ViSual: An HCI-inspired Simulator for Blending Visual Subgraph Query Construction and Processing

Sourav S Bhowmick[§] Huey Eng Chua[§] Benji Thian[§] Byron Choi[†]

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

***Abstract*—In [3], we laid out the vision of a novel graph query processing paradigm, where visual subgraph query formulation is interleaved (or "blended") with query processing by exploiting the latency offered by the GUI. Our recent attempts at implementing this vision [6], [7] do not provide any robust framework to systematically investigate the performance of this novel paradigm. This is because it is prohibitively expensive to engage a large number of users to formulate a large number of visual queries in order to measure the performance of blending query formulation with query processing. In this demonstration, we present a novel *synthetic visual subgraph query simulator* called ViSual that can evaluate the performance of this paradigm for a large number of visual subgraph queries *without* requiring a large number of users to formulate them. Specifically, it leverages principles from HCI to quantify the GUI latency that is necessary to realistically simulate blending of query formulation and query processing.**

I. Introduction

Querying graph databases has emerged as an important research problem for many real-world applications that are based on graph-structured data. A number of graph query languages (*e.g.,* SPARQL) have been proposed that can be used to formulate graph queries. 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 (*e.g.,* cheminformatics). A popular approach to address this query formulation challenge is to build a user-friendly visual querying framework on top of a state-of-the-art graph query processing technique (*e.g.,* [9]).

In [3], [6], [7], we laid out the vision of a novel visual graph query processing paradigm where instead of processing a query graph after its construction, it *interleaves* the two traditionally orthogonal steps, namely visual query construction and processing, bringing in at least two key benefits. First, it significantly improves 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. Second, as a side effect of this paradigm, it is possible to enhance usability of a graph querying system by providing relevant *guidance* and *feedback* during query formulation. For instance, whenever a newly constructed edge makes a graph query fragment yield empty answers, it can be immediately detected by processing the prefetched results and notified to the user in a timely fashion. 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.

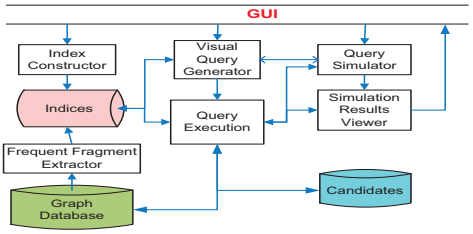

Fig. 1. Architecture of ViSual.

The aforementioned framework, however, lacks of adequate support to undertake comprehensive empirical study to evaluate the performance of this new paradigm. In contrast to traditional query processing paradigm, where the runtime performance of a large number of subgraph queries can be easily measured by automatically extracting a random collection of subgraphs from the underlying data and executing them, each visual query in [7] *must* be formulated by a set of real users. This is because in this paradigm the availability of the GUI latency at each formulation step is exploited to prefetch and refine candidate matches. Note that each query may follow many different *query formulation sequences* (QFS) and the GUI latency at each step may vary across different users. Consequently, it is important to realistically simulate each visual query construction step to investigate the performance of a query. Unfortunately, it is prohibitively expensive to engage a large number of users to formulate a large number of visual queries with different QFS. In fact, our experience suggests that such aspiration strongly deters users to participate in the empirical study.

In this demonstration, we advocate that a different and more palatable approach to address the aforementioned problem is to create a novel *synthetic visual subgraph query simulator* that realistically simulates visual subgraph query construction to support the paradigm of blending query formulation and query processing. Specifically, we present ViSual (**VI**sual **S**ubgraph **QU**ery Formul**A**tion Simu**L**ator), which is build on top of PRAGUE [7] and focuses on simulating subgraph query construction on a database containing a large number of small or medium-sized graphs. Using ViSual, one can automatically generate a large number of *test* subgraph queries having different *user-specified characteristics*, simulate their formulation based on different QFS without requiring real users,

and generate a *multi-level* interactive graphical view of their performances in [7]. A key feature of this simulator is that it leverages principles from HCI on visual task completion to *quantitatively* model the GUI latency that arises during visual query formulation. This model is leveraged to realistically simulate the visual query formulation task by estimating the time required to construct an edge in a query graph (*i.e.,* GUI latency) *without* requiring any real users.

II. System Overview

Figure 1 shows the system architecture of ViSual and mainly comprises of the following modules.

The GUI module: Figure 2 depicts the screenshot of the visual interface of ViSual. A user begins by choosing a target database using Panel 1 on which visual construction of subgraph queries and their processing will be simulated. Panel 2 displays the unique labels of nodes that appear in the dataset in lexicographic order. Note that during the query formulation process, these labels are chosen for creating the nodes in the query graph. Panel 3 depicts the area that seeks input to the *Visual Query Generator* module. Specifically, a user provides details of the number, size, and *types* of subgraph queries for simulation. Panel 4 displays the simulation of the proposed paradigm in real time when a user clicks on the `Simulate` button in Panel 5. Specifically, it *estimates* the GUI latency available at each formulation step and blends it with query processing. Note that a user may select a set of queries or a specific query for simulation. Panel 5 enables us to change various settings related to simulation (see discussion related to the *Query Simulator* module) as well as to view simulation results (see *Simulation Results Viewer* module) by clicking appropriate buttons.

The Frequent Fragment Extractor 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* [8]). 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.

The Index Constructor module: This module constructs two indexes to facilitate generation of synthetic visual subgraph queries. Note that these indexes are also used in [6], [7] for query processing. The *frequent index* is used to generate frequent query graphs and comprises of a *memory-based frequent index* (MF-index) and a *disk-based frequent index* (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 node $v \in V_C$ 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

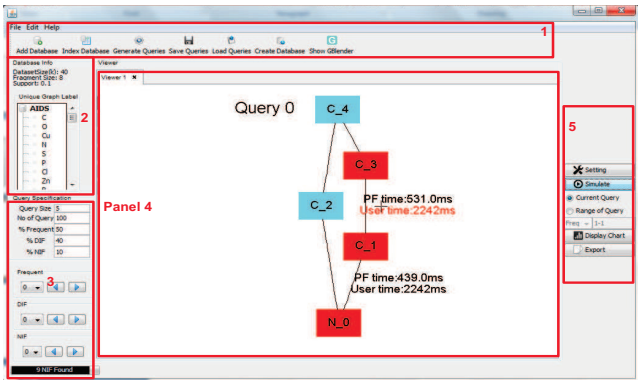

Fig. 2. [Best viewed in color] The ViSual GUI.

subgraph of f (denoted as $f' \subset f$) and $|f| = |f'| + 1$. Each fragment f of v is represented by its CAM code. Each node with fragment f in C points to $fsgIds(f)$. MF-index, on the other hand, indexes all frequent fragments having size less than or equal to β and has a similar structure as the DF-index.

The *infrequent index* indexes infrequent fragments to support generation of infrequent subgraph queries. We index only the *discriminative* infrequent fragments (DIFs), which are infrequent fragments whose subgraphs are all frequent [7]. Intuitively, it 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 $fsgIds(g)$. For distinction, we refer to an infrequent fragment that is not a DIF as a *non*-DIF *infrequent fragment* (NIF).

The Visual Query Generator module: Given the query size s, the total number of size-s query graphs that needs to be generated, and the distributions of query types (Panel 3), this module generates appropriate number of frequent and infrequent (both DIF and NIF) queries by leveraging the aforementioned indexes. Specifically, a frequent query is constructed as follows. If $s < \beta$ then the MF-index is traversed to the level containing frequent subgraphs of size s and an appropriate number of CAM codes are randomly selected and transformed to frequent queries. Otherwise, the DF-index is invoked to randomly select CAM codes of size-s subgraphs to generate the queries. Similarly, the infrequent index is scanned to randomly select a specified number of CAM codes of size-s DIFs, which are then transformed to DIF queries. Lastly, the generation of NIF queries is relatively more involved as this type of infrequent subgraphs are not indexed. However, a NIF contains at least one DIF [7]. Hence, the infrequent index is scanned first to randomly choose the CAM code of a DIF f whose size is less than s. Then the $fsgIds(f)$ is used to retrieve a set of graphs in $\mathcal{D}$ containing f where the size of each graph G is at least s. The subgraph f in each G is expanded to f' by traversing it until $|f'| = s$. Then f' is returned as a candidate NIF query. The above steps are repeated iteratively until the specified number of NIF queries are generated.

The Query Simulator module: This module simulates the blending of visual query formulation and query processing by estimating the GUI latency available at each formulation step,

which is leveraged by the *Query Execution module* to retrieve candidate matches. Most visual interfaces for graph query formulation [5], [6] comprise of at least three key panels: (a) A *Label Panel* to display the set of labels of nodes or edges of the underlying data graphs; (b) A *Query Panel* for constructing a subgraph query graphically by adding a fragment iteratively; (c) A *Results Panel* that displays the query results.

Specifically, a user may take the following steps to formulate a query graph[1]. (1) Move the mouse cursor to the *Label Panel*. (2) Scan and select a label (*e.g.,* label C). (3) Drag the selected item to the *Query Panel* and drop it. Each such action represents formulation of a query node (denoted by u) with the specified label. (4) Repeat Steps 1–3 for constructing another node v. (5) Construct an edge between u and v by clicking on them, respectively. (6) Repeat Steps 4 and 5 until the complete query is formulated.

Observe that the key challenge to simulate the aforementioned steps is that different users may take different time to complete each step. As mentioned earlier, it is important to accurately estimate the time taken by a user for each step as this latency is exploited by the query processing paradigm to prefetch candidate matches [3], [7]. Hence, instead of assuming any arbitrary time to finish each of the aforementioned steps, we need a systematic approach to estimate them. Here we drew upon the literature in HCI to quantitatively model the time available to perform each step.

Let us now refer to the times taken to complete Steps 1, 2, 3, and 5 as *movement time* (denoted by T_m), *selection time* (T_s), *drag time* (T_d), and *edge construction time* (T_e), respectively. We can quantify these times as follows.

Estimating movement time T_m. Reconsider Step 1. It involves acquisition of a target in the *Label Panel* at a distance D from the mouse cursor which is in the *Query Panel*. Note that typically the *Label Panel* is a rectangular two-dimensional target. Hence, we adopt the model in [1] that focuses on acquiring targets having rectangular, square, or circular shapes. The movement time T_m is quantified as follows.

$$T_m = a + b \log_2 \left(\sqrt{\left(\frac{D}{W}\right)^2 + \eta \left(\frac{D}{H}\right)^2} + 1 \right) \quad (1)$$

where D is the *Label Panel*'s distance to the cursor, H and W denote the Panel's height and width, respectively. The parameter a varies in the range of [-50, 200], b in [100, 170], and η in [1/7, 1/3]. We set $\eta = 0.33$ according to [1].

Estimating selection time T_s. The above model for computing T_m can only be applied if the mouse movement is one-directional and involves a single target which is rectangular, square, or circular in shape. Consequently, selection of a label item cannot be modeled using it. Observe that searching for a label involves moving the cursor over multiple targets to select an item. In fact, the *Label Panel* is similar to a hierarchical menu and one needs to select an item during query formulation by navigating the cursor through the hierarchy using predominately vertical movements to select the desired label.

Note that we assume the labels are organized vertically and hence ignore horizontal movements in this panel as the horizontal width is negligible here[2]. Furthermore, we assume that the items in this panel are organized in a specific order (*e.g.,* lexicographically ordered). Hence, a user can move to the direction of the target item rapidly using an "open loop" movement. Consequently, we adopt the following *logarithmic* model proposed by Ahlström [2] for modeling selection time of an item, which integrates both the time to find the item and the time to move to the target.

$$T_s = m + n \times (\log_2(p + 1)) \quad (2)$$

where p is the *position number* of the target label, and m and n are empirically-determined constants.

Estimating drag time T_d. The *drag time* can be modeled using Equation 1 as the cursor is now moving from the *Label Panel* to the *Query Panel*. Specifically, in this case D is the *Query Panel*'s distance to the cursor (we assume that the label is dropped in mid region of the panel), H and W denote *Query Panel*'s height and width, respectively.

Estimating edge construction time T_e. Lastly, T_e models the Step 5 of the query formulation process comprising of (a) clicking on the node v (assuming the last node dropped on the panel is v); (b) moving the cursor from v to u; (c) clicking on the node u. Let t_c be the time to click a node, n_c be the number of clicks on a node (in our case $n_c = 1$), and T_m be the movement time from v to u. Then,

$$T_e = 2n_c \times t_c + T_m \quad (3)$$

We assume $t_c = 80ms$ [4]. Note that T_m can be computed using Equation 1 where H and W now denote the height and width of a node (constant for all nodes in a given GUI)[3], respectively, and D is the shortest distance between u and v.

This module implements the above model to simulate the visual query construction process. Given a query (generated by the *Visual Query Generator* module), we traverse it in a depth-first manner to construct it iteratively in Panel 4 (Figure 2). For each step in the aforementioned query construction process, the algorithm *waits* for an appropriate amount of time (as quantified above) to simulate the execution of the task by a user before moving to the next step. Each node that has been traversed is colored red in the query indicating it has been formulated. Observe that a query graph can be drawn by following different sequence of edge construction. Hence, VISUAL allows automatic simulation of all possible query formulation sequences iteratively for a given query.

The Query Execution module: This module implements the visual subgraph query processing paradigm described in [3], [7] where the GUI latency (created by the *wait times*

[1] In this demonstration, we assume "edge-at-a-time" visual query formulation strategy. Hence usage of built-in *canned patterns* (*e.g.,* benzene ring) for composing queries is beyond the scope of this work.

[2] The model in [2] can support horizontal movement as well in the case that the labels are displayed as a "matrix"-like structure.

[3] [1] can model square, rectangular, or circular-shaped query nodes.

in the above simulation process) is leveraged to prefetch candidate data graphs. We do not focus on this module in detail here as it is already demonstrated in [6].

The Simulation Results Viewer module: This module is invoked (by clicking on the `Display Chart` button) to provide a real-time three-level graphical view of the simulation results. At the *query collection level*, it enables visualization of results of an entire collection of subgraph queries. For example, Figure 3(a) plots the *average user latency time* (corresponds to the wait times in the *Visual Query Simulator* module) that is available for drawing each edge (shown by black dotted line) and corresponding *average prefetching time* (shown by red dotted line) of candidate data graphs for a collection of 100 size-5 queries. Figure 3(b) plots the *average* SRT*s* of each query (averaged over all query formulation sequences) in a collection of 20 queries. At the *query level*, the viewer depicts the performance of all query formulation sequences (QFS) of a specific query. Figures 3(c)-(d) depict the performance of a size-5 query for a set of QFS. At the QFS *level*, the results of a specific QFS of a query graph can be visualized. For instance, Figure 3(e) depicts the query evaluation process of a specific QFS at every step. The bottom part of the screen displays the sizes of candidate data graphs at different steps. The top part of the display plots the time taken by the *Query Execution* module to compute and maintain the candidate data graphs at every step along with the user latency time.

III. Related Systems and Novelty

There has been considerable research in querying graph data. There has also been research in visual query languages for graph databases [5], [9]. However unlike ViSual, none of these efforts focus on building an HCI-inspired visual query simulator to evaluate query performance as these efforts follow the conventional query formulation and processing paradigm (visual or textual).

More germane to this work is our previous research in [3], [6], [7] where we propose a novel paradigm of blending graph query processing with visual query formulation. Specifically, these work focus on the *Index Constructor* and *Query Execution* modules in Figure 1. In contrast, ViSual is built on top of them to realistically simulate the visual query construction process in this query processing paradigm. That is, it focuses on the *Visual Query Generator*, *Query Simulator*, and the *Simulation Results Viewer* modules.

IV. Demonstration Objectives

ViSual is implemented in Java JDK 1.7. Our demonstration will be loaded with synthetic datasets and a few real datasets (e.g., AIDS Antiviral dataset containing 43K graphs) with different sizes. The key objective of the demonstration is to enable the audience to interactively experience the following modules through the ViSual GUI. A video of ViSual is available at http://www.youtube.com/watch?v=TzDmIxTylew.

Interactive experience with the Visual Query Generator module. Through the GUI (Figure 2), one will be able to select the relevant data source (Panels 1 and 2), specify the details of subgraph queries she would like to generate (Panel 3),

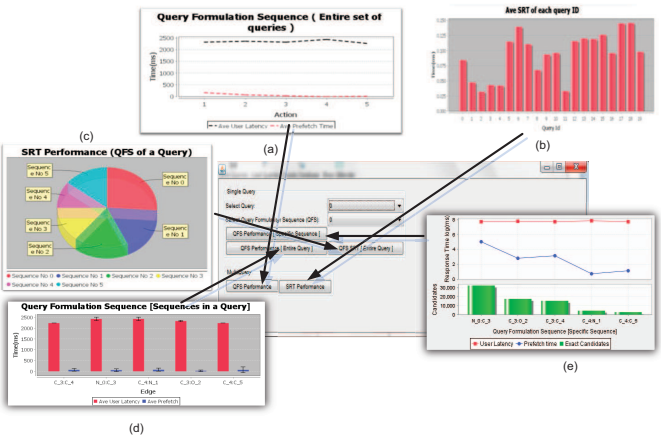

Fig. 3. [Best viewed in color] Visualization of simulation results.

and (optionally) view the generated visual queries in Panel 4. Specifically, she will be able to automatically generate a large number of subgraph queries having different characteristics without constructing them manually.

Interactive experience with the Query Simulator module. After generating the visual queries, one will be able to simulate the proposed paradigm *without* manually constructing any query. In particular, once she clicks on the `Simulate` button in Panel 5, she will be able to observe the construction process of a query as well as the wait times (GUI latency) during each query formulation step (Panel 4). Also, by modifying various settings of the model parameters (Panel 5), she will be able to experience the impact of these parameters on the simulation process.

Interactive experience with the Simulation Results Viewer module. Once the simulation of the proposed paradigm is completed for the chosen query set, one will be able to visualize detailed performance of the queries at different levels of granularity (Figure 3) and appreciate the fact that the simulated GUI latency at each step is leveraged for the prefetching candidate matches. Importantly, as discussed earlier the user will be able to observe multi-faceted performance results of a large number of synthetic subgraph queries without manually constructing them.

Acknowledgment

Sourav S Bhowmick was supported by the Singapore-MOE AcRF Tier-1 Grant RG24/12. Byron Choi was partially supported by the HKBU grant FRG2/12-13/079.

References

[1] J. Accot, S. Zhai. Refining Fitts' Law Models for Bivariate Pointing. *In ACM SIGCHI*, 2003.
[2] D. Ahlstrom. Modeling and Improving Selection in Cascading Pull-Down Menus Using Fitt's Law, the Steering Law, and Force Fields. *In CHI*, 2005.
[3] S. S. Bhowmick et al. VOGUE: Towards A Visual Interaction-aware Graph Query Processing Framework. *In CIDR*, 2013.
[4] S. Card et al. The Keystroke-level Model for User Performance Time with Interactive Systems. *CACM*, 23(7), 1980.
[5] D. H. Chau et al. GRAPHITE: A Visual Query System for Large Graphs. *ICDM Workshop* , 2008.
[6] C. Jin et al. GBLENDER: Visual Subgraph Query Formulation Meets Query Processing. *In SIGMOD*, 2011.
[7] C. Jin et al. PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. *In ICDE*, 2012.
[8] X. Yan et al. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.
[9] S. Yang et al. SLQ: A User-friendly Graph Querying System. *In SIGMOD*, 2014.