PRAGUE: Towards Blending Practical 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

***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 [6]. Our first attempt at implementing this vision, called GBLENDER [6], 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 (PRactical visuAl Graph QUery blEnder), 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.**

I. INTRODUCTION

Querying graph databases has emerged as an important research problem due to explosive growth of graph-structured data in recent years. 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 [14]. 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 "approximately" contain the query graph, which is called the *substructure similarity search* problem [12] (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).

A number of graph query languages (*e.g.,* SPARQL, GraphQL [4]) 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. Unfortunately, in many real life domains (*e.g.,* life sciences) it is unrealistic to assume that users (*e.g.,* biologists) are proficient in expressing such queries.

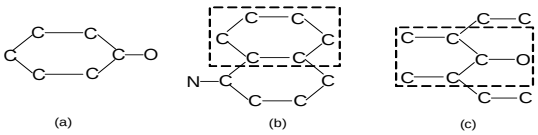


Fig. 1. A query graph (a) and data graphs ((b) and (c)).

A. 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.,* [2]). Figure 2 depicts an example of such a visual interface. A 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

[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.

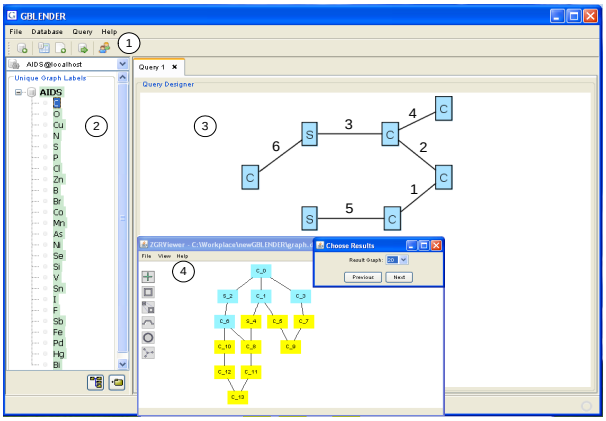


Fig. 2. Visual interface for formulating graph queries.

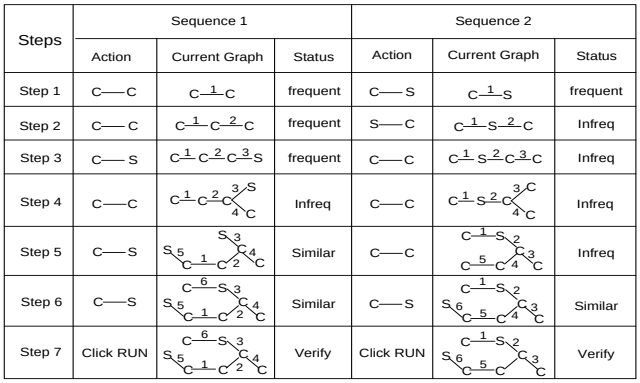


Fig. 3. Query formulation steps.

results in slower *system response time* (SRT)[2] as the query processor remains idle during query formulation.

In [6], 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 [13]. These fragments are then used to construct the *action-aware frequent index* (A^{2}F) and *action-aware infrequent index* (A^{2}I) 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 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 clicks 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 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 II). In this paper, we propose a novel framework that provides a unified solution to the aforementioned limitations.

B. Overview and Contributions

We present a novel algorithm called PRAGUE (**PR**actical visu**A**l **G**raph **QU**ery bl**E**nder) (see Section IV) 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 V for details).

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 exact matches in the database (see Section VI). 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

[2]Duration between the time a user presses the `Run` icon to the time when the user gets the query results [6].

[3]Note that there are two different research streams processing graph queries [10]. 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.

to be deleted to make R_q non-empty. 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 VII). (b) Otherwise, it invokes substructure similarity search to retrieve approximate matches to q (see Section VI).

In Section VIII, 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 [8], [11], [12]. Consequently, in spite of adopting a simple subgraph similarity verification technique [3], its SRT is often significantly smaller than these techniques.

II. Related Work

Most germane to this work is our previous effort in [6] 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 result set. Secondly, we present an efficient framework to support modifications to a visual query any time during query formulation.

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. 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.

TABLE I
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

III. Background

For the sake of completeness, in this section we briefly describe the *action-aware* indexing schemes of GBLENDER [6], which we shall be exploiting in the sequel. The key notations used in this paper are summarized in Table I.

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.

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 $G_1 \not\supseteq G_2$.

Frequent and infrequent fragments. Let $\mathcal{D}$ be a graph database containing a set of data graphs. We assign a 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 [6]. 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 are shown in parenthesis).

Given a fragment $g \in \mathcal{D}$, if $sup(g) < \alpha|\mathcal{D}|$ then g is an *infrequent* fragment [6]. 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). 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 [6]).

- 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.

Action-aware indexes. The *action-aware frequent index* (A^{2}F) 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 [5]. 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 $fsgIds(f) \cap fsgIds(f') = fsgIds(f)$ [2]. GBLENDER exploits this to store only $delId(f) \subset fsgIds(f)$.

[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.

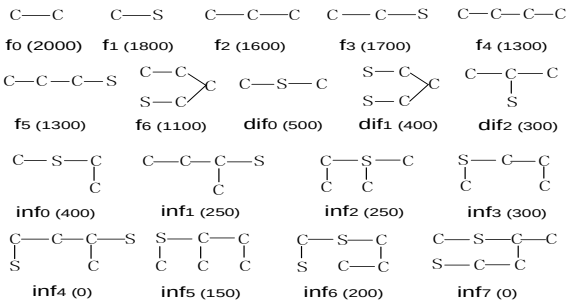


Fig. 4. Frequent and infrequent fragments.

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 the 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 an 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)).

IV. Overview of PRAGUE

A. Substructure Similarity Search Problem

Most of the existing subgraph similarity query processing techniques [8], [11], [12], [15] measure similarity between two graphs using distance measures that are either based on *graph edit distance* [15] or *maximum common subgraph* [11], [12]. 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*

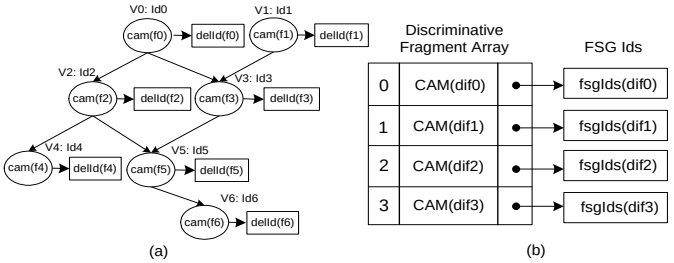


Fig. 5. Examples of MF-index and A^{2}I-index.

Subgraph (MCS) [1] between the query graph and the data graphs, and measures the similarity based on the difference of the query graph and the MCS. Grafil [12] 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 [11]. 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. [11] 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 [15], in this paper we use MCCS for similarity search for the following reasons. Firstly, as highlighted in [1] any edit distance measure critically depends on the costs of the underlying edit operations. 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 be easily depicted in the results by highlighting the MCCS in the matched data graphs.

Definition 1: (**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 allowed to be missed (deleted) in Q in order to match G.

Definition 2: (**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 3: (**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 $\delta = 5/7$. ∎

Algorithm 1: PRAGUE

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

```
1  Initialize SPIG set S = ∅;
2  if Action is New then
3      q ← q + e_ℓ;
4      S_ℓ ← SpigConstruct(q, Q, e_ℓ, S) /*Algorithm 2*/;
5      if simFlag = false then
6          R_q ← ExactSubCandidates(S_ℓ.v_target)
           /*Algorithm 3*/;
7          if R_q = ∅ then
8              Action ← OptionDialogueDisplay();
9      else
10         (R_free, R_ver) ← SimilarSubCandidates(q, σ, S)
           /*Algorithm 4*/;
11 else if Action is Modify then
12     q ←QueryModification(q, R_q, S, σ) /*Algorithm 6*/;
13 else if Action is SimQuery then
14     Set simFlag = true;
15     (R_free, R_ver) ← SimilarSubCandidates(q, σ, S);
16 else if Action is Run then
17     if simFlag = false then
18         Results ← ExactVerification(R_q);
19         if Results = ∅ then
20             (R_free, R_ver) ← SimilarSubCandidates(q, σ, S);
21             Results ← SimilarResultsGen(q, R_free, R_ver,
               σ) /*Algorithm 5*/;
22     else
23         Results ←SimilarResultsGen(q, R_free, R_ver, σ);
```

B. 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 4). If

$simFlag$ is *false*, it retrieves the FSG identifiers of q (R_q) by invoking the *ExactSubCandidates* procedure (Line 6).

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 8). 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 12). 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 15). These steps are repeated for each new edge until the user clicks the `Run` icon (Line 16). If $simFlag$ is *false*, then the exact results $Results$ are returned from the candidate graphs (Line 18). If $Results$ is empty after candidate verification (subgraph isomorphism test) then the substructure similarity search is invoked to retrieve approximate matches (Lines 20-21). 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 23).

V. SPINDLE-SHAPED GRAPH (SPIG)

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

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 III), 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 III, 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.

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 $v'_{\ell,i} \leftarrow$ Search $cam(g'_i)$ in the $|g'_i|$-th level of $S_{\ell'}$;
11 Attach $v'_{\ell,i}$'s $FragmentList$ to $v_{\ell,i}$;
12 **else**
13 Attach $v_{\ell,i}$ with $difId(g_i)$ or $freqId(g_i)$;
14 **if** $|g_i|$=$|q|$ **then**
15 Add S_ℓ in $\mathcal{S}$;
16 **return** S_ℓ;
17 **else**
18 **foreach** $g_i \subset g_j \subset q$ *and* $|g_j|$=$|g_i|$+1 **do**
19 **if** $v_{\ell,j} \notin \mathbb{Q}$ **then**
20 $v_{\ell,j} \leftarrow f(g_j)$;
21 **Enqueue**$(v_{\ell,j}, \mathbb{Q})$;
22 **Insert**$(v_{\ell,j}, S_\ell)$;
23 Connect $edge(v_{\ell,i}, v_{\ell,j})$;

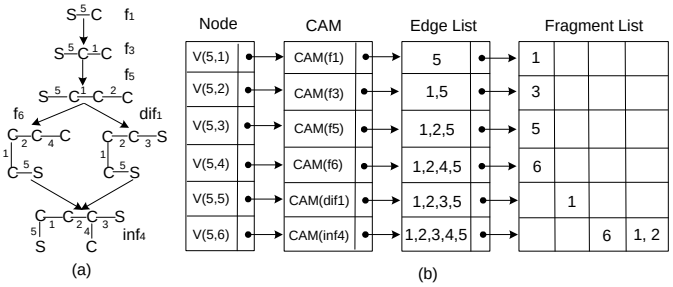


Fig. 6. The vertices of the spindle-shaped graph in step 5.

- 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 the 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 4: (**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.*

- *For each $v \in V_\ell$, $\exists$ an injective function f: $v \rightarrow f(g)$ s.t. e_ℓ is contained in g and $g \subseteq q$.*

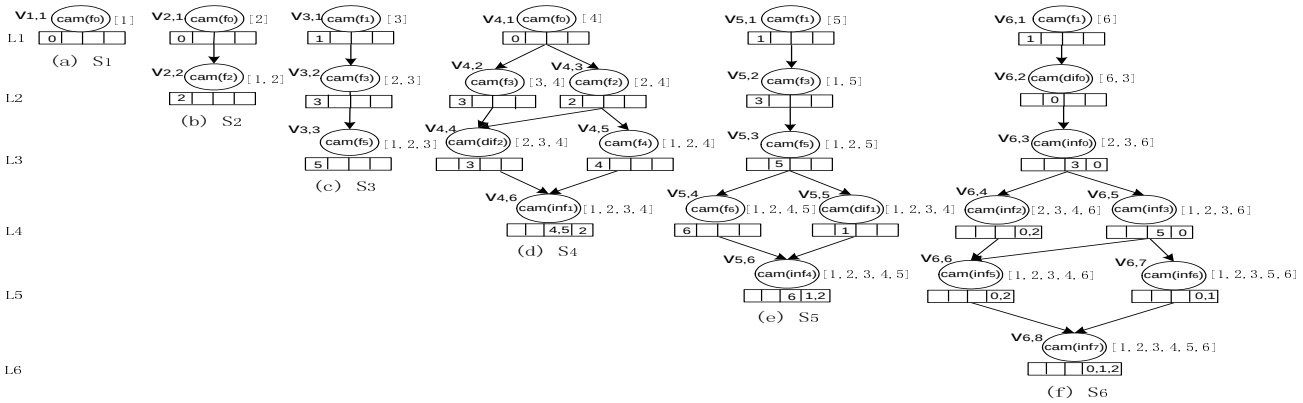


Fig. 7. The SPIG set for Sequence 1 (Edge Lists are in square brackets and Fragment Lists are shown in rectangular boxes).

- *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:*
  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 2: 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 4 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, $\Upsilon(v_{5,6}) = \{1, 2\}$. ∎

A. Algorithm for SPIG Construction

The algorithm for building a SPIG 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). 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 $S_{\ell'}$ (Lines 9-10). Then it attaches the relevant ids in $FragmentList$ of $v'_{\ell,i}$ to $v_{\ell,i}$ (Line 11). 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 them to its subgraphs and retrieving them by probing action-aware-indexes.

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 13). If $|g_i| = |q|$, then the SPIG construction process is terminated and S_ℓ is added to $\mathcal{S}$ (Lines 14-16). 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 18-23).

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

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

1 **if** $freqId(v) \neq \varnothing$ **then**
2 $i = freqId(v)$;
3 $R_q \leftarrow$ retrieve $fsgIds(g_i)$ from A^{2}F-index;
4 **else if** $difId(v) \neq \varnothing$ **then**
5 $i = difId(v)$;
6 $R_q \leftarrow$ retrieve $fsgIds(g_i)$ from A^{2}I-index;
7 **else**
8 **foreach** $i \in \Phi(v)$, $j \in \Upsilon(v)$ **do**
9 $R_q \leftarrow R_q \cap fsgIds(g_i) \cap fsgIds(g_j)$;

instance, Figure 7 shows 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.

B. 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 $\mathcal{C}_{n-1}^{k-1}$. Consequently, the total number of vertices in S_ℓ is: $\sum_{k=1}^{n} \mathcal{C}_{n-1}^{k-1}$. 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 VIII.

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: The proof is given in [7]. ■

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, $N_i(k) = N_j(k)$.

VI. Substructure Similarity Search

We begin by describing SPIG-based candidates generation for exact substructure search (*ExactSubCandidates* procedure). Note that this will be exploited by substructure similarity search and our query modification strategy.

A. 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).

Algorithm 4: ***SimilarSubCandidates***

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

1 **for** i=$|q|$*-1 to* $|q|$-σ **do**
2 **foreach** v_j *in ith level of* $\mathcal{S}$ **do**
3 **if** $freqId(v_j) \neq \varnothing$ *or* $difId(v_j) \neq \varnothing$ **then**
4 $R_{free}(i) \leftarrow R_{free}(i) \cup$ **ExactSubCandidates(**v_j**)**;
5 **else**
6 $R_{ver}(i) \leftarrow R_{ver}(i) \cup$ **ExactSubCandidates(**v_j**)**;
7 $R_{ver}(i) \leftarrow R_{ver}(i) - (R_{free}(i) \cap R_{ver}(i))$;
8 Add $R_{free}(i)$ in R_{free} and $R_{ver}(i)$ in R_{ver};

B. 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 [12]. 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 $\mathcal{S}$ (Line 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).

Analysis of candidate graph set. Observe that the candidate set is equal to the 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 at a specific formulation step. Then,* $R_{cand} = \bigcup_{k=|q|-\sigma}^{|q|-1} \bigcup_{i=0}^{N(k)} fsgIds(v_i)$.

Proof: The proof is given in [7]. ■

Notably, the query formulation sequences *do not* have any effect on the candidate graphs set for both subgraph contain-

ment 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.* Our empirical study in Section VIII confirms this argument.

C. Generation of Approximate Query Results

As the data graphs in the result set of a substructure similarity search have varying degree of similarity with respect to the query graph, 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 [3] to handle MCCS-based similarity verification. This procedure is encapsulated by the *SimVerify* procedure (the formal algorithm is given in [7]). 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.

Note 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.,* [11]). 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 VIII).

VII. 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 7-8 in Algorithm 1); (b) she may commit a mistake or may change her mind during query formulation and modify the query fragment accordingly (Lines 11-12 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 6 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

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

Algorithm 5: ***SimilarResultsGen***

Input: q, R_{free}, R_{ver} and σ
Output: Ordered result set $Results$
1 **for** i=$|q|$-σ *to* $|q|$-*1* **do**
2 $Results \leftarrow Results \cup R_{free}(i)$;
3 $R_{ver}(i) \leftarrow R_{ver}(i) \cap Results$;
4 $Results \leftarrow Results \cup$ **SimVerify**(q, $R_{ver}(i)$, i);

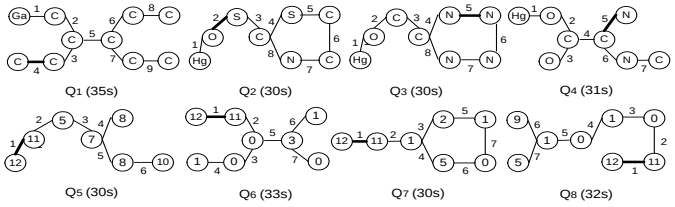


Fig. 8. Queries on real and synthetic datasets.

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 $cam(g) = cam(g')$ [5]. 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. Due to space constraint, an example illustrating this algorithm is given in [7].

VIII. Performance Study

PRAGUE is implemented in Java JDK 1.6 and the results display component is implemented using *ZGRViewer* [9]. 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 based on traditional paradigm: *Grafil* [12] (denoted by GR), SIGMA [8] (denoted by SG), and *restricted version*[6] of *DistVP* [11] (denoted by DVP). These programs are all implemented in C++.

A. Experimental Setup

Datasets. We use the AIDS Antiviral dataset containing 40K $(40,000)$ graphs as real-world dataset (similar to [8], [10],

[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.

Algorithm 6: ***QueryModification***

Input: Query q, Deleted edge e_d, $\mathcal{S}$, 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 $v_i \in S_j, S_j \in \mathcal{S}, e_d \in \mathcal{L}_E(v_i)$ **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');

TABLE II
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

[12]). 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 [2] to generate five synthetic datasets with sizes from $10,000$ to $80,000$ (denoted by 10K - 80K). The average number of graph edges in each dataset is set to 30 and the average 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. Additionally, unlike traditional approaches [8], [11], [12], [15] 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 8 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: $[(\mathsf{Hg},\mathsf{O}),(\mathsf{O},\mathsf{C}),(\mathsf{C},\mathsf{C}),(\mathsf{C},\mathsf{N}),(\mathsf{N},\mathsf{N}),(\mathsf{N},\mathsf{N}),(\mathsf{C},\mathsf{N})]$. 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 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).

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. 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 8.

B. Performance on Real Graph Dataset

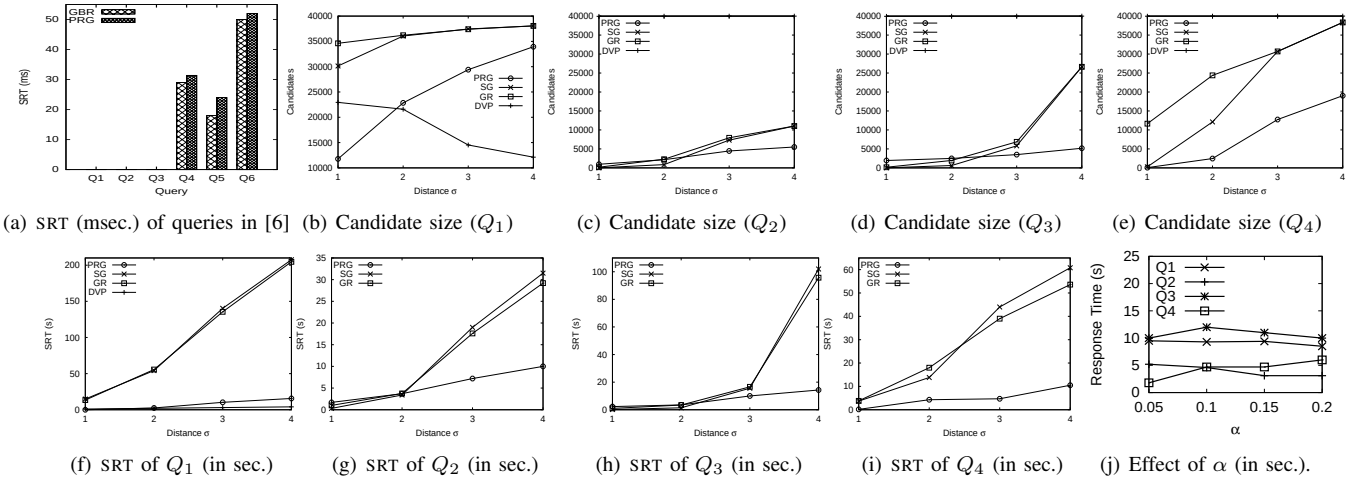
For the AIDS dataset, we set $\alpha = 0.1$, $\beta = 8$ for PRG and $\sigma = 3$ for all techniques unless specified otherwise. Note that we do not study the effect of different values of β here as in [6] we have demonstrated that it has negligible effect on frequent subgraph containment queries (candidate 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 [12], [8] and [11], respectively.

Index size comparison. Table II 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 [6] (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 [6] (denoted by Q_1 - Q_6 in [6]) as test queries. Figure 9(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.

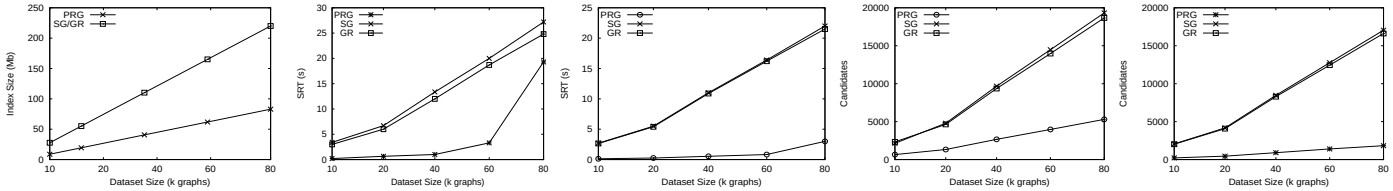
Candidate size and system response time (SRT). Next, we compare the performances of the representative systems for evaluating subgraph similarity queries by varying σ from 1 to 4. Figures 9(b)-(e) report the candidate sizes of representative queries for different values of σ. Note that in PRG, GR, and SG,



(a) SRT (msec.) of queries in [6] (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.).

Fig. 9. Experimental results for real dataset.



(a) Index size (MB) (b) SRT in sec. (Q_6) (c) SRT in sec. (Q_8) (d) Candidate size (Q_6) (e) Candidate size (Q_8)

Fig. 10. Experimental results for synthetic datasets.

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 9(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 9(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 9(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 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 6. 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 IV 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. Table III lists two different formulation sequences for Q_1 and Q_3 and the average time (all participants) to construct the SPIGs at different steps. Performances of remaining queries are similar and are reported in [7]. 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

[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.

TABLE III

EFFECT OF VARIATION IN QUERY FORMULATION SEQUENCE ON SPIG CONSTRUCTION (IN SEC.)

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	0.265	0.266	0.3	0.234	0.303	0.156	0.265	0.093	10.3
	4,3,2,1,5,6,7,8,9	0.265	0.266	0.235	0.5	0.453	0.313	0.157	0.266	0.093	10.2
Q_3	1,2,3,4,5,6,7,8	0	0.063	0.078	0	0	0.016	0	0.016		7.2
	3,2,1,4,8,5,7,6	0.19	0.177	0.22	0.016	0.031	0.016	0	0.016		7.1

TABLE IV

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		

TABLE V

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

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 be easily constructed by exploiting the latency offered by the GUI. Lastly, the formulation sequences only have minor effect on the SPIG construction time and SRT highlighting the robustness of our technique.

Effect of α. Lastly, we compare the performance of PRG for different values of α (from 0.05 to 0.2). Note that α affects the number of frequent fragments and DIFs built in the action-aware indexes and also the distribution of candidates in R_{free} and R_{ver}. We use the queries on the real-world dataset (Q_1-Q_4) as representative queries. Figure 9(j) reports the SRT of these queries for different values of α. Observe that the SRTs fluctuate in a small range with the variations of α.

C. Performance on Synthetic Graph Dataset

For synthetic datasets, we set $\beta = 4$, $\alpha = 0.05$ for PRG and $\sigma = 3$ for PRG, SG and GR. We do not compare DVP here as it failed to build indexes for the synthetic datasets[10].

Size of indexes. Figure 10(a) 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.

SRT and size of candidate graphs. Figures 10(b)-(e) depict the SRTs and sizes of candidate graphs of Q_6 and Q_8 for the five datasets. The performances of Q_5 and Q_7 are similar and are reported in [7]. 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_6 (for 80K dataset) in PRG is primarily due to the simple verification method we have used rather than its candidates pruning ability.

Query modification cost. Table V 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.

IX. CONCLUSIONS

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. Experimental studies on real and synthetic graphs validated the practical merit and superiority of PRAGUE.

Acknowledgments. The authors thank X. Yan for providing *gSpan*; H. Shang for providing *DistVP*; M. Mongiovi for providing *Grafil* and SIGMA. Shuigeng Zhou was supported by National Natural Science Foundation of China (NSFC) under grant No. 60873070.

REFERENCES

[1] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *In Pattern Recognition Letters*, 1998.
[2] J. Cheng, Y. Ke, W. Ng, A. Lu. FG-Index: Towards Verification-Free Query Processing On Graph Databases.*In SIGMOD*, 2007.
[3] L.P. Cordella, P. Foggia, et al. An improved algorithm for matching large graphs. *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.
[4] H. He, A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. *In SIGMOD*, 2008.
[5] J. P. Huan, W. Wang. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In ICDM*, 2003.
[6] C. Jin, et al. GBLENDER: Towards Blending Visual Query Formulation and Query Processing in Graph Databases. *In ACM SIGMOD*, 2010.
[7] C. Jin, et al. PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. *Technical report*, 2011 (Available at `www.cais.ntu.edu.sg/~assourav/TechReports/prague-TR.pdf`)
[8] M. Mongiovi, R. Di Natale, et al. SIGMA: A Set-cover-based Inexact Graph Matching Algorithm. *In J. of Bioinformatics and Comp. Biology*, 2010
[9] E. Pietriga. A Toolkit for Addressing HCI Issues in Visual Language Environments.*In IEEE Symp. on Vis. Lang. and Human-Centric Comp.*, 2005.
[10] W.-S Han et al. iGraph: A Framework for Comparisons of Disk Based Graph Indexing Techniques. *In VLDB*, 2010.
[11] H. Shang, et al. Connected Substructure Similarity Search. *In SIGMOD*, 2010.
[12] X. Yan, et al. Substructure Similarity Search in Graph Databases. *In SIGMOD*, 2005.
[13] X. Yan, J. Han. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.
[14] X. Yan, et al. Graph Indexing: A Frequent Structure-Based Approach. *In SIGMOD*, 2004.
[15] Z. Zeng, A. K. H. Tung, J. Wang, et al. Comparing Stars: On Approximating Graph Edit Distance. *In VLDB*, 2009.

[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.

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