AURORA: Data-driven Construction of Visual Graph Query Interfaces for Graph Databases

Sourav S Bhowmick[‡] Kai Huang [‡,§] Huey Eng Chua[‡] Zifeng Yuan[‡,§]
Byron Choi[†] Shuigeng Zhou[§]

[‡]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[§]Shanghai Key Lab of Intelligent Information Processing, Sch. of Computer Science, Fudan University, China
[†]Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR
assourav|hechua@ntu.edu.sg,bchoi@comp.hkbu.edu.hk,khuang14|zfyuan16|sgzhou@fudan.edu.cn

ABSTRACT

Several commercial and academic frameworks for querying a large collection of small- or medium-sized data graphs (*e.g.,* chemical compounds) provide visual graph query interfaces (a.k.a GUI) to facilitate non-programmers to query these sources. However, construction of these visual interfaces is not *data-driven.* That is, it does not exploit the underlying data graphs to *automatically* generate the contents of various panels in a GUI. Such data-driven construction has several benefits such as facilitating efficient subgraph query formulation and portability of the interface across different application domains and sources. In this demonstration, we present a novel data-driven visual subgraph query interface construction engine called AURORA. Specifically, given a graph repository D containing a collection of small- or medium-sized data graphs, it automatically generates the GUI for D by populating various components of the interface. We demonstrate various innovative features of AURORA.

ACM Reference Format:
Sourav S Bhowmick, Kai Huang, Huey Eng Chua, Zifeng Yuan, Byron Choi, Shuigeng Zhou. 2020. AURORA: Data-driven Construction of Visual Graph Query Interfaces for Graph Databases. In 2020 International Conference on Management of Data (SIGMOD '20), June 14–June 19, 2020, Portland, OR, USA. ACM, NewYork, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384681

SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384681

1 INTRODUCTION

Visual graph query interfaces (a.k.a GUI) widen the reach of graph querying frameworks by enabling non-programmers to use them. Consequently, several commercial and academic frameworks for querying a large collection of small- or medium-sized data graphs (*i.e.,* graph database) expose such interfaces for formulating subgraph queries. For instance, *PubChem* provides a visual query interface for structure-based chemical compound search (https://pubchem.ncbi.nlm.nih.gov/edit3/index.html).

A core component of many real-world subgraph query interfaces is a panel containing a set of *patterns* (*i.e.,* small subgraphs), which potentially expedites a visual query formulation task by reducing the number of query formulation steps (steps for brevity) [3, 8]. Specifically, a pattern enables a user to construct multiple nodes and edges in a subgraph query by performing a *single* click-and-drag action (*i.e., pattern-at-a-time* mode) in lieu of iterative construction of edges one-at-a-time (*i.e., edge-at-a-time* mode). For example, the GUI of *PubChem* provides a library of patterns containing benzene ring, carboxyl group, etc.

The contents of several components of a GUI are typically created *manually* based on domain knowledge [3, 8]. For example, the patterns to be displayed on a GUI are manually selected by a programmer or a domain expert and then coded using a programming language. Unfortunately, such manual creation is very challenging as it demands a comprehensive topological knowledge of the underlying data graphs. As a result, the selected content may not be *diverse* enough to expedite formulation of a wide range of subgraph queries [8]. Naturally, such manual selection also limits the portability of GUIs across different domains and sources as one has to reimplement and customize the GUI for each one of them [3].

In this demonstration, we present a novel data-driven visual subgraph query interface construction system called AURORA (d**A**ta-driven q**U**ery inte**R**face c**O**nstruction for g**R**aph d**A**tabases). Given a graph database D, AURORA automatically

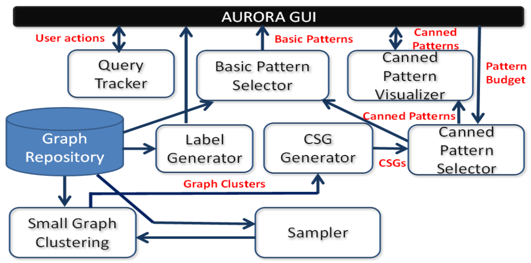

Figure 1: Architecture of AURORA.

populates various panels (*e.g.*, node labels, patterns) of the GUI from D. Although the set of labels of nodes/edges for populating the *Attribute* panel can be easily generated by traversing the underlying data graphs, automatically generating diverse patterns is a NP-hard problem [8]. Particularly, the data-driven nature of AURORA makes it highly *portable* as it can automatically construct the visual query interface for *any* domain-specific graph querying application (*e.g.*, drug discovery, computer vision) centered around a collection of small- or medium-sized data graphs. Note that AURORA focuses on the interface for query formulation and not for visualization of query results. The latter is application- and query-specific and can be built on top of the former.

It is worth noting that AURORA goes against the traditional mantra of *"one constructor does not fit all"* in visual query interface generation. Visual interfaces for graph query formulation are traditionally manually constructed for each data source or domain. We argue that as more and more graph data sources become prevalent in a wide variety of domains, *one constructor will have to fit all* in order to minimize the cost of development and maintenance of user-friendly interfaces that support efficient query formulation. The AURORA approach paves the way for a *single* framework to automatically construct the query formulation interface for any domain or source involving graph databases.

2 DESIGN PHILOSOPHY

AURORA is designed to give end users the freedom to easily and quickly construct a visual query interface for any data source without resorting to coding. Concretely, its design is based on the following three principles:

(1) **Work with independent graph databases.** Our data-driven approach should be able to work with any data source or application domain centered around a collection of small- or medium-sized data graphs. As the number of such sources grows with time, AURORA will offer sufficient benefits to developers and end users of graph databases by making generation of visual interfaces for query formulation effortless.
(2) **Cognitive load-aware pattern selection.** A key issue in exposing patterns to support query formulation is that a user should be able to visually interpret a pattern (*i.e.*, edge relationships) quickly so that she can determine if it is useful for her query. Such efficient interpretation naturally reduces the overall time a user may take to formulate a query. Hence, the AURORA framework needs to select subgraphs that potentially impose low *cognitive load* on users.
(3) **Oblivious to query logs.** AURORA is query log-oblivious as such log data may often be unavailable in practice, especially in "cold start" cases. In the case where query logs are available, the pattern generation framework should be easily extensible to incorporate them [8].

In practice, any graph database system can easily integrate AURORA in the following way. It can first invoke AURORA to generate the GUI for a data source and then install it on top of the query engine for query formulation and processing.

3 SYSTEM OVERVIEW

Figure 1 depicts the architecture of AURORA. It consists of the following components.

The GUI module. Figure 2 depicts a screenshot of the AURORA GUI. *Panel 1* enables a user to select a graph database, specify a *pattern budget* for *canned* patterns (minimum and maximum size, number of patterns to display on the GUI), and load previously generated patterns. Specifically, when a user invokes the GUI of AURORA, she first chooses her graph repository of interest and specifies the pattern budget using this panel. Note that the user can easily customize the interface according to her needs by specifying the budgets accordingly. *Panel 2* contains a list of distinct vertex labels in the selected dataset. An edge label list can be easily incorporated if required. *Panel 3* tracks statistics related to a subgraph query formulation activity. *Panel 4* is used for query formulation. AURORA generates two types of patterns, *basic* and *canned*, and displays them in *Panels 5* and *6*, respectively. The former are patterns with size less than 3 (*e.g.*, 1-edge, 2-edge) and the latter are larger in size.

Observe that the contents of Panels 1 and 4 are provided by a user and the contents of Panel 3 depends on a user's actions in Panel 4. On the other hand, the contents of Panels 2, 5, and 6 depend on the graph repository. Hence, the goal of AURORA is to automatically populate Panels 2, 5, and 6.

Small graph clustering module. A large collection of small- or medium-sized data graphs D is likely to contain groups of graphs having similar topology. Hence, the goal of this module is to partition D into a set of *graph clusters* $C = \{C_1, C_2, \ldots, C_d\}$, where $C_i \subseteq D$, $C_i \cap C_j = \emptyset \; \forall i \neq j$, and it maximizes a *clustering property* objective function $f : C \rightarrow \mathbb{R}$. Here we present an overview of the generation of C. The reader may refer to [8] for details.

A 2-step clustering approach is used to partition D. The first step (*coarse clustering*) is a *feature vector-based* approach

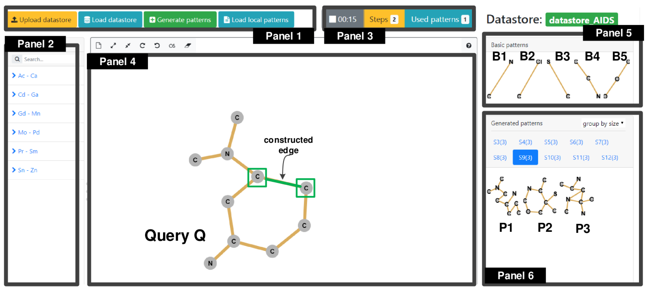

Figure 2: Visual interface constructed by AURORA.

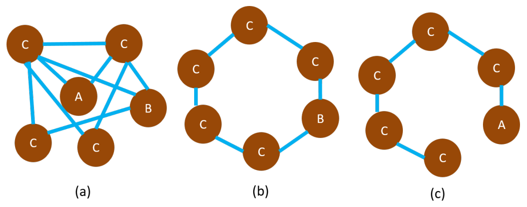

Figure 3: Cognitive load and diversity of patterns.

that uses *frequent subtrees* [4] of D as feature vector for k-means clustering where the k seeds are chosen using the k-means++ algorithm [2]. Note that the clusters (referred to as *coarse clusters*) generated by coarse clustering may still be large and expensive for generating *cluster summary graphs* (CSGs). The second step (*fine clustering*) is performed on those coarse clusters that exceed the maximum cluster size threshold N. In particular, it leverages *maximum connected common subgraph* (MCCS) as the clustering property.

Sampler module. Since clustering can be computationally expensive for large D, *eager sampling* is performed *prior* to the small graph clustering phase and the *lazy sampling* is performed *after* the coarse clustering phase [8]. The former generates a random sample of data graphs from D. Since some clusters generated by coarse clustering may still be too large for efficient processing, the latter performs stratified random sampling of large clusters to further reduce their sizes. Note that fine clustering still needs to be performed if the size of the sampled graphset is larger than N.

CSG generator module. This module [8] summarizes data graphs in *each* cluster $C_i \in C$ into a *closure graph* generated by leveraging the technique in [5]. We refer to it as *cluster summary graph* (CSG). A *closure graph* [5] is a generalized graph generated by performing a *union* on the structures of a set of graphs. It integrates graphs of varying sizes into a single graph referred to as *extended graph* (denoted by $G^* = (V^*, E^*)$) by inserting dummy vertices or edges with a special label ε such that every vertex and edge is represented in G^*. Given two extended graphs G_1^* and G_2^* and a *mapping* ϕ between them, a *vertex closure* and an *edge closure* can be obtained by performing an element-wise union of the attribute values of each vertex and each edge in the two graphs, respectively. Then the *closure graph* of G_1^* and G_2^* is a labelled graph $G_c = (V_c, E_c)$ where V_c is the vertex closure of V_1^* and V_2^* and E_c is the edge closure of E_1^* and E_2^*. Note that these closures may contain attribute values ε corresponding to a dummy vertex or edge, which are removed from G_c.

Label generator module. This module traverses the underlying data graphs to generate the set of unique labels in D, which is then displayed on the GUI (Panel 2).

Canned pattern selector module. Given a user-specified *pattern budget* $b = (\eta_{min}, \eta_{max}, \gamma)$ where $\eta_{min} > 2$ (resp. η_{max}) is the minimum (resp. maximum) size of a canned pattern and γ is the number of patterns to be displayed on Panel 6, the aim is to select a set of canned patterns $\mathcal{P}$ satisfying b from the set of CSGs $\mathbb{S}$.

In particular, the goal is to select $\mathcal{P}$ by maximizing *coverage* and *diversity* and minimizing *cognitive load* of $\mathcal{P}$ [8]. A pattern $p \in \mathcal{P}$ *covers* a data graph $G \in D$ if G contains a subgraph s that is isomorphic to p. The pattern set $\mathcal{P}$ should ideally cover as many data graphs in D as possible. Furthermore, it should contain structurally diverse patterns to make efficient use of the limited display space on the GUI. For example, consider the patterns in Figures 3(b) and (c). AURORA avoids selecting both simultaneously for display as they are structurally very similar. Note that a user needs to view the canned patterns in Panel 6 during query formulation and determine the ones that are relevant. However, recent research reveal that viewing a large, dense and complex graph may demand substantial *cognitive load* (*i.e.,* memory demand or mental effort required to perform a given task) from a user [6]. For example, intuitively a user may take more time to visually inspect the pattern in Figure 3(a) to determine if it is useful for formulating her query compared to the one in Figure 3(b) or Figure 3(c). This is because the former pattern is denser and more complex than the latter. Hence, given two candidate patterns p_1 and p_2, AURORA prefers p_1 to p_2 if p_1 has lower cognitive load than p_2. In particular, in AURORA, diversity and cognitive load of a pattern are computed using a graph edit distance-based and graph density-based techniques, respectively [8].

Given a set of CSGs $\mathbb{S}$, a greedy iterative approach based on *weighted random walks* is undertaken for selecting canned patterns based on the aforementioned constraints [8]. First, each edge in every CSG is assigned a *weight* based on its label coverage in the dataset and in the cluster. Next, it performs random walks on these weighted CSGs. Given a weighted CSG S, for each size in the range $[\eta_{min} - \eta_{max}]$ (*i.e.,* pattern budget b), it leverages on the statistics obtained from the walks to propose a variety of *potential candidate patterns* (PCP) from which a *final candidate pattern* (FCP) is derived. Note that each random walk starts with a *seed edge* (*i.e.,* edge with largest weight). The walk proceeds by moving to an adjacent edge until the required number of edges is traversed or when no more edges can be added. The adjacent edge is selected from a list of *candidate adjacent edges* (CAE).

Particularly, edges with higher weights are selected with higher probability. Then, the FCP of a particular size η_i for a CSG is found by retrieving a connected subgraph of size η_i with the most frequently traversed edges during the walks. A *pattern score* s_p based on coverage, diversity, and cognitive load is computed for each FCP. The candidate pattern with the largest pattern score is selected as the best pattern to be added to $\mathcal{P}$. Weights of the CSGs are then updated using the *multiplicative weights update* approach [1]. These steps are repeated until either γ patterns are found or when no new pattern can be found.

Note that the above approach can be extended to handle query logs (if they are available). Specifically, the weights of the edges in S can be modified to incorporate information of subgraphs that appear frequently or infrequently in a log.

Basic pattern selector module. This module generates the basic patterns (*edge*, *2-edge*) after the selection of canned patterns. Specifically, Panel 5 in AURORA GUI provides m (default is 5) basic patterns. The steps for selecting the basic patterns for *each* pattern type (the *edge* pattern is used for illustration) are as follows: (1) Rank the edges in decreasing level of support in D. The ranked list is denoted as E_r. (2) Compute the number of steps ($step_e(e_1)$) required to draw e_1 on Panel 4 using edge-at-a-time approach where $e_1 \in E_r$ is the edge with the highest support. (3) Compute the minimum number of steps ($minStep_p(e_1)$) required to draw e_1 on Panel 4 using pattern-at-a-time approach. Note that patterns in Panels 5 and 6 are utilized here and edge deletion is allowed. (4) Select e_1 as a basic pattern if $step_e(e_1) < minStep_p(e_1)$ and $|\mathcal{B}_{edge}| < \alpha$ where $\mathcal{B}_{edge}$ is a set of basic edge pattern and α is the maximum number of such patterns allowed in Panel 5. Remove e_1 from E_r. (5) Repeat Steps 2 to 4 until $|\mathcal{B}_{edge}| = \alpha$. Note that 2-edge basic patterns can be selected by following similar steps after selecting 2-edge candidates using any state-of-the-art frequent pattern mining technique.

Pattern visualizer module. This module facilitates visualization of the patterns on the GUI. All patterns are displayed using *ForceAtlas2* layout [7]. A user may select different options (group-by-size, single-page, x-per-page) available in a drop-down box of Panel 6 to display the canned patterns.

Query tracker module. Finally, this module tracks various information related to subgraph query formulation activities (*e.g.,* number of steps taken, query formulation time).

4 RELATED SYSTEMS

Most germane to this work is the DaVinci system presented in [9]. In particular, it neither incorporates frequent tree-based clustering nor sampling strategies to handle larger graph databases. Furthermore, canned patterns are generated greedily using breadth-first-search by ignoring diversity and cognitive load of these patterns. Also, candidate pattern generation and selection are treated as two separate phases. In contrast, they are intertwined in AURORA by utilizing a random walk-based strategy. Additionally, basic patterns are not extracted in [9]. Lastly, the GUI of AURORA is different and superior to DaVinci. Panels 3 and 5 are unavailable in DaVinci. Our canned pattern display panel (Panel 6) is more flexible and compact compared to DaVinci. In summary, except for the *Label Generator* and *CSG Generator* modules, all modules are novel in AURORA compared to DaVinci [9].

5 DEMONSTRATION OVERVIEW

AURORA is implemented in Java JDK 1.8 and Javascript 1.6. Our demonstration will be loaded with real datasets (*e.g.,* AIDS). Example query graphs that can be constructed using the patterns will be presented for formulation. Users can also write their own ad-hoc queries through our GUI. The key objectives of the demonstration are as follows.

Data-driven construction of GUI. Through AURORA's GUI (Figure 2), the audience will be able to select a graph database and a pattern budget using Panel 1 to automatically construct the contents of Panels 2, 5, and 6. One will also be able to interactively select different graph repositories as well as pattern budgets (through Panel 1) to appreciate the portable and data-driven nature of AURORA. Specifically, by utilizing the same interface (Figure 2), one will be able to generate different graph query interfaces across different application domains and sources.

Formulation of subgraph queries. Using Panels 2, 5, and 6, an audience can quickly and interactively formulate all possible subgraph queries on a graph database. An audience may also formulate the same query using a commercial GUI such as *PubChem* and experience first-hand query construction inefficiencies (using Panel 3) of these interfaces.

A **short video** to illustrate the main features using example use cases is available at https://youtu.be/pUfVzU2HDJk.

Acknowledgments. The first four authors are supported by the AcRF Tier-2 Grant MOE2015-T2-1-040. Byron Choi is supported by HKRGC GRF 12201119 and 12201518.

REFERENCES

[1] S. Arora, E. Hazan, S. Kale. The Multiplicative Weights Update Method: a Meta-Algorithm and Applications. *Theory Comput.* 8(1), 2012.
[2] D. Arthur, S. Vassilvitskii. k-means++: The Advantages of Careful Seeding. *In SIAM*, 2007.
[3] S. S. Bhowmick, et al. Data-driven Visual Graph Query Interface Construction and Maintenance: Challenges and Opportunities. *PVLDB*, 9(12), 2016.
[4] Y. Chi, et al. Indexing and Mining Free Trees. *In ICDM*, 2003.
[5] H. He, A.K. Singh. Closure-tree: An Index Structure for Graph Queries. *In ICDE*, 2006.
[6] W. Huang, P. Eades, S.H. Hong. Measuring Effectiveness of Graph Visualizations: A Cognitive Load Perspective. *Inf. Vis.*, 8(3), 2009.
[7] M. Jacomy, T. Venturini, S. Heymann, M. Bastian. ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. *PloS one*, 9(6), e98679, 2014.
[8] H. Kai, H.-E Chua, et al. CATAPULT: Data-driven Selection of Canned Patterns for Efficient Visual Graph Query Formulation. *In SIGMOD*, 2019.
[9] J. Zhang, et al. DaVinci: Data-driven Visual Interface Construction for Subgraph Search in Graph Databases. *In ICDE*, 2015.