VOGUE: Towards A Visual Interaction-aware Graph Query Processing Framework

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

[§]School of Computer Engineering, Nanyang Technological University, Singapore
[†]Department of Computer Science, Hong Kong Baptist University, China
[‡]School of Computer Science, Fudan University, China
assourav@ntu.edu.sg, choi@hkbu.edu.hk, sgzhou@fudan.edu.cn

ABSTRACT

Due to the complexity of graph query languages, the need for visual query interfaces that can reduce the burden of query formulation is fundamental to the spreading of graph data management tools to wider community. We present a novel HCI (human-computer interaction)-aware graph query processing paradigm, where instead of processing a query graph after its construction, it *interleaves* visual query construction and processing to improve *system response time*. We present the architecture of a system called VOGUE that exploits GUI latency to *prune false results and prefetch candidate data graphs* by employing a novel *action-aware* indexing and query processing schemes. We discuss various non-traditional design challenges and innovative features of VOGUE and highlight its practicality in evaluating subgraph queries.

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 cheminformatics 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. Although data models such as XML come close to graph representations, they do not support graphs as the primary object. Recently, due to increasing growth of graph-structured data in many domains, it is imperative to devise efficient techniques for analysis and querying large graph databases.

Querying any kind of database (*e.g.,* relational, XML) typically involves two key steps: query formulation using a query language (*e.g.,* SQL for relational databases) and efficient processing of the formulated query using a set of data structures and algorithms. In the context of graph database, a number of query languages (*e.g.,* GraphQL [7], SPARQL, and PQL [12]) have recently been proposed to address the first step. While most of these languages can express a wide variety of graph queries, the complexity of the syntax of a graph query language makes it unsuitable for ordinary users in a variety of domains. To address the second step, recently several innovative solutions have been designed to process a variety of graph queries [5, 8, 16, 18, 20, 21, 23–25]. Designing efficient strategy for graph query evaluation is a challenging problem due to its inherent computational hardness.

The traditional approach to address the challenge of query formulation is to build an intuitive and user-friendly visual framework on top of a state-of-the-art graph query processing technique. Such graphical interface is designed to reduce the burden of learning a query language by enabling visual query formulation using click-and-drag approach. Figure 1 depicts an example of such a visual interface for formulating graph queries. In this traditional paradigm the query processing module remains idle during query formulation and is only initiated after the Run icon is clicked. *That is, the query formulation and query processing activities are independent to one another.* Observe that although the final query that a user intends to pose is revealed gradually in a step-by-step manner during visual query construction, it is not exploited by the query processor prior to clicking of the Run icon to execute the query. In this paper, we challenge this traditional graph query processing paradigm in a visual querying environment by seeking answers to the following fundamental questions:

- Why wait for the complete visual query to be constructed before initiating query evaluation? Why cannot we start query evaluation immediately during query formulation?
- If we can then how can we interleave (blend) these two orthogonal steps together?

Specifically, we take the first major step to explore, design, and implement a visual graph querying framework called VOGUE (**V**isual Interacti**O**n-aware **G**raph **QUE**rying) that blends the two traditionally orthogonal steps (query formulation and query processing) to provide answers to the above questions.

The key benefits of the aforementioned paradigm are at least three-fold. First, it ensures that the query processor does not remain idle during visual query formulation. Second, it provides us an opportunity to significantly improve the *system response time* (SRT), which is the duration between the time a user presses the Run icon to the time when the user gets the query results. In traditional graph processing paradigm, SRT is identical to the time taken to evaluate the entire query. In contrast, in the new paradigm since we initiate query processing during query construction, SRT is the time taken to process a part of the query that is yet to be evaluated (if any). Third, since the GUI latency is exploited to *prune false results and prefetch candidate data graphs* during query construction, this paradigm provides us opportunities to enhance usability of graph databases by providing relevant *guidance* and *feedback* during query formulation. For instance, whenever a newly constructed edge makes a graph query fragment *unsatisfiable* (yield

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.
.

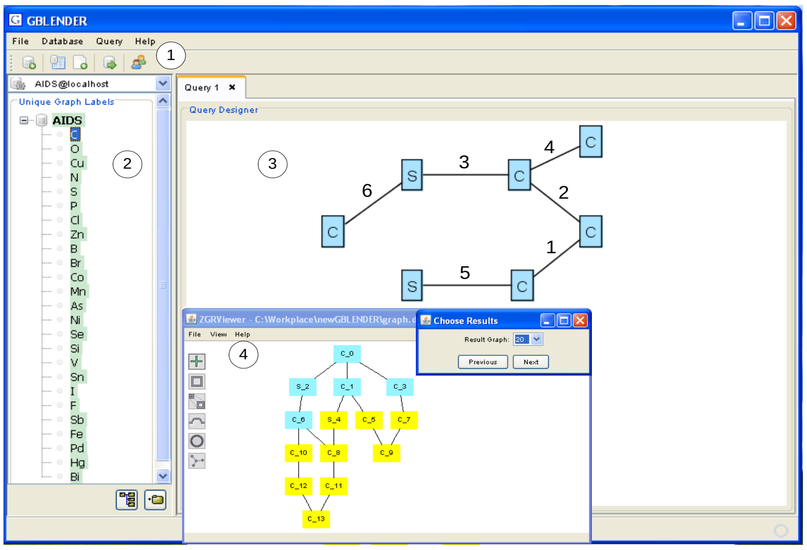


Figure 1: Visual interface for formulating graph queries (labels on the edges represent order of formulation of the edges).

empty answers), it can be immediately detected by processing the prefetched results and notified to the user in a timely fashion. It is not efficient if the unsatisfiability of the query is only detected at the *end* of query formulation as a user may have wasted her time and effort in formulating additional constraints. Note that such immediate feedback opportunity is lost in traditional paradigm where any kind of processing of a query only occurs *after* it has been completely formulated. Similarly, the partial results can be leveraged to provide suggestions, guidance, and recommendation for formulating correct queries.

The rest of the paper is organized as follows. In Section 2, we identify the non-traditional design issues that need to be considered to realize the proposed paradigm. We present the functional architecture of VOGUE in Section 3. We walk through a real-world application of VOGUE in Section 4. Section 5 summarizes the current implementation of the VOGUE architecture. We review related research in Section 6. The last section concludes the paper by highlighting interesting research directions.

2. NON-TRADITIONAL DESIGN ISSUES

We begin by identifying key characteristics associated with visual graph query formulation. Next, we highlight unique design issues that need to be considered in realizing this novel paradigm.

2.1 Visual Query Formulation

First, a visual query construction may follow either a *node/edge-at-a-time* or a *subgraph-at-a-time* approach. In the former case, a visual interface may support only incremental addition of a new node or edge for query formulation. Consequently, after every step the size of the query fragment grows by one. Note that it may be time consuming to formulate a query with large number of edges using this approach. In order to alleviate this problem, a sophisticated visual interface may subscribe to the *subgraph-at-a-time* approach in which a user may readily use a set of *canned subgraph patterns* (*e.g.,* benzene, chlorobenzene patterns) provided by the interface to formulate a visual query. For instance, instead of drawing six edges incrementally to construct a benzene ring in a query, we can construct it with a single click-and-drag if it is available as a canned pattern. Consequently, after every step the size of the query fragment grows by $k \geq 1$, which is the size of the canned pattern added to the query fragment.

Second, any practical visual graph querying system should allow users to modify a query fragment at any time during query formulation. 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 VOGUE; (b) she may commit a mistake or may change her mind during query formulation and modify the query fragment accordingly. Hence, the size of the query graph may not always increase monotonically with time.

Third, a visual graph query can be formulated in different ways by following different sequences of GUI actions. Figure 2 shows two different sequences of visual actions or *steps* (denoted by *Sequence 1* and *Sequence 2*) users may undertake to formulate the query in Figure 1. However, it is impossible to speculate *apriori* which query graph a user intends to construct following which particular sequence.

Fourth, the structure of the query fragment can evolve from a path to a tree or graph. At any step, the partial query graph formulated thus far, is either a frequent or infrequent fragment. Typically, as more edges are added, the chance of a query to remain frequent diminishes. Once it becomes infrequent, it remains as infrequent for the rest of the formulation steps unless previously constructed query fragment is modified. Note that the specific step at which a query fragment becomes infrequent is dependent on the query formulation sequence followed by a user. For instance, in Figure 2 the partial query evolved from a frequent fragment to an infrequent one after Step 4 in *Sequence 1* whereas it becomes infrequent after the second step in *Sequence 2*.

2.2 Action-Aware Indexing Schemes

In our proposed paradigm of blending visual query formulation and query processing, it is important to filter negative results after every visual action taken by a user. Consequently, we need an efficient indexing scheme which can exploit the above visual interaction characteristics effectively to prune false results. We envisage that such an *action-aware* indexing scheme should support the following key features:

- It should be able to prune a part of irrelevant results even if only *partial* query graph is known during query formulation.

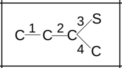
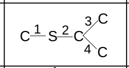
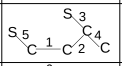
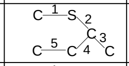
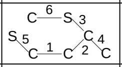
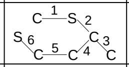
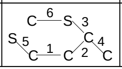
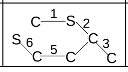
Steps	Sequence 1			Sequence 2		
	Action	Current Graph	Status	Action	Current Graph	Status
Step 1	C—C	$C \overset{1}{-} C$	frequent	C—S	$C \overset{1}{-} S$	frequent
Step 2	C—C	$C \overset{1}{-} C \overset{2}{-} C$	frequent	S—C	$C \overset{1}{-} S \overset{2}{-} C$	Infreq
Step 3	C—S	$C \overset{1}{-} C \overset{2}{-} C \overset{3}{-} S$	frequent	C—C	$C \overset{1}{-} S \overset{2}{-} C \overset{3}{-} C$	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 2: Query formulation steps.

- Since the size of a partial query graph g' grows by k, given a list of graphs that satisfy the fragment g' in *Step i*, it is important to support efficient strategy for identifying the graphs that match (exact or approximate) the fragment g'' (generated at *Step* $i + 1$) where $g' \subset g''$ and $|g''| = |g'| + k$.
- A partial query graph may evolve from being a frequent fragment to an infrequent one in the database. Furthermore, it may also evolve from a simple path to a complex graph structure. Hence, the proposed strategy should be able to support pruning based on both graph-structured frequent and infrequent fragments.
- It should be able to support modifications to a query graph efficiently. Note that such modification may result in reduction of query graph size as well as transformation of an infrequent query fragment to a frequent fragment again.
- Since smaller fragments always appear more often in different visual queries compared to larger-sized fragments, smaller-sized graph fragments should be efficiently indexed to support fast retrieval.
- Lastly, since subgraph isomorphism testing is known to be NP-complete, the indexing scheme should minimize expensive candidate verification while retrieving partial results.

While state-of-the-art indexing strategies are certainly innovative and powerful, we found out that they cannot be directly adopted for efficiently blending visual query formulation and processing for the following reasons.

- Firstly, these schemes are based on the conventional paradigm that the *entire* query graph must be available *before* query processing. However, in our proposed paradigm query processing is initiated as soon as a fragment of the query graph is visually formulated. For instance, *gIndex* [23] uses *apriori*-like strategy to enumerate a set of fragments of the query by checking whether a fragment belongs to the underlying *frequent subgraph index*. In order to generate this fragment set, the entire query graph should be available.

- Secondly, a key feature of action-aware indexing scheme is that it should be able to exploit both frequent and infrequent subgraph fragments to prune false results. However, very few existing techniques support both types of fragments. For example, *FG-Index* [5] uses frequent subgraphs as index features. Frequent graph queries are answered without verification and infrequent queries require only a small number of verifications. While it supports infrequent edges, it does not support infrequent graphs.

- Lastly, since existing indexes are designed for conventional subgraph matching paradigm, they do not require to support efficient traversal and retrieval of graph fragments g and g' where $|g| = |g'| + k$. Further, relatively more efficient pruning of smaller-sized frequent fragments compared to larger-sized fragments and query modification-friendly indexing support are also not important requirements for existing approaches.

2.3 Materialization of Intermediate Results

To realize the aforementioned paradigm, we need a visual graph query processor that supports evaluation of each formulated query fragment immediately after its construction as well as materialization of information related to all partial candidate graphs matching the query fragment. While this has always been considered as an unreasonable assumption in traditional databases, materialization of all intermediate results of a query is often the normal operating procedure in non-database systems [2, 14]. More recently, this strategy has also been supported in databases to enhance database usability [3] and query performance [26]. However, this issue is yet to be explored in the context of graph databases. Note that it is particularly challenging due to computational hardness of subgraph isomorphism test. Hence judicious strategy to minimize candidate verification while retrieving partial candidates is required.

2.4 Selectivity-free Query Processing

Selectivity-based query processing, that exploits estimation of predicate selectivities to optimize query processing, has been a longstanding approach in classical databases. Unfortunately, this strategy is ineffective in our proposed paradigm as users can formulate low and high selective fragments in any arbitrary sequence of actions. As query processing is interleaved with the construction (modification) of each fragment, it is also not possible to "push-down" highly selective fragments which requires knowledge of the *entire* query. The only possible way to bypass this stumbling block in this environment is to ensure that the sequence of visual actions formulated by a user is ordered by their selectivities. For example, consider the two query formulation sequences in Figure 2. Note that `c-s` has higher selectivity than `c-c`. Hence, it is beneficial if the former is formulated and processed before the latter (*Sequence 2* is a better choice than *Sequence 1*). However, users cannot be expected to be aware of such knowledge and it is unrealistic to expect them to formulate a query in a "selectivity-aware" order.

Additionally, due to the unavailability of the entire query graph during query processing, the classical approach of physical query plan generation is ineffective here as well. That is, the longstanding and highly successful strategy of parsing the query to generate an optimized logical plan first and then transform it to a physical query plan which can be executed by the VOGUE system cannot be leveraged in this novel paradigm.

2.5 Focus on Waiting Time of Users

Recall from Section 1, a key objective of this new paradigm is to improve the system response time (SRT). SRT is significant from an end user's perspective as it is the time she has to wait to view the results of her query. A longer SRT will add up to her frustration in using a graph querying system. On the other hand, typically she is not very concerned of the backend processing cost during query formulation as it does not effect her interaction with the system.

In order to ensure that the SRT is significantly reduced in VOGUE in comparison to traditional graph querying paradigm, several challenges has to be addressed. Firstly, the naïve strategy of matching every fragment a user draws on the query canvas to the underlying database can be prohibitively expensive due to multiple subgraph isomorphism tests and repeated access to the disk. Hence, efficient index-based strategy is required that can minimize disk access as well as subgraph isomorphism tests. Ideally, evaluation of each new query fragment should be finished during the latency offered by the GUI while constructing the succeeding fragment. Secondly, SRT should be robust to different query formulation sequences of a query. As different users may follow different order of visual steps to formulate a query, the SRT of the query should not vary significantly for different sequences.

3. ARCHITECTURE OF VOGUE

Figure 3 shows the system architecture of VOGUE designed for single-user environment. The VOGUE GUI, the *Query Wizard*, and the *Query Feedback* components of the *Visual Interface Manager* provide an interactive visual environment to enable users to formulate queries without the knowledge of complex graph query languages. The *Query Wizard* and the *Query Feedback* modules are responsible for intelligent guidance and feedback to users to further ease the cognitive overhead associated with query formulation. The *Feature Extraction* module mines the frequent fragments from the underlying graph database using an existing frequent graph mining technique. The *Action-Aware Index Constructor* module takes as input the discovered frequent and infrequent fragments and builds an array of *action-aware indexes* to support efficient query processing in our proposed paradigm. The *Action-Aware Query Processing* module embodies a series of innovative index-based algorithms that utilize the latency offered by the GUI actions to retrieve partial results of a graph query. These results are progressively refined based on the subsequent actions by the user. Lastly, upon successful execution of a graph query, the *Results Visualizer* module displays the results in graphical format. We now elaborate on these components.

3.1 Visual Interface Manager

Figure 1 depicts the screenshot of the current visual interface of VOGUE supporting edge-at-a-time query formulation approach. 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. Panel 3 depicts the area for formulating graph queries. A user 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. Then, she 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. Finally, the user can execute the query by clicking on the Run icon in Panel 1. Panel 4 displays the query results.

The above query construction activities are supported by a *feedback* and *guidance mechanism* (*Query Feedback* and *Query Wizard* modules) that guide users to formulate queries by (a) helping them to find nodes in Panel 2 quickly that may be of interest to them and (b) providing appropriate feedback whenever necessary in a timely manner. In order to realize the former, several features of the dataset such as frequency of a node's label, degree of connectivity of a node with other nodes, and popularity of a label in past

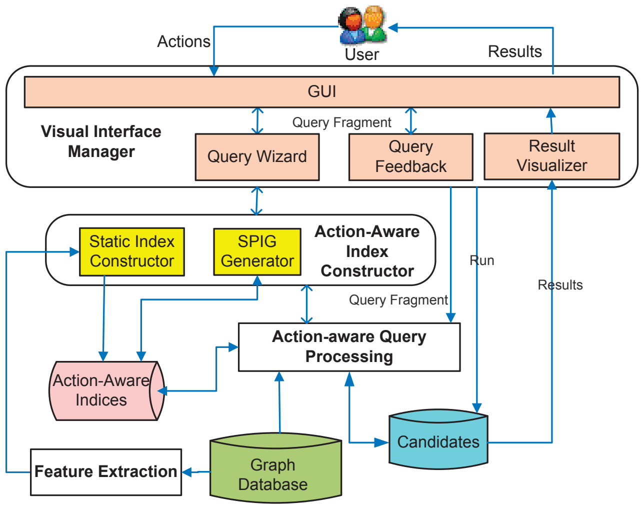


Figure 3: Architecture of VOGUE.

queries from users, are determined by analyzing the structure of the dataset and query log. On the other hand, query feedbacks are essential during query construction as a user may not know if the query she is trying to formulate will return any results. We observe user's actions during query formulation and notify her if a query fragment constructed at a particular step fails to return any results using our *action-aware indexing schemes* (discussed below). Further, it also advises the user on which edge in the formulated query fragment is the "best" to remove in order to get non-empty result set. These feedbacks are provided in a timely fashion in order to minimize the cognitive overhead associated with tasks interruption. Lastly, given a partial query fragment already drawn by a user, the query guidance mechanism aims to provide suggestion of a concise set of edges that she is likely to draw in the next step.

Upon successful execution of a subgraph query, the *Result Visualizer* module displays the results in graphical format (Panel 4). The result graphs are ordered according to increasing value of their *similarity distance*.

3.2 Feature Extraction Module

This module mines the *frequent fragments* from the graph database $\mathcal{D}$ using an existing frequent graph mining technique (the current version uses *gSpan* [22]). Informally, we use the term *fragment* (resp. *query fragment*) to refer to a small subgraph existing in graph databases (resp. query graphs). Given a fragment g which is a subgraph of G (denoted as $g \subseteq G$) and $G \in \mathcal{D}$, we refer to G as the *fragment support graph* (FSG) of g. Since each data graph in $\mathcal{D}$ is denoted by an unique identifier, $fsgIds(g)$ denotes the set of identifiers of FSGs of g. A fragment g is *frequent* in $\mathcal{D}$ if its support is no less than $\alpha|\mathcal{D}|$ where $0 < \alpha < 1$ is the *minimum support threshold*. Otherwise, g is an *infrequent* fragment.

3.3 Action-Aware Index Constructor Module

This component of VOGUE is responsible for constructing two types of indexes, namely *action-aware static* and *action-aware dynamic* indexes [10, 11]. These indexes are constructed based on the assumption that the current version of VOGUE supports edge-at-a-time query formulation strategy.

Action-aware static index. We support two types action-aware static index which are build on $\mathcal{D}$. The *action-aware frequent* *index* (A^{2}F) is a graph-structured index having a *memory-resident* and a *disk-resident* components. We refer to them as *memory-based frequent index* (MF-index) and *disk-based frequent index* (DF-index), respectively. Specifically, small-sized frequent fragments (frequently utilized) are stored in MF-index whereas larger frequent fragments (less frequently utilized) reside in DF-index.

The DF-index is an array of *fragment clusters*. A *fragment cluster* is a directed graph $C = (V_C, E_C)$ where each vertex[1] $v \in V_C$ 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 $|f| = |f'| + 1$. We denote the root vertex (node with no incoming edge) of C as $root(C)$. Each fragment f of v is represented by its CAM code [9], denoted as $cam(g)$. Each vertex with fragment f in C points to a set of FSG identifiers of f ($fsgIds(f)$). Note that given the frequent fragments f and f', if $f' \subset f$ then $fsgIds(f) \cap fsgIds(f') = fsgIds(f)$. That is, vertex v' (representing f') and its child vertex v (representing f) share a large number of FSGS. Consequently, VOGUE store only a subset of $fsgIds(f)$ at each vertex.

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 vertices and edges have same semantics as C. In addition, vertices representing frequent fragments of size β are leaf vertices in G_M and do not have any child fragments. 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 C_j in the DF-index such that $f \subset root(C_j)$.

The *action-aware infrequent index* (A^{2}I-index) indexes infrequent fragments to prune the candidate space for infrequent queries. In order to ensure that the index is space-efficient, we index only the *discriminative* infrequent fragments (DIFS). Informally, a DIF is a smallest infrequent subgraph of an infrequent fragment. Given an infrequent fragment g, let $sub(g)$ be the set of all subgraphs of g. If $sub(g)$ contains only frequent fragments or $|g| = 1$, then g is a DIF in $\mathcal{D}$. Intuitively, A^{2}I-index consists of an array of DIFS arranged in ascending order of their sizes. Each entry in the index stores the CAM code of a DIF g and $fsgIds(g)$.

Action-aware dynamic index. The SPIG *Generator* module generates a dynamic index on-the-fly during visual query construction. For each *new* edge e_ℓ created by the user, this module create a ***spi**ndle-shaped **g**raph* (SPIG) using the action-aware indexes. Each edge is assigned 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 its *label*. The edge with the *largest* ℓ is referred to as *new edge* (most recently added). Note that a set of SPIGS are created for a query graph.

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. 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, then the identifier of the vertex or entry representing g in the corresponding index is stored in *frequent id* or DIF *id* attribute, respectively. Note that 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 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.

[1]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.

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.

3.4 Action-Aware Query Processing Module

This module implements an innovative SPIG-based query processing algorithm that utilizes the latency offered by the GUI actions to retrieve partial candidate data graphs. The current version of VOGUE has the following key operators to realize the proposed paradigm.

`Select`. This operator takes as input a new edge e_ℓ (graph fragment) drawn by a user on the query canvas and retrieves identifiers of data graphs containing the query fragment q (denoted by R_q).

`ExactMatch`. This operator implements an action-aware index-based subgraph containment search for retrieving candidate graphs for q. That is, it takes q as input and generates a set of candidate data graph identifiers R_q as output. If q is a frequent fragment, then it retrieves FSG identifiers of corresponding g by probing A^{2}F-index. Otherwise, if g represents a DIF, then it retrieves the FSG identifiers from A^{2}I-index. If g is neither a DIF nor a frequent fragment then for each identifier in the frequent subgraph id set and DIF subgraph id set of g in the SPIG, 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.

`SimilarMatch`. This operator implements a SPIG-based subgraph similarity search[2] for retrieving approximate matches of q. Given the *subgraph distance threshold* σ, it exploits the SPIG set 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 the SPIG set. The candidate set are separated into two parts, namely R_{free} and R_{ver}, storing the identifiers of verification-free candidate graphs and data graphs that need verification, respectively. For each vertex in the *i-th* level, if it is a frequent fragment or DIF, then the candidates satisfying the node is retrieved using the aforementioned exact substructure search procedure and combine them with existing R_{free}. Otherwise, it is neither a frequent fragment nor a DIF. Consequently, the candidate data graphs are once again computed using frequent subgraph id set and DIF subgraph id set (see above) and combined with existing R_{ver}. Lastly, candidates that exist in both R_{free} and R_{ver} are removed from R_{ver}. Note that `ExactMatch` and `SimilarMatch` are typically used inside another operator, such as `Select` or `Update`.

`Update`. This operator handles modification to a query in VOGUE by enabling edge deletion[3]. It is invoked under following scenario: (a) the user selects deletion of an existing edge (the modified query

[2]Similar to [16], the current version of this operator adopts the *maximum connected common subgraphs* (MCCS) for computing similarity between a pair of graphs.

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

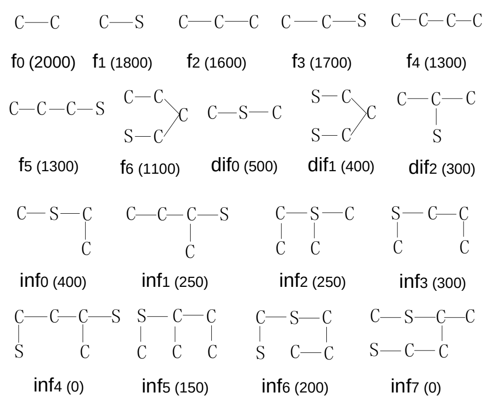


Figure 4: Frequent and infrequent fragments.

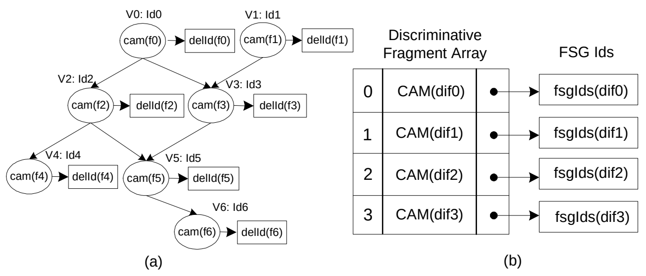


Figure 5: Examples of MF-index and A^{2}I-index.

graph must be connected graph at all times); (b) the system recommends deletion of an edge whose removal would maximize the size of the candidate graph set of the modified query fragment.

Let e_m be the newest edge in q and e_d be the edge deleted from q by the user at any time during query formulation where $0 < d \leq m$. The new query fragment q' is formed by deleting e_d from q. The SPIG set is updated by removing SPIGs and vertexes related to e_d. Finally, the new candidate set is generated by invoking the `ExactMatch` or `SimilarMatch` operator.

`Verify`. This operator returns the exact results by filtering the false candidate data graphs. The current implementation uses Ullman's algorithm for subgraph isomorphism test and extends VF2 [6] to handle MCCS-based similarity verification.

Query processing strategy. When a user draws a new edge e_ℓ on the query canvas, the `Select` operator is invoked to retrieve identifiers of data graphs containing the query fragment q (denoted by R_q) and monitors its status. If R_q is non-empty at a specific step then the `ExactMatch` operator is invoked as q has exact matches in the database. If R_q becomes empty (*e.g.,* in Figure 2 (Sequence 1), the query fragment after Step 5 does not have any match) then it efficiently support the following two steps. (a) If the user chooses to modify q then it invokes the `Update` operator to handle the modification process. (b) Otherwise, it uses the `SimilarMatch` operator to retrieve approximate matches to q.

If the final query is a frequent subgraph containment query or a DIF, then the results are directly computed without subgraph isomorphism test (without invoking the `Verify` operator). If it is a non-DIF infrequent subgraph containment query, when the `Run` icon is clicked, the `Verify` operator returns the exact results by filtering the false candidates. Otherwise, if the final query has evolved to a subgraph similarity query then firstly the candidates in R_{free} are added to result set without any verification test. Next, it generates the result set from the candidates in R_{ver} by invoking the `Verify` operator. Observe that our query processing strategy does not rely on selectivities of the added edges or the order in which they are constructed.

4. AN APPLICATION SCENARIO

In this section, we illustrate with an example how VOGUE can be deployed on a real-world dataset to support our proposed visual querying paradigm.

Consider a graph-structured chemical compounds dataset (*e.g.,* AIDS Antiviral dataset, *DrugBank* [19]). VOGUE first mines and extracts the frequent fragments and DIFS from this dataset (*Feature Extraction* module). Figure 4 depicts some of these frequent and infrequent fragments (support values are shown in parenthesis). Specifically, the fragments $f_0 - f_6$ are frequent fragments and the rest are infrequent fragments. Among the infrequent fragments, $dif_0 - dif_2$ are DIFS as all subgraphs in each fragment are frequent. These fragments are then used to construct the A^{2}F (MF-index and DF-index) and A^{2}I-indexes (*Action-Aware Index Constructor* module) to support efficient matching of frequent and infrequent query fragments, respectively, while formulating a visual query. For instance, Figure 5(a) depicts an examples of MF-index based on the frequent fragments in Figure 4 (for $\beta = 4$). 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)). Figure 5(b) depicts an A^{2}I-index based on the DIFS. The identifier of each DIF g in the index is denoted by $a2iId(g)$ (*e.g.,* $a2iId(dif_1) = 1$ in Figure 5(b)).

We now illustrate how query processing is blended with visual query formulation in VOGUE. Suppose a user formulates the visual query in Figure 1 by following the formulation sequence *Sequence 1* in Figure 2. Let $\sigma = 2$. Observe that the query remains as frequent in the first three steps. After Step 4, it evolves to an infrequent query. For each new step, the SPIG *Generator* module of VOGUE dynamically generates a SPIG using the action-aware indexes. Then, using these indexes the *Action-Aware Query Processing* module computes candidate data graphs matching (exact or approximate) the partial query fragment formulated so far. We first give an example of the SPIG generation process.

Consider Step 5. 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 frequent id, DIF id, frequent subgraph id set, and DIF subgraph id set, 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 frequent ids are 1, 3, 5, and 6, respectively. Since $v_{5,5}$ represents dif_1, the DIF id is 1 (Figure 5(b)). As $v_{5,6}$ represents the non-DIF infrequent pattern inf_4, it is neither indexed by A^{2}F-index nor by A^{2}I-index. Consequently, frequent and DIF ids of $v_{5,6}$ are empty. 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, frequent subgraph id set contains only 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

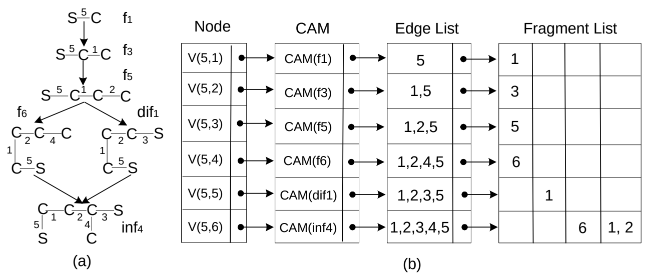


Figure 6: The vertices of the spindle-shaped graph in step 5.

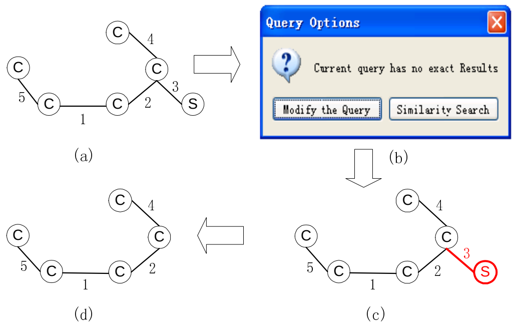


Figure 8: The query modification procedure in Step 5.

ids 1 and 2 in Figure 5(b)). Consequently, DIF subgraph id set of $v_{5,6}$ contains 1 and 2. Figure 7 depicts the set of SPIGS constructed for Steps 1 to 6. Observe that the fragments represented by vertices of two consecutive SPIGS (*e.g.,* S_5 and S_6) can be quite different.

We now illustrate how candidate data graphs are generated at each step. In the first step (*Sequence 1*), edge e_1 is added and S_1 is constructed as shown in Figure 7(a). As frequent id of $v_{1,1}$ is 0, the `ExactMatch` operator is used to locate it in A^{2}F-index and retrieve $fsgIds(f_0)$ as the candidate set of current query fragment. In the second step, the SPIG S_2 is constructed. Since frequent id of $v_{2,2}$ ($v_{2,2}$ is the target vertex) is 2, its FSG identifiers are again retrieved by probing A^{2}F-index. After Step 3, $v_{3,3}$ is the target vertex in S_3 and frequent id of $v_{3,3}$ is 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 non-DIF infrequent fragment and frequent subgraph id set and DIF subgraph id set are $\{4,5\}$ and $\{2\}$, respectively. Hence, the FSG identifiers of these fragments are computed as follows: $fsgIds(v_{4,6})$=$fsgIds(dif_2) \cap fsgIds(f_4) \cap fsgIds(f_5) = fsgIds(inf_1)$. 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. Suppose the user chose the latter option. Then, the `SimilarMatch` operator is invoked. Recall that $\sigma = 2$. That is, in Step 5 two edges are allowed to be missed in the results of substructure similarity search. Consequently, $R_{free}(3)$ and $R_{ver}(3)$ are generated for the vertexes in the third levels of the SPIGS ($v_{3,3}$, $v_{4,4}$, $v_{4,5}$ 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 ($v_{5,4}$, $v_{5,5}$, and $v_{4,6}$). Consequently, $R_{free} = R_{free}(4) = fsgIds(v_{5,4}) \cup fsgIds(v_{5,5}) = fsgIds(f_6) \cup fsgIds(dif_1)$ whereas $R_{ver} = R_{ver}(4) = fsgIds(v_{4,6})$. 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 $|fsgIds(v_{6,6})| = 150$, $|R_{ver}(5)| \leq 350$.

Lastly, in Step 7, when the user clicks on the `Run` icon, the list of data graphs that match the query approximately are returned. In this step, the verification-free matches (R_{free} in the above step) are returned directly. On the other hand, candidate data graphs that need verification (R_{ver}) are first verified using the `Verify` operator and then the relevant matches are returned to the user.

Visual query modification. We now illustrate with an example how query modification is handled in VOGUE by the `Update` operator. For clarity, we illustrate only single edge deletion. It is trivial to extend it to support multiple edge deletions. Reconsider Step 5 of *Sequence 1* in Figure 2. The state of the query fragment is depicted in Figure 8(a). Assume that the user now selects the query modification option (Figure 8(b)). Since $|fsgIds(v_{5,4})| = |fsgIds(f_6)| = 1300$ 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 8(c)). Figure 8(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 SPIG set is shown in Figure 9.

Now suppose the user chooses to invoke substructure similarity search instead at Step 5 (as discussed above) and then deletes edge 6 after Step 6 (the case where deletion of an edge is selected by the user). Now q' matches $v_{5,6}$ and the target vertex of S_5. Hence, the updated SPIG set now excludes S_6. At last, the new candidates are calculated based on this updated SPIG set.

5. IMPLEMENTATION OF VOGUE

VOGUE is implemented in Java JDK 1.6. The current version of VOGUE supports formulation of subgraph containment and subgraph similarity search queries on a large set of small or medium-sized graphs (*e.g.,* chemical compounds). The *Visual Interface Manager* currently supports edge-at-a-time query construction. The *Query Wizard* and *Query Feedback* modules are yet to be implemented. The *Result Visualizer* module is implemented using *ZGRViewer* [15]. The *Feature Extraction*, *Action-Aware Index Constructor*, and *Action-Aware Query Processing* modules have all been implemented to support efficient blending of aforementioned queries. The reader may refer to [10, 11] for detailed algorithms associated with realizing these modules.

Performance summary of VOGUE. Our empirical study in [10, 11] highlights the benefits of the proposed paradigm. The SRTS of subgraph containment and subgraph similarity search queries are significantly lower than state-of-the-art approaches based on traditional paradigm [16, 21]. More importantly, the SRT of VOGUE grows gracefully with the size of underlying database. Also, the construction process of action-aware indexes is very efficient. In particular,

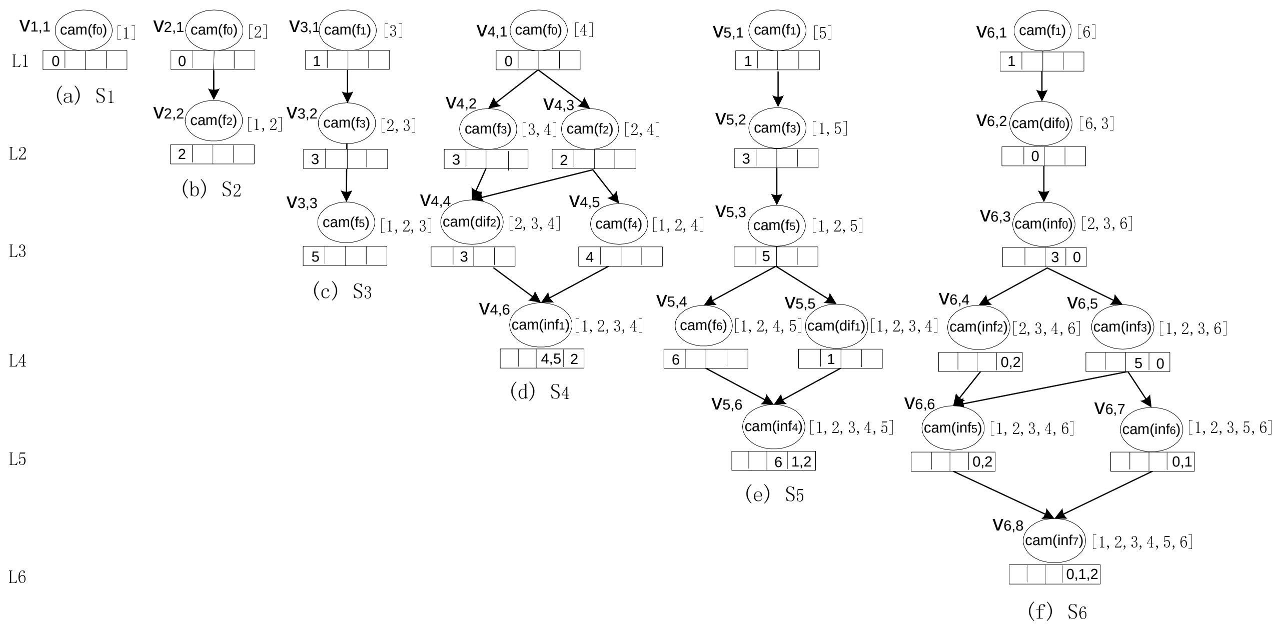


Figure 7: The SPIG set for Sequence 1 (Edge Lists are in square brackets and Fragment Lists are shown in rectangular boxes.)

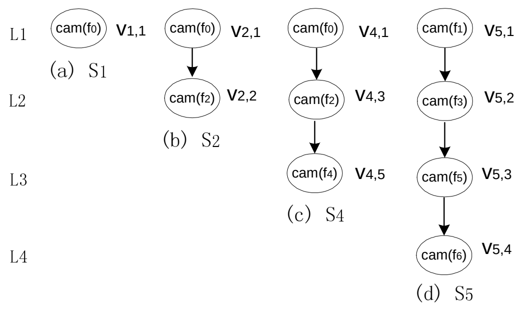


Figure 9: The updated SPIG set after deleting edge 3 in Step 5.

the generation of action-aware dynamic index (SPIG) at each step 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). Furthermore, the formulation sequences of a query only have minor effect on the SPIG construction time and SRT highlighting the robustness of VOGUE. Lastly, the query modification cost is cognitively negligible and can easily be completed by exploiting the GUI latency.

6. RELATED WORK

Recent efforts to address subgraph containment and similarity search problems can be broadly classified into two streams. One stream focuses on processing subgraph queries on a large number of small or medium-sized graphs such as chemical compounds [5,8, 16,21,23,25]. The other stream aims to handle query processing on a small number of large graphs (*e.g.,* protein interaction networks, social networks) [18, 20, 24]. Nevertheless, all these aforementioned 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. In contrast, VOGUE realizes a novel query processing paradigm by blending two traditionally independent areas, namely human-computer interaction and database query processing. Specifically in the proposed paradigm, when a graph query is visually formulated, its evaluation is interleaved with the formulation activities. Hence, our method is orthogonal to existing studies related to graph query processing. Additionally, in order to speed up subgraph evaluation, most existing works focus on developing indexing techniques to support efficient searching. As remarked earlier, the indexing scheme necessary to support the proposed paradigm is different from these traditional approaches. In particular, unlike existing strategies, the design of the proposed indexing scheme is influenced by the characteristics of users' visual interaction behaviors during query formulation.

There has been some research in the arena of visual query languages for graph databases [1,4]. *QGraph* [1] is a visual language for querying and updating graph databases. In *QGraph* the user can draw a query consisting of some nodes and edges with specified relations between their attributes. The output of the system is a collection of all subgraphs in the underlying database that have the desired pattern. *Graphite* [4] allows the user to visually construct a graph query pattern over large attributed graphs, finds both its exact and approximate matching subgraphs, and visualizes the matches. VOGUE differs from these efforts in the following way. Although the main objective of these approaches being easy to use; all these systems follow the traditional paradigm of query processing where evaluation of a visual query pattern is initiated only *after* the complete query has been formulated.

7. DISCUSSIONS

In this paper, we have presented VOGUE - a novel approach for processing graph queries which are formulated using a visual interface. To the best of our knowledge, it is the first system that makes a strong connection between graph query processing and visual query formulation. VOGUE employs a novel indexing scheme, which exploits some of the users' interaction characteristics with visual interfaces to support efficient pruning and retrieval. The innovative subgraph querying engine exploits the latency offered by the GUI-based query formulation to prune false results and prefetch partial query results.

We have barely scratched the surface of this novel query processing paradigm. We are currently exploring following interesting issues (non-exhaustive) to enhance the framework of VOGUE.

***Enhancing usability of* VOGUE GUI.** We are currently enhancing the usability of VOGUE GUI by incorporating query feedback and guidance mechanisms in the *Visual Interface Manager*. Specifically, for the first time, we aim to integrate well-founded principles from HCI and cognitive psychology with graph query processing to devise effective solutions towards this goal. Additionally, recall that the current version of VOGUE only supports edge-at-a-time query formulation. We are extending it to support subgraph-at-a-time query construction so that large subgraph queries can be easily constructed with relatively fewer clicks. Consequently, the underlying action-aware indexing schemes and query processing strategies need to be adapted to take advantage of such query construction strategy.

"Blendability" of complex graph queries. Currently, VOGUE successfully blends subgraph containment and subgraph similarity search queries. We aim to enhance the expressiveness of VOGUE by investigating if more complex subgraph queries are "blendable" and demonstrate superior performance. For instance, we intend to investigate blending of supergraph containment queries, homeomorphic graph queries, and *generalized* subgraph queries [13].

Visual query processing on massive graphs. VOGUE currently supports querying a large set of small or medium-sized graphs. A natural extension to this problem is to support similar queries on massive graphs. However, generating feature-based action-aware indexes is a challenging problem as it requires us to determine frequent fragments in a very large graph which is prohibitively expensive operation and a long standing problem [20]. Furthermore, it is also space-inefficient to index location of all the possible occurrences of a fragment in a massive network as it may appear numerous times. Lastly, visualizing query results becomes cognitively and computationally challenging. Even if a data graph contains few thousands of nodes and edges, it imposes significant cognitive burden on the end user if it is shown in its entirety. Particularly, the entire graph looks like a giant hairball and the subgraphs that match the query are lost in the visual maze. We are currently exploring a novel graph partitioning-based technique in the VOGUE framework to address these challenges.

Synthetic visual query simulator. Lastly, there is a pressing need for a framework to support comprehensive empirical study of VOGUE. In contrast to traditional paradigm, each query in VOGUE must be formulated by a set of real users for empirical study. Furthermore, each query can follow many different query formulation sequences. The challenge here is that it is prohibitively expensive to find and engage a large number of users who are willing to formulate a large number of visual queries. In fact, our experience suggests that such aspiration strongly deters end users to participate in the empirical study. To address this limitation, we are currently building a *synthetic visual query simulator* that simulates visual graph query formulation by real users. A key feature of this simulator is that it leverages principles from HCI on visual task completion to simulate users' interaction behaviors. It is then integrated with the VOGUE architecture to simulate our proposed paradigm of blending query formulation and query processing.

Acknowledgement: Shuigeng Zhou was supported by the Research Innovation Program of Shanghai Municipal Education Committee under grant No. 13ZZ003. We would also like to thank Changjiu Jin for implementing several features of VOGUE.

8. REFERENCES

[1] H. Blau , N. Immerman , D. Jensen. A Visual Language for Querying and Updating Graphs. *Technical Report 2002-037*, University of Massachusetts, Amherst, 2002.

[2] S. P. Callahan, J. Freire, et al. VisTrails: Visualization Meets Data Management. *SIGMOD*, 2006.

[3] A. Chapman, H. V. Jagadish. Why Not? *In SIGMOD*, 2009.

[4] D. H. Chau , C. Faloutsos, H. Tong, et al. GRAPHITE: A Visual Query System for Large Graphs. *ICDM Workshop* , 2008.

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

[6] 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.

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

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

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

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

[11] C. Jin, et al. PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. *In ICDE*, 2012.

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

[13] W. Lin, X. Xiao, et al. Efficient Algorithms for Generalized Subgraph Query Processing. *In CIKM*, 2012.

[14] T. Oinn, M. Greenwood, M. Addis et al. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences: Research Articees. *Concurr. Comput.: Pract. Exper.*, 18(10), 2006.

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

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

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

[18] Y. Tian, J. Patel. TALE: A Tool for Approximate Large Graph Matching. *In ICDE*, 2008.

[19] D. S. Wishart, C. Knox, A. C. Guo, et al. DrugBank: A Knowledgebase for Drugs, Drug Actions and Drug Targets. *Nucleic Acids Research*, Vol. 36, D901-D906, 2008.

[20] Y. Xie, P. S. Yu. CP-Index: On the Efficient Indexing of Large Graphs. *In CIKM*, 2011.

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

[22] X. YAN, J. HAN. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.

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

[24] S. Zhang, et al. SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs. *In VLDB*, 2010.

[25] P. Zhao, et al. Graph Indexing: Tree + delta ≥ Graph. *In VLDB*, 2007.

[26] Y. ZHOU, S. S. BHOWMICK, ET AL. XBLEND: Visual XML Query Formulation Meets Query Processing. *In ICDE*, 2009.

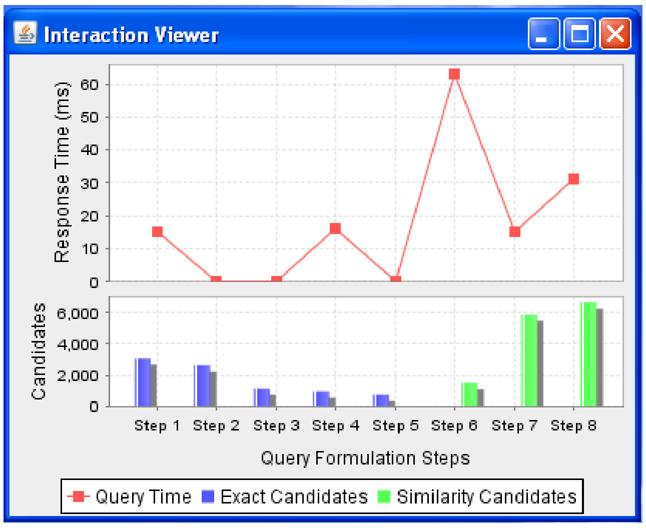


Figure 10: Demonstration viewer.

APPENDIX

Demonstration Objectives

VOGUE is implemented in Java JDK 1.6. Our demonstration will be loaded with synthetic datasets and a few real datasets (e.g., AIDS Antiviral dataset containing 43K graphs) with different sizes. Example query graphs will be presented. Users can also write their own ad-hoc queries through our GUI (Figure 1).

Interactive experience of the novel query evaluation paradigm. One of the key objectives of the demonstration is to enable the audience to interactively experience the proposed query processing paradigm in real-time. During the visual construction of a subgraph query, one will be able to view the generation of candidate data graphs at each visual step, evolution of a containment query to a similarity query (if necessary) and their effect on the size of candidate set (bottom part of Figure 10). Additionally, the user will be able to experience the time taken by VOGUE at each visual step for fetching candidate data graphs (top part of Figure 10) and appreciate the fact that the latency offered by the GUI at each step is sufficient to finish this prefetching task. Furthermore, she will be able to visualize in real-time the effect of the type of subgraph query fragment (containment or similarity) on the prefetching time.

Robustness to query modification. We shall interactively show the following two features of VOGUE to highlight its robustness to query modification. First, we shall show the automatic edge recommendation process for deletion (Figure 11). Second, we shall demonstrate in real-time how VOGUE efficiently handle query modification in response to deletion of any edge by a user during query formulation.

Superior performance of VOGUE. We shall demonstrate that the proposed paradigm significantly improves SRT compared to traditional graph query evaluation systems.

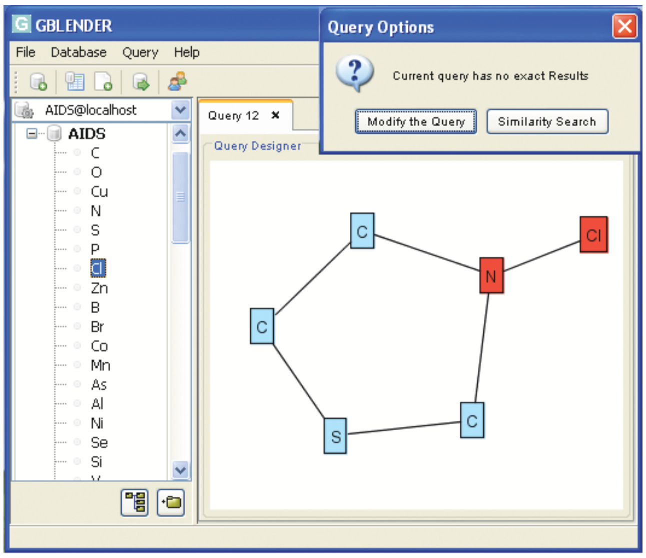


Figure 11: Query modification.