborate on the similarity search procedure. We begin by describing SPIG-based candidates generation for exact substructure search (*Exact-SubCandidates* procedure). Note that this procedure will be exploited by substructure similarity search and our query modification strategy.

6.1 Exact Substructure Candidates Set Generation

Algorithm 3 outlines the SPIG-based procedure for retrieving R_q at a specific step. Given the target vertex v in the SPIG S_ℓ representing the query fragment q, if v represents a frequent fragment, then it retrieves FSG identifiers of v from A^{2}F-index (Lines 1-3). Otherwise, if v represents a DIF, then the algorithm retrieve the FSG identifiers from A^{2}I-index (Lines 4-6). If v represents a NIF then for each identifier in the frequent subgraph id set ($\Phi(v)$) and DIF subgraph id set ($\Upsilon(v)$) of v, it retrieves the corresponding FSG identifiers from A^{2}F-index and A^{2}I-index, respectively, and then intersect them with R_q to generate the candidate set (Lines 8-9).

6.2 Similar Substructure Candidates Set Generation

A key challenge in substructure similarity search is that the similar subgraph verification for a large candidate set is prohibitively expensive [20]. Our strategy for reducing the verification cost is as follows: (a) retrieve only candidates that are "nearly" similar to the query fragment and (b) identify verification-free candidates among them.

Algorithm 4 describes the *SimilarSubCandidates* procedure. It separates the candidate set into two parts, namely R_{free} and R_{ver}. R_{free} stores the identifiers of verification-free candidate graphs whereas R_{ver} stores identifiers of candidate data graphs that need verification. Given the subgraph distance threshold σ, the algorithm exploits the SPIG set $\mathcal{S}$ to identify the relevant subgraphs of q that need to be matched for retrieving approximate candidate sets. Specifically, these subgraphs are query fragments represented by the vertices at levels $|q| - 1$ to $|q| - \sigma$ in $\mathcal{S}$ (Lines 1). Let $R_{free}(i)$ and $R_{ver}(i)$ store the verification free candidates and candidates that need verification in the *i-th* ($|q|$-$\sigma \leq i < |q|$) level of $\mathcal{S}$, respectively. For each vertex v_j in the *i-th* level, if it is a frequent fragment or DIF, then the algorithm retrieves the candidates satisfying v_j using the *ExactSubCandidates(v_j)* procedure and combine them with $R_{free}(i)$ (Lines 3-4). Otherwise, v_j is a NIF. Consequently, $R_{ver}(i)$ is computed by combining $R_{ver}(i)$ with the candidates returned by *ExactSubCandidates(v_j)* (Lines 5-6). Next, it removes the candidates that exist in both $R_{free}(i)$ and $R_{ver}(i)$ from $R_{ver}(i)$ as these are already identified as verification-free candidates (Line 7). Finally, it adds $R_{ver}(i)$ and $R_{free}(i)$ in R_{ver} and R_{free}, respectively (Line 8).

Example 5 Reconsider the SPIG set in Figure 7 generated based on the query formulation sequence in *Sequence 1* (Figure 3). In the first step, edge e_1 is added and S_1 is constructed as shown in Figure 7(a) (Line 3 in Algorithm 1). As $freqId(v_{1,1}) = 0$, *ExactSubCandidates* procedure is invoked (Line 5 in Algorithm 1) to locate it in A^{2}F-index and retrieve $fsgIds(f_0)$ as the candidate set of current query fragment (Lines 1-3 in Algorithm 3). In the second step, the SPIG S_2 is constructed. Since $freqId(v_{2,2}) = 2$ ($v_{2,2}$ is the target vertex), its FSG identifiers are again retrieved by probing A^{2}F-index using the *ExactSubCandidates* procedure. After Step 3, $v_{3,3}$ is the target vertex in S_3 and $freqId(v_{3,3}) = 5$. Hence $fsgIds(f_5)$ is retrieved by probing A^{2}F-index as the candidate set for exact substructure match. Observe that so far the query is a frequent fragment. After Step 4, the target vertex $v_{4,6}$ in S_4 is a NIF and $\Psi(v_{4,6}) = \{4, 5\}$ and $\Upsilon(v_{4,6}) = 2$. Hence, the FSG identifiers of these fragments are retrieved by executing Lines 8-9 in Algorithm 3 ($fsgIds(v_{4,6})$=$fsgIds(dif_2) \cap fsgIds(f_4) \cap fsgIds(f_5) = fsgIds(inf_1)$). Also, $|fsgIds(v_{4,6})| = 250$.

After Step 5 since the target vertex is $v_{5,6}$ in S_5 and $fsgIds(inf_4) = 0$, the user is given an option to either modify the query or relax it to a subgraph similarity query and retrieve approximate matches (Line 11 in Algorithm 1). Suppose the user chose the latter option. Then, the *SimilarSubCandidates* procedure is invoked. Assume that $\sigma = 2$. That is, in Step 5 two edges are allowed to be missed in the results of substructure similarity search ($i \in \{3, 4\}$ in Algorithm 4). Therefore, Lines 2-8 in Algorithm 4 are executed twice. $R_{free}(3)$ and $R_{ver}(3)$ are generated for the vertexes in the third levels of the SPIGs in $\mathcal{S}$ ($v_{3,3}$, $v_{4,4}$, $v_{4,5}$ and $v_{5,3}$). $R_{ver}(3) = \emptyset$ and $R_{free}(3) = fsgIds(v_{3,3}) \cup fsgIds(v_{4,4}) \cup fsgIds(v_{4,5}) \cup fsgIds(v_{5,3}) = fsgIds(f_5) \cup fsgIds(dif_3) \cup fsgIds(f_4) \cup fsgIds(dif_5)$. Observe that $|R_{free}(3)| \geq 1300$. $R_{free}(4)$ and $R_{ver}(4)$ are generated for the vertexes in the fourth levels of the SPIGs in $\mathcal{S}$ ($v_{5,4}$, $v_{5,5}$, and $v_{4,6}$). Consequently, $R_{free} = R_{free}(4) = fsgIds(v_{5,4}) \cup fsgIds(v_{5,5}) = fsgIds(f_6) \cup fsgIds(dif_1)$ (Line 4) whereas $R_{ver} = R_{ver}(4) = fsgIds(v_{4,6})$ (Lines 7-8). Observe that $|R_{free}(4)| \geq 1100$ and $|R_{ver}(4)| = 250$. If the user clicks on the `Run` icon now, at most 250 candidate graphs in R_{ver} need candidate verification. However, at least 2400 candidate graphs in R_{free} are returned directly without verification.

When another edge is added in Step 6, R_{free}=$R_{free}(4) \cup R_{free}(5)$ and R_{ver}=$R_{ver}(4) \cup R_{ver}(5)$ where

$$\begin{aligned} R_{ver}(4) &= R_{ver}(4) \cup fsgIds(v_{6,4}) \cup fsgIds(v_{6,5}) \\ &= R_{ver}(4) \cup fsgIds(inf_2) \cup fsgIds(inf_3) \end{aligned}$$

Note that $|R_{ver}| \leq 800$. Similarly, $R_{free}(5) = 0$ and $R_{ver}(5) = fsgIds(v_{5,6}) \cup fsgIds(v_{6,6}) \cup fsgIds(v_{6,7})$. Since $fsgIds(v_{5,6}) = 0$, $fsgIds(v_{6,6})$=$fsgIds(dif_0) \cap fsgIds(dif_2)$ and $fsgIds(v_{6,7})$= $fsgIds(dif_0) \cap fsgIds(dif_1)$. Also, since $|fsgIds(v_{6,7})| = 200$ and $|fsgIds(v_{6,6})| = 150$, $|R_{ver}(5)| \leq 350$. ∎

Analysis of candidate graph set. Observe that the candidate set is equal to the

Algorithm 4: ***SimilarSubCandidates***

Input: Query fragment q, σ, SPIG set $\mathcal{S}$
Output: R_{free}, R_{ver}

```
for i=|q|-1 to |q|-σ do
    foreach v_j in ith level of S do
        if freqId(v_j) ≠ ∅ or difId(v_j) ≠ ∅ then
            R_free(i) ← R_free(i) ∪ ExactSubCandidates(v_j);
        else
            R_ver(i) ← R_ver(i) ∪ ExactSubCandidates(v_j);
    R_ver(i) ← R_ver(i) − (R_free(i) ∩ R_ver(i));
    Add R_free(i) in R_free and R_ver(i) in R_ver;
```

Algorithm 5: ***SimilarResultsGen***

Input: q, R_{free}, R_{ver} and σ
Output: Ordered result set $Results$

```
for i=|q|-σ to |q|-1 do
    Results ← Results ∪ R_free(i);
    R_ver(i) ← R_ver(i) ∩ Results;
    Results ← Results ∪ SimVerify(q, R_ver(i), i);
```

union of the FSG identifiers of vertexes in the levels $|q|$-σ to $|q|-1$ of the SPIGS in $\mathcal{S}$.

LEMMA 2 *Let R_{cand} be the candidate set for a query fragment at a specific formulation step. Then,*

$$R_{cand} = \bigcup_{k=|q|-\sigma}^{|q|-1} \bigcup_{i=0}^{N(k)} fsgIds(v_i)$$

Proof 2 *(Sketch)* $\forall g_i \subset q, |g_i| \geq |q| - \sigma, \sum_{i=0}^{N(k)} v_i$ is the sum of vertices in the k-th level of $\mathcal{S}$. $\bigcup_{i=0}^{N(k)} fsgIds(v_i)$ is its candidates. Therefore, the candidates satisfying σ edge missing are given as follows. $\bigcup_{k=|q|-\sigma}^{|q|-1} \bigcup_{i=0}^{N(k)} fsgIds(v_i)$.

Notably, the query formulation sequences *do not* have any effect on the candidate graphs set for both subgraph containment and similarity queries. That is, given two SPIG sets $\mathcal{S}_i$ and $\mathcal{S}_j$ of a query q, $R_{cand}(i) = R_{cand}(j)$. *Consequently, different formulation sequences do not have significant effect on the* SRT *as it is primarily influenced by the size of candidate set.* In fact, the time take to evaluate a query fragment at each formulation step is only slightly effected by different SPIG construction for different formulation sequences. Our empirical study in Section 8 confirms this argument.

Algorithm 6: *SimVerify*

```
   Input: query q, candidate graph g, σ, M(s)
   Output: the matching result
1  if M(s) covers |q|-σ edges of q then
2      return true;
3  else
4      Compute the candidate pairs set P(s) from q, g;
5      foreach (n, m) ∈ P(s) do
6          if F(s, n, m) then
7              s' = s ∪ (n, m);
8              SimVerify(s');
9      Restore M(s);
```

6.3 Generation of Approximate Query Results

Typically, the data graphs in the result set of a substructure similarity search have varying degree of similarity with respect to the query graph. Therefore, it is important to order these result matches. We order them based on the following rule. Let g_1 and g_2 be two candidate graphs that approximately match the query q. If $dist(g_1, q) < dist(g_2, q)$ then $Rank(g_1) < Rank(g_2)$. Note that a lower rank of g indicates that g is more similar to q.

Algorithm 5 outlines the procedure for generating ordered query results. As the subgraph distance of candidate graphs associated with the *i-th* level of SPIGs in $\mathcal{S}$ is $|q| - i$, the higher level the candidate graph is in $\mathcal{S}$, the more similar it is to the query graph. For the candidate graphs that are associated with level i, firstly the verification-free candidates ($R_{free}(i)$) are added in $Results$ (Line 2). Next, it generates the result set from the candidates in $R_{ver}(i)$ (Lines 3-4). Here we extend VF2 [4] to handle MCCS-based similarity verification. This procedure is encapsulated by the *SimVerify* procedure (discussed below). The verified candidates are then added to $Results$ (Line 4). The aforementioned procedure is repeatedly executed up to (|q|-1)*-th* level of the SPIGs. The results are returned ordered by increasing σ values.

***SimVerify* method.** The *SimVerify* procedure is outlined in Algorithm 6, which is a simple extension of VF2 [4]. Given q=(N_1, E_1), g=(N_2, E_2), M denotes a set of pairs (n, m) ($n \in N_1, m \in N_2$). A mapping $M \subset N_1 \times N_2$ is said to be a subgraph isomorphism *iff* M is an isomorphism between q and g. Let $M(s) \subset M$ contains only the nodes of q and g associated with state s. If $M(s)$ covers |q|-σ edges of q, the algorithm returns `true` (Lines 1-2). Otherwise, it computes all the possible pairs candidate $P(s)$ for current state s. $P(s)$ is obtained by considering the sets of nodes directly connected to q and g (Line 4). For each pair (n, m) in $P(s)$, if it satisfies the *feasibility Rules* [4], where $F(s, n, m)$ is a boolean function to prune the search tree, then it updates s to s' by combining (n, m). Next, it calls $SimVerify$ to verify the new state s' (Lines 5-8). If there is no possibility of state

s' to reach the required coverage, then the algorithm restores the state s for new candidate pairs computation (Line 9).

It is worth mentioning that our focus here is not to develop an efficient similar subgraph verification technique. In fact, we can easily replace the implementation of *SimVerify* with a more efficient technique (e.g., [17]). Fortunately, in spite of using such a simple verification technique, PRAGUE has very good performance due to its superior candidates pruning ability as well as its ability to exploit GUI latency (demonstrated in Section 8).

7 Supporting Query Modification

In PRAGUE a user may modify a visual query due to two key reasons: (a) if the candidate set of the formulated query fragment is empty then she may modify the query when prompted by the system (Lines 6-7 in Algorithm 1); (b) she may commit a mistake or may change her mind during query formulation and modify the query fragment accordingly (Lines 10-11 in Algorithm 1). We now discuss how such query modification is efficiently supported.

In the current version of PRAGUE, modification to a query is achieved by edge deletion[5]. The user can delete any edge as long as the modified query graph is a connected graph at all times. For clarity, we introduce our query modification algorithm based on single edge deletion at a time. It is trivial to extend it to support multiple edge deletions.

Algorithm 7 outlines our SPIG-based strategy for handling query modifications. Let e_ℓ be the most recently added edge in q and e_d be the edge deleted from q where $0 < d \leq \ell$. When the candidate set of subgraph containment query fragment q becomes empty and the user opts for query modification then Lines 3-8 are executed to provide modification suggestion to her. For each possible deleted edge in q, it matches the corresponding vertex v_i in the $|q'|$-*th* level of the SPIGs in $\mathcal{S}$ by performing the graph isomorphism test of q' and v_i. Note that two graphs g and g' are isomorphic to each other, if and only if $cam(g) = cam(g')$ [7]. It recommends the edge, that returns the largest candidate set $R_{q'}$, for deletion to the user (Lines 6-8). On the other hand, if e_d is selected by the user at any time during query formulation, the new query fragment q' is formed by deleting e_d from q (Line 11). The SPIG set $\mathcal{S}$ is updated by removing SPIGs and vertexes related to e_d (Lines 12-14). Finally, if modification occurs when the query fragment is already a subgraph similarity query, then the new candidate set is generated by *SimilarSubCandidates* procedure (Line 16). Otherwise, the candidate set is generated by invoking the *ExactSubCandidates* procedure.

Example 6 Consider Step 5 of *Sequence 1* in Figure 3. The state of the query fragment is depicted in Figure 9(a). Assume that the user selects the query modification option when prompted by PRAGUE (Figure 9(b)). Since $|fsgIds(v_{5,4})| =$

[5]Node relabeling can be expressed as deletion of edge(s) following by insertion of new edge(s) and node.

Algorithm 7: ***QueryModification***

Input: Query q, Deleted edge e_d, Spindle-shaped graph set $\mathcal{S}$, R_q, σ
Output: R_q

Initialize e_d to be deleted edge;
if R_q=$\emptyset$ *and* $e_d = \varnothing$ **then**
  foreach $e_i \subset q$ **do**
    $q' \leftarrow q - e_i$;
    $v_i \leftarrow$Match q' in the $|q'|$-th level of $\mathcal{S}$;
    if $|fsgIds(v_i)| > |R_{q'}|$ **then**
      $e_d \leftarrow e_i$;
      $R_{q'} \leftarrow fsgIds(v_i)$;
else
  $e_d \leftarrow$ edge deleted by the user;
  $q' \leftarrow q - e_d$;
Remove S_d from $\mathcal{S}$;
foreach $v_i \in S_j, e_d \in \mathcal{L}_E(v_i)$ **do**
  $S_j \leftarrow$ Delete v_i and its edges in S_j;
if R_q=$\emptyset$ **then**
  SimilarSubCandidates(q', σ, $\mathcal{S}$);
else
  ExactSubCandidates(q');

$|fsgIds(f_6)| = 1300$ is larger than both $|fsgIds(v_{4,6})|$ and $|fsgIds(v_{5,5})|$ in the fourth level of the SPIGs in Figure 7, q' is modified to f_6 and the edge 3 is suggested for deletion (Figure 9(c)). As we shall see in the next section, PRAGUE can finish computing this process by exploiting the latency offered by the GUI in the current step. Figure 9(d) shows the modified query fragment q' after the user accepted the suggestion. At the same time, the spindle-shaped graph set is updated by removing S_3 and updating the SPIGs S_4 and S_5 by deleting the vertexes with edge 3 in their Edge Lists. The updated $\mathcal{S}$ is shown in Figure 10.

Now suppose the user chooses to invoke substructure similarity search instead at Step 5 and then deletes edge 6 after Step 6. Now q' matches $v_{5,6}$ and the target vertex of S_5. Hence, the updated $\mathcal{S}$ consists of $(\mathcal{S} - S_6)$. At last, the new candidates are calculated based on this updated $\mathcal{S}$. ∎

8 Performance Study

PRAGUE is implemented in Java JDK 1.6 and the results display component is implemented using *ZGRViewer* [13]. We run all experiments on an AMD 3.4GHz machine with 3.25GB RAM, running Ubuntu 9.10 system. Since there is no existing system that realizes our new paradigm in the context of substructure similarity search, we are confined to compare PRAGUE (denoted by PRG for brevity) against the following state-of-the-art MCS-based substructure similarity search methods

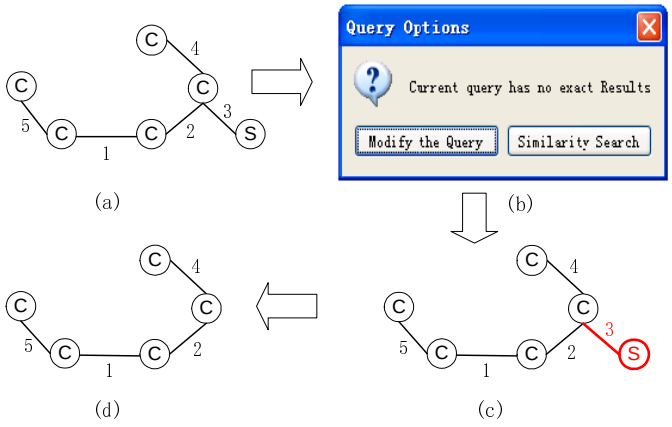


Figure 9: The query modification procedure in Step 5.

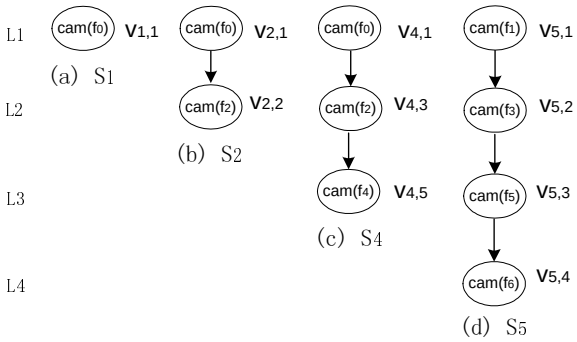


Figure 10: The updated SPIG set after deleting edge 3 in Step 5.

based on traditional paradigm: *Grafil* [20] (denoted by GR), SIGMA [12] (denoted by SG), and *restricted version*[6] of *DistVP* [17] (denoted by DVP). These programs are all implemented in C++. We do not compare against [23] as it performs substructure similarity search based on edit distance.

8.1 Experimental Setup

Datasets. Two kinds of datasets were used for our experimental study: one real dataset and a series of synthetic datasets. We use the AIDS Antiviral dataset containing 40K (40,000) graphs as real-world dataset. It has been widely used by many graph querying techniques (e.g., [12, 14, 20]). The average size of a graph is 25 vertices and 27 edges. The maximum size of a graph is 222 vertices and 251 edges. We use the *Graphgen* of FG-Index [3] to generate five synthetic datasets with sizes from 10,000 to 80,000 (denoted by 10K - 80K). Note that all existing substructure similarity search techniques [6, 12, 17, 20, 23] have used datasets containing at most 10K data graphs. In our study, we push the boundary of the number

[6]The publicly-available executable file limits our performance evaluation due to problems highlighted later. We do understand that such problems may exist as it is not an official release version.

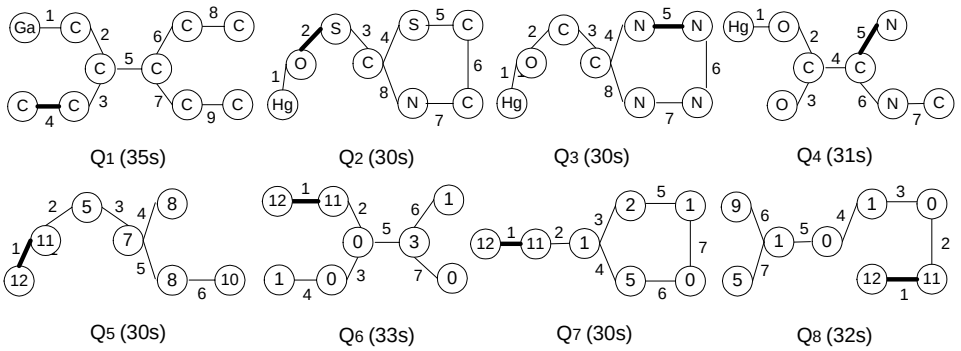


Figure 11: Queries on real and synthetic datasets.

Table 2: Index size comparison (MB)

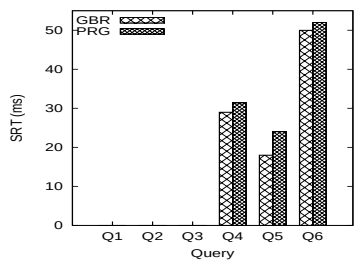
	DVP				PRG	SG/GR
σ	**1**	**2**	**3**	**4**		
Size	179.5	381.4	630.4	918.7	36.1	11.1

of graphs by a factor of 8 to test scalability. We set the number of distinct labels to 10. The avg. number of graph edges in each dataset is set to 30 and the avg. graph density is 0.1.

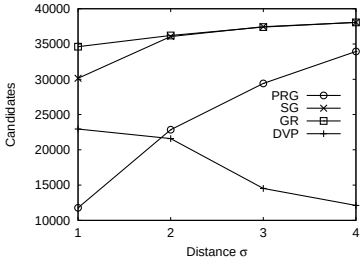
Query Sets. $Q_1 - Q_4$ are queries on the AIDS dataset whereas $Q_5 - Q_8$ are queries on the synthetic datasets. Since these queries are formulated by end users using the visual interface, it is not realistic to expect a user to formulate large queries visually. Therefore, we chose query graphs whose sizes do not exceed 10. Note that PRG can handle larger query graphs gracefully. Additionally, unlike traditional approaches [6, 12, 17, 20, 23] where the benchmark queries are automatically generated from the graph database, the queries here are visually formulated by real end users. Hence, it is not possible to generate a large number of visual queries as our preliminary study revealed that such aspiration strongly deters end users to participate in the empirical study.

The labels on the edges of a query in Figure 11 represent the default sequence of steps for query formulation in PRG. For example, in Q_3 the default sequence of steps for query formulation is: $[(\mathsf{Hg}, \mathsf{O}), (\mathsf{O}, \mathsf{C}), (\mathsf{C}, \mathsf{C}), (\mathsf{C}, \mathsf{N}), (\mathsf{N}, \mathsf{N}), (\mathsf{N}, \mathsf{N}), (\mathsf{N}, \mathsf{N}), (\mathsf{C}, \mathsf{N})]$. Unless mentioned otherwise, we shall be using the default sequence for formulating a particular query. The specific step in a query when R_q becomes empty is shown by **bold** edge (e.g., Step 4 in Q_1).

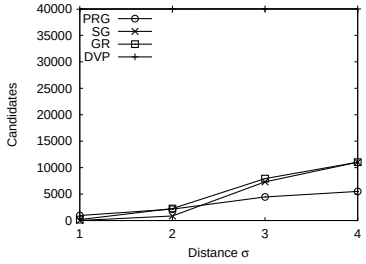
Recall that the candidate set of PRG consists of two parts: R_{free} and R_{ver}. Obviously, the more candidates are in R_{free}, the better it is for PRG as these candidates are verification-free. Hence, we chose the query set to study *best* and *worst* case behaviors of PRG with respect to R_{free} and R_{ver}. Specifically, all candidates



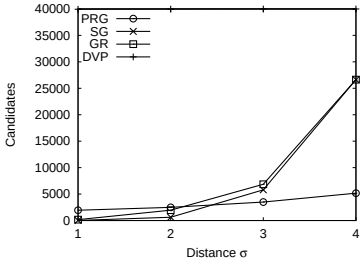
(a) SRT (msec.) of queries in [10]



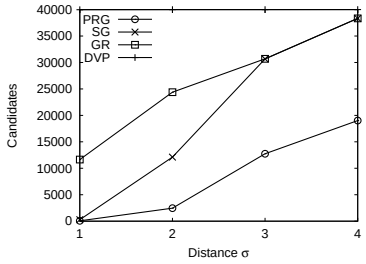
(b) Candidate size (Q_1)



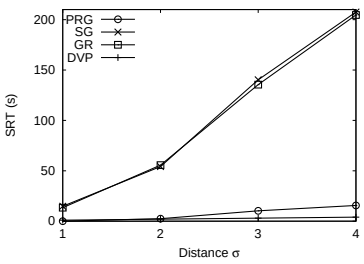
(c) Candidate size (Q_2)



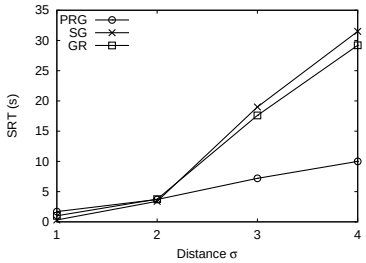
(d) Candidate size (Q_3)



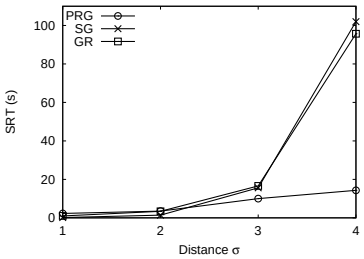
(e) Candidate size (Q_4)



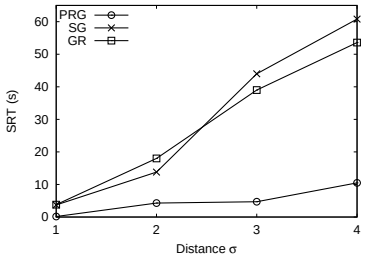
(f) SRT of Q_1 (in sec.)



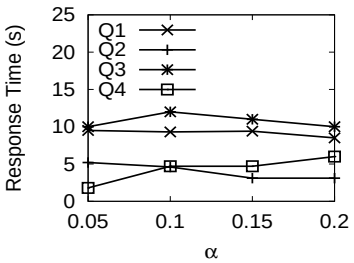
(g) SRT of Q_2 (in sec.)



(h) SRT of Q_3 (in sec.)



(i) SRT of Q_4 (in sec.)



(j) Effect of α (in sec.).

Figure 12: Experimental results for real dataset.

Table 3: Query modification costs for AIDS dataset (msec.).

Query	e_4	e_5	e_6	e_7	e_8	e_9
Q_1	20	36	36	36	37	37
Q_2	0	0	0	15	15	
Q_3	0	0	0	0	0	
Q_4	16	16	16	16		

of Q_1 is in R_{free} ("best" case). In contrast, all candidates of Q_2, Q_3, $Q_5 - Q_8$ are in R_{ver} ("worst" case). For Q_4, the candidate data graphs are scattered in both R_{ver} and R_{free}.

Participants profile. Eight unpaid male volunteers (ages from 21 to 27) participated in the experiments. None of them are familiar with any graph query languages. They were first trained to use the GUI of PRG. For every query, the participants were given some time to determine the steps that are needed to formulate the query visually. This is to ensure that the effect of thinking time is minimized during query formulation. Note that faster a user formulates a query, the lesser time PRG has for SPIG construction. The participants were given one query at a time. That is, only after the correct formulation of the current query, a participant was given the next query. If an error was committed by a participant then that particular formulation effort is ignored and he had to start afresh. Each query was formulated five times by each participant (using the default sequence unless specified otherwise) and reading of the first formulation was ignored. The average query formulation time (QFT) for a query by all participants is shown in parenthesis in Figure 11.

8.2 Performance on Real Graph Dataset

We discuss the performance of PRG on the AIDS dataset from a variety of aspects. For the AIDS dataset, we set $\alpha = 0.1$, $\beta = 8$ for PRG and $\sigma = 3$ for all techniques unless specified otherwise. Note that we do not study the effect of different values of β here as in [10] we have demonstrated that it has negligible effect on frequent subgraph containment queries (candidate pruning depends on frequent fragments). For subgraph similarity query, the candidate pruning is mainly based on DIFs. Hence, the variation of β has even lesser effect on similarity queries. For other parameters, we use the default settings of GR, SG, and DVP as suggested in [20], [12] and [17], respectively.

Index size comparison. Table 2 shows the index sizes of PRG, GR, SG, and DVP. Note that GR and SG use the same indexing scheme. Except DVP, all the indexing strategies of representative systems are independent of σ. Observe that the index size of DVP is significantly larger than PRG for all σ (highest observed factor being 25).

SPIG-based subgraph containment query performance. Recall that if a query has exact matches, then PRG will invoke Algorithm 3. However, in contrast to the

exact subgraph matching strategy in [10] (denoted by GBR), Algorithm 3 generates exact matches by exploiting the SPIGs. Hence, we compare PRG and GBR here over subgraph containment queries. We use the subgraph containment queries used for empirical study **in [10]** (denoted by Q_1 - Q_6 in [10]) as **test queries**. Figure 12(a) depicts the query performance. The *average* SRT is computed by taking the average of the SRTs of all participants (last four formulations). In the sequel, SRT of PRG refers to this average SRT unless specified otherwise.

Observe that the SRT of PRG is similar to GBR (SRTs of Q_1-Q_3 are less than $0.1ms$). This is favorable to PRG as it can support a unified framework for both subgraph containment and similarity queries without sacrificing performance of the former type of queries in comparison to GBR.

Candidate size and system response time (SRT). Next, we compare the performances of the representative systems for evaluating subgraph similarity queries by varying σ from 1 to 4. Figures 12(b)-(e) report the candidate sizes of representative queries for different values of σ. Note that in PRG, GR, and SG, candidate size refers to $|R_{free} \cup R_{ver}|$. In fact, GR and SG do not separate the candidates into these two categories. However, candidate size in DVP refers to $|R_{ver}|$ only[7]. Observe that for most cases the candidate size of PRG is significantly less than GR, SG, and DVP. In Figures 12(c) and (d) ("worst" case queries), although the candidate size of PRG is larger than GR and SG when $\sigma \in \{1, 2\}$, it becomes less than these approaches when σ increases to 3 and 4. The candidate pruning of PRG depends on the DIFs and frequent fragments. Typically, DIFs have stronger pruning ability. In contrast, pruning of SG and GR mainly depends on the frequent fragments. In the worst cases, there are less DIFs in the queries with smaller σ, which weakens pruning ability of PRG. Additionally, the candidate sizes of DVP in Q_1 ("best" case) is significantly lesser than PRG for $\sigma \in \{3, 4\}$. This is primarily because DVP only reports $|R_{ver}|$. For Q_1, $|R_{ver}| = 0$ in PRG. For Q_2-Q_4, the candidate sizes of DVP are close to the entire dataset ($\sim$ 40K).

Figures 12(f)-(i) report the SRTs for different values of σ. In GR, SG, and DVP, SRT refers to the execution time of a query. Each query was executed five times in each approach and the results from the first run were always discarded. Observe that we only display the SRTs of DVP for Q_1 only. This is because in contrast to the remaining approaches, DVP returns empty results for the remaining queries[8].

It is evident that the performance of PRG is better than the existing strategies. Although in Figures 12(g) and (h) (worst case queries), the SRT of PRG is a little bit longer than GR and SG for $\sigma \in \{1, 2\}$, it is less than these approaches for larger σ. SG/GR converts the subgraph similarity verification problem to the exact subgraph isomorphism verification problem. The latter is typically faster than the former. In the worst cases, all the candidates in PRG need to be verified. Note that SG/GR loses this advantage when σ increases as they have to perform a large number of exact subgraph verification. More importantly, the SRT of PRG grows gracefully with

[7] The current version of DVP program outputs only $|R_{ver}|$.

[8] We have also manually verified that the result sets are indeed non-empty.

Query	*Sequence*	*Step1*	*Step2*	*Step3*	*Step4*	*Step5*	*Step6*	*Step7*	*Step8*	*Step9*	Avg. SRT
Q_1	1,2,3,4,5,6,7, 8, 9	0	0	0	0	0.004	0.006	0.045	0.069	0.477	9.3
	1,2,3 4,5,6,8,7, 9	0	0	0	0	0.005	0.007	0.021	0.285	0.478	9.5
Q_2	1,2,3,4,5,6, 7, 8	0	0.15	0.3	0.134	0.343	0.21	0.312	0.24		8.2
	1,2,3,8,7,6, 5, 4	0	0.15	0.3	0	0.135	0.3	0.34	0.35		8.4
Q_3	1,2,3,4,6,8, 7, 5	0.002	0.003	0	0	0	0.004	0.004	0.168		10
	3,2,1,5,7,8, 6, 4	0	0.002	0.002	0	0	0.002	0.004	0.128		10.2
Q_4	1,2,3,4,5,6,7	0.1	0.32	0.15	0.21	0.35	0.12	0.32			4.8
	1,2,4,3,6,7,5	0.1	0.32	0.33	0.14	0.2	0.4	0.15			5
Q_5	1,2,3,4,5,6	0.015	0	0	0.01	0.015	0.016				0.58
	1,2,3,5,6,4	0.015	0	0	0.03	0.02	0				0.54
Q_6	1,2,3,4,5,6,7	0	0	0.01	0.03	0.01	0.015	0.024			1.1
	4,3,2,1,5,7,6	0.015	0	0.015	0.015	0.026	0	0.03			1
Q_7	1,2,3,4,5,6,7	0.01	0.014	0	0.015	0.015	0.02	0.015			0.73
	1,2,4,6,7,5,3	0.01	0.014	0.02	0.032	0.016	0	0.015			0.6
Q_8	1,2,3,4,5,6,7	0.023	0.013	0.023	0.02	0.013	0.016	0.015			0.62
	7,6,5,4,3,2,1	0.015	0	0.015	0.03	0.02	0	0.023			0.54

Table 4: Effect of variation in query formulation sequence on SPIG construction (in sec.)

σ. Lastly, *only* PRG orders the query results according to their subgraph distance. Inevitably, this increases the SRT of PRG.

Query modification costs. We now compare the cost of modifying a visual query using Algorithm 7. We vary the steps when a user performs modification, namely from addition of the 4-*th* edge (e_4) to the 9-*th* edge (e_9) if any. We always delete the first edge (e_1) from $Q_1 - Q_4$ to simulate worst case scenario. Table 3 reports the performances. Observe that the modification cost of PRG is cognitively negligible (virtually "zero"). This also implies that the cost of updating the SPIG set is negligible. Since the time taken to construct an edge in PRG typically is at least 2 seconds, query modification can easily be completed by exploiting the GUI latency.

SPIG construction cost and query formulation sequence. Next, we assess the effect of different formulation sequences on the SPIG construction time and SRT. Table 4 lists different formulation sequences for $Q_1 - Q_8$ and the average time (all participants) to construct the SPIGs at different steps. Observe that the SPIG construction process at each step is very efficient and takes negligible time. It is significantly lower (almost an order of magnitude) than the available GUI latency (at least two seconds to draw an edge[9]). Also, SPIG construction is not adversely affected by addition of new edges to a query fragment. In summary, SPIGs can easily be constructed by exploiting the latency offered by the GUI. Lastly, the formulation sequences only have minor effect on the SPIG construction time and

[9]Here we ignore the 'user thinking time". As the thinking time increases, the latency offered by the GUI increases as well at each step.

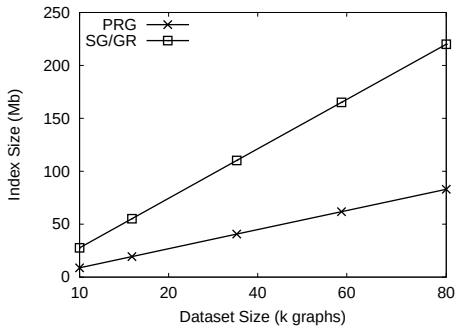


Figure 13: Index size comparison.

Table 5: Query modification cost for synthetic dataset (msec).

Query	**10K**	**20K**	**40K**	**60K**	**80K**
Q_5	0	0	0	16	16
Q_6	0	0	0	0	15
Q_7	0	0	0	15	30
Q_8	0	0	15	30	40

SRT highlighting the robustness of our technique.

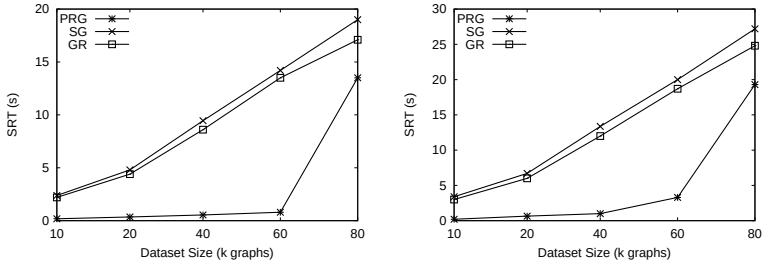
Effect of α. Lastly, we compare the performance of PRG under different values of α from 0.05 to 0.2. α affects the number of frequent fragments and DIFs built in the action-aware indices and also the distribution of candidates in R_{free} and R_{ver}. Figure 12(j) reports the SRT of Q_1-Q_4 in Figure 11 for different values of α. Observe that the SRTs fluctuate in a small range in most cases with the variations of α. Notice that although the SRTs of subgraph containment queries (Figure 12(a)) is in the order of *msec*, the response time of similarity queries in Figure 12(j) is in the order of *seconds*. This is because the candidate sets of similarity queries are typically larger than that of containment queries. Besides, the verification cost of a similarity query is typically higher than that of a containment query.

8.3 Performance on Synthetic Graph Dataset

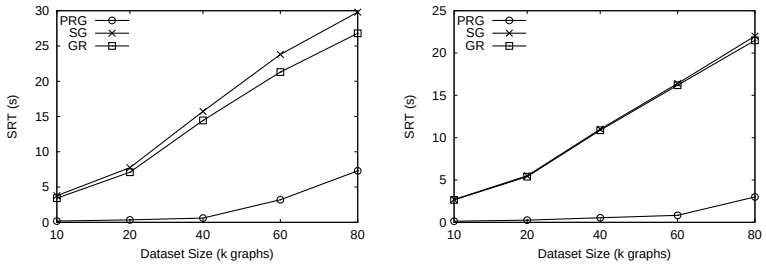
We now assess the scalability of PRG using the synthetic datasets and the queries Q_5 to Q_8. For synthetic datasets, we set $\beta = 4$ and $\alpha = 0.05$ for PRG and $\sigma = 3$ for PRG, SG and GR. Note that since DVP failed to build indices for the synthetic datasets[10], we are not able to compare its performance here.

Size of indexes. Figure 13 reports the size of indexes with increase in dataset size. Observe that the index size of PRG increases slowly and is smaller than SG/GR for all datasets.

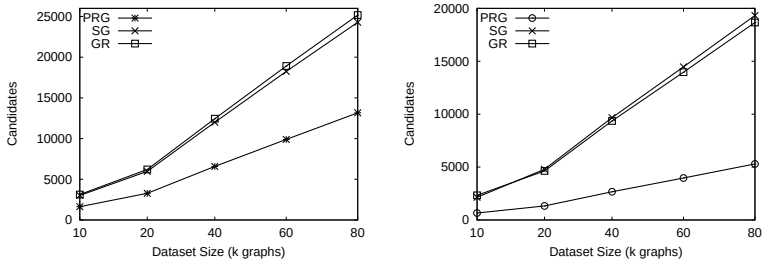
[10] DVP simply exits index building. No specific error message is displayed.



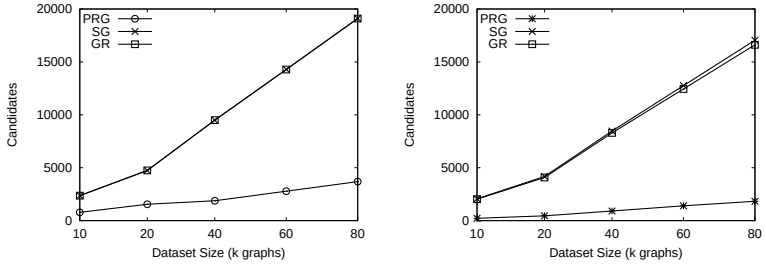
(a) SRT in sec. (Q_5) (b) SRT in sec. (Q_6)



(c) SRT in sec. (Q_7) (d) SRT in sec. (Q_8)



(e) Candidate size (Q_5) (f) Candidate size (Q_6)



(g) Candidate size (Q_7) (h) Candidate size (Q_8)

Figure 14: Experimental results for synthetic datasets.

SRT and size of candidate graphs. Figures 14 depicts the SRTs and sizes of candidate graphs of $Q_5 - Q_8$ for the five datasets. Clearly, SRT of PRG is lower than SG and GR and it has the least candidates across all datasets and queries, confirming the strengths of PRG. More importantly, our proposed paradigm enables PRG to scale gracefully. Note that the sharp increase in SRT for Q_5 (for 80K dataset) in PRG is primarily due to the simple verification method we have used rather than its candidates pruning ability. As mentioned earlier, a more efficient similarity verification technique (e.g., [17]) can significantly lower the SRTs of PRG even further.

Query modification cost. Table 5 reports the modification costs of $Q_5 - Q_8$. For each query we modify at the last step and the first edge is always deleted. Observe that the modification is very efficient for PRG and scales gracefully across all datasets. Importantly, it can be easily completed during the latency provided by the GUI.

9 Conclusions and Future Work

In this paper, we have presented PRAGUE - a practical and unified visual framework that supports processing of modification-efficient subgraph containment and similarity queries by blending their evaluation with visual query formulation. It employs a data structure called SPIG, which succinctly records various information related to the set of supergraphs of newly added edge in the visual query fragment. These information along with the latency offered by the GUI-based query formulation are exploited by our innovative subgraph query evaluation algorithms and query modification technique to efficiently retrieve and update candidate data graphs. Importantly, it gracefully accommodates modifications to a visual query during construction. All these features are important for deployment of PRAGUE in real-world environment. Experimental studies on real and synthetic graphs validated the practical merit and superiority of PRAGUE.

We intend to extend PRAGUE to support *richer* variety of queries (e.g., query nodes representing variables, selection predicates). Also, we wish to support more advanced GUI that allows users to add *canned subgraphs* at each visual step.

Acknowledgments. The authors thank X. Yan for providing *gSpan*; H. Shang for providing *DistVP*; M. Mongiovi for providing *Grafil* and SIGMA.

References

[1] S. Abiteboul, R. Agrawal, P. Bernstein et al. The Lowell Database Research Self-Assessment. *In CACM*, 48(5), 2005.

[2] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *In Pattern Recognition Letters*, 1998.

[3] J. Cheng, Y. Ke, W. Ng, A. Lu. FG-Index: Towards Verification-Free Query Processing On Graph Databases.*In SIGMOD*, 2007.

[4] L.P. Cordella, P. Foggia, C. Sansone, M. Vento. An improved algorithm for matching large graphs. *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, pages 149-159*, 2001.

[5] H. He, A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. *In SIGMOD*, 2008.

[6] H. He, A. K. Singh. Closure-Tree: An Index Structure for Graph Queries. *In ICDE*, 2006.

[7] J. P. Huan, W. Wang. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In ICDM*, 2003.

[8] H. V. Jagadish, et al. Making Database Systems Usable. *In SIGMOD*, 2007.

[9] H. Jiang, H. Wang, P. S. Yu, S. Zhou. GString: A Novel Approach for Efficient Search in Graph Databases. *In ICDE*, 2007.

[10] C. Jin, et al. GBLENDER: Towards Blending Visual Query Formulation and Query Processing in Graph Databases. *In ACM SIGMOD*, 2010.

[11] U. Leser. A Query Language for Biological Networks. *In Bioinformatics*, 21:ii33–ii39, 2005.

[12] M. Mongiovi, R. Di Natale, et al. SIGMA: A Set-cover-based Inexact Graph Matching Algorithm. *In J. of Bioinformatics and Comp. Biology*, 2010

[13] E. Pietriga. A Toolkit for Addressing HCI Issues in Visual Language Environments.*In IEEE Symp. on Vis. Lang. and Human-Centric Comp.*, 2005.

[14] W.-S Han et al. iGraph: A Framework for Comparisons of Disk Based Graph Indexing Techniques. *In VLDB*, 2010.

[15] H. Shang, Y. Zhang, X. Lin, J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *In VLDB*, 2008.

[16] H. Shang, K. Zhu, X. Lin, Y. Zhang, R. Ichise. Similarity Search on Supergraph Containment. *In ICDE*, 2010.

[17] H. Shang, et al. Connected Substructure Similarity Search. *In SIGMOD*, 2010.

[18] Y. Tian, R. C. McEachin, C. Santos, et al. SAGA: A subgraph matching tool for biological graphs. *In Bioinformatics*, 2006.

[19] J. R. Ullman. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1), 1976.

[20] X. Yan, et al. Substructure Similarity Search in Graph Databases. *In SIGMOD*, 2005.

[21] X. Yan, J. Han. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.

[22] X. Yan, et al. Graph Indexing: A Frequent Structure-Based Approach. *In SIGMOD*, 2004.

[23] Z. Zeng, A. K. H. Tung, J. Wang, et al. Comparing Stars: On Approximating Graph Edit Distance. *In VLDB*, 2009.

[24] S. Zhang, et al. TreePi: A Novel Graph Indexing Method. *In ICDE*, 2007.

[25] S. Zhang, et al. GADDI: Distance Index Based Subgraph Matching in Biological Networks. *In EDBT*, 2009.

[26] P. Zhao, et al. Graph Indexing: Tree + delta $\geq$ Graph. *In VLDB*, 2007.

[27] P. Zhao, J. Han. On Graph Query Optimization in Large Networks. *In VLDB*, 2010.

[28] J. L. Zou, et al. A Novel Spectral Coding in a Large Graph Database. *In EDBT*, 2008.