DaVinci: Data-driven Visual Interface Construction for Subgraph Search in Graph Databases

Jinbo Zhang* Sourav S Bhowmick§ Hong H. Nguyen§ Byron Choi‡ Feida Zhu†

§School of Computer Engineering, Nanyang Technological University, Singapore
†School of Information Systems, Singapore Management University, Singapore
*School of Electronics Engineering and Computer Science, Peking University, China
‡Department of Computer Science, Hong Kong Baptist University, Hong Kong
assourav@ntu.edu.sg, choi@hkbu.edu.hk, fdzhu@smu.edu.sg

***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 a wider community. Despite the significant progress towards building such query interfaces to simplify visual subgraph query formulation task, construction of current generation 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 the interface. Such data-driven construction has several benefits such as superior support for subgraph query formulation and *portability* of the interface across different graph databases. In this demonstration, we present a novel data-driven visual subgraph query interface construction engine called DaVinci. Specifically, it automatically generates from the underlying database two key components of the visual interface to aid subgraph query formulation, namely *canned patterns* and *node labels*.**

I. Introduction

Formulation of a textual query using a database query language (*e.g.,* SQL, SPARQL) often demands considerable cognitive effort from end users. A popular approach to tackle this challenge is to improve the user-friendliness of the task by providing a visual query interface to replace data retrieval aspects of a query language. As many important real-world applications are centered on graph data, there is a growing need to build such user-friendly visual framework (*e.g.,* [1], [4], [7]) on top of any state-of-the-art graph query processing engine (*e.g.,* [5]) to enable easy formulation of subgraph search queries. Such queries retrieve a set of data graphs, each containing subgraph(s) that exactly or approximately match a user-specified query graph[1].

Typically, a visual interface for subgraph query formulation is composed of several panels such as a panel to display the set of labels of nodes or edges of the underlying data graphs, a panel to construct a subgraph query graphically, a panel containing *canned patterns* (small graph-structured patterns) to aid query formulation, and a results panel. For example, Figure 1 depicts the screenshot of a visual interface provided by PubChem[2] for substructure or subgraph search on chemical compounds. Specifically, Panel 3 provides a list

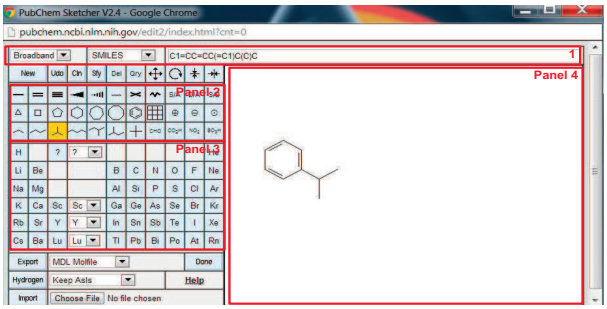

Fig. 1. GUI for substructure search in PubChem.

of chemical symbols that a user can choose from to assign labels to nodes of a query graph. Panel 2 lists a set of *canned patterns* (*e.g.,* benzene ring) which a user may drag and drop in Panel 4 during query construction. Note that the availability of such patterns greatly improves usability of the interface by enabling users to quickly construct a query graph with fewer clicks compared to constructing it in an "edge-at-a-time" mode. For instance, the query graph in Panel 4 can be constructed by dragging and dropping two such canned patterns from Panel 2 instead of taking the tedious route of constructing 9 edges iteratively. Particularly, Panel 2 is useful if (a) there is sufficiently diverse collection of patterns that can aid a user to formulate most of her queries; and (b) a user can quickly absorb and find relevant patterns from the collection.

Such user-friendly visual query interface is typically built by leveraging decades of research (by the HCI community) related to various theoretical models of visual tasks, menu design, and human factors. Unfortunately, despite the significant progress this community brought towards constructing user-friendly query interfaces, such approach suffers from at least two key drawbacks. First, the contents of several key components (*e.g.,* Panels 2 and 3) are often created *manually* based on domain knowledge rather than by automatic generation from the underlying database. For instance, the patterns in Panel 2 are manually selected and added to the GUI. An immediate aftermath of such manual selection is that the set of canned patterns may not be sufficiently diverse enough to support a wide range of subgraph queries as it is unrealistic to expect a domain expert to have comprehensive knowledge of the

[1]In this demonstration, we focus on subgraph queries that are processed on a large number of small or medium-sized graphs.

[2]http://pubchem.ncbi.nlm.nih.gov/edit2/index.html?cnt=0

topology of the entire graph dataset. Consequently, an end user may not find the canned patterns in Panel 2 useful in formulating certain query graphs. Similar problem may also arise in Panel 3 where the labels of nodes may be manually added instead of automatically generated from the underlying data. Second, such visual interface lacks of portability as the same interface cannot be seamlessly integrated on a graph database in a different application domain (*e.g.,* computer vision). As the contents of Panels 2 and 3 are domain-dependent and manually created, the GUI needs to be reconstructed from scratch when the domain changes in order to accommodate new domain-specific canned patterns and labels.

There is one common theme that runs through the aforementioned limitations: *the visual query interface construction is not data-driven*. Specifically, the GUI does not analyse the underlying data graphs to automatically generate the contents of various panels. In this demonstration, we present a novel data-driven visual subgraph query interface construction framework called DAVINCI (**DA**ta-driven **V**isual **IN**terface **C**onstruction Eng**I**ne) that, *to the best of our knowledge, is world's first endeavor in the context of graph search*. While the unique set of labels of nodes or edges of the data graphs (Panel 3) can be easily generated by traversing the underlying data graphs, automatically generating the set of canned patterns is computationally challenging. These patterns should not only be able to *maximally cover* the underlying data graphs but should also minimize *topological similarity* (redundancy) among themselves so that a diverse set of canned patterns is available to the user. Note that there can be prohibitively large number of such patterns. Hence, the size of the pattern set should not be too large due to limited display space on the GUI as well as users' inability to absorb too many displayed patterns for query formulation.

II. System Overview

Figure 2 shows the system architecture of DAVINCI. The *Node Label Generator* module traverses the underlying data graphs to generate the set of unique labels in the database $\mathcal{D}$, which are then displayed on the GUI. Since this is a straightforward technique, we do not elaborate on it further. The *Cluster Generator* module constructs clusters of data graphs from $\mathcal{D}$ where the similarity among data graphs in the same cluster is high while it is low for graphs in different clusters. The *Closure Graph Set Computation* module *combines* all the data graphs in each cluster into a single graph called the *closure graph* based on their topological similarities. The *Canned Pattern Generator* module then extracts a collection of canned patterns from the set of closure graphs[3]. These canned patterns are then displayed on the GUI grouped by their size. Note that the canned patterns are typically generated offline as they remain invariant for a given database instance. The *Query Processor* and the *Results Visualizer* modules are used to evaluate the formulated query and display the query results, respectively. Although we add these two modules

[3]The current version of DAVINCI does not assume the existence of query logs to generate canned patterns, which can be easily accommodated in the future.

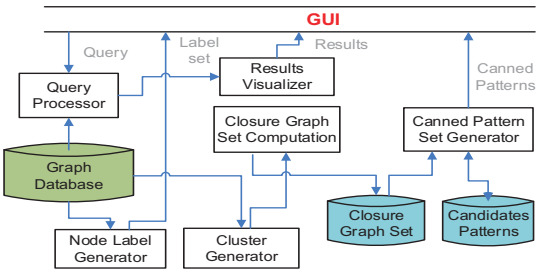

Fig. 2. Architecture of DAVINCI.

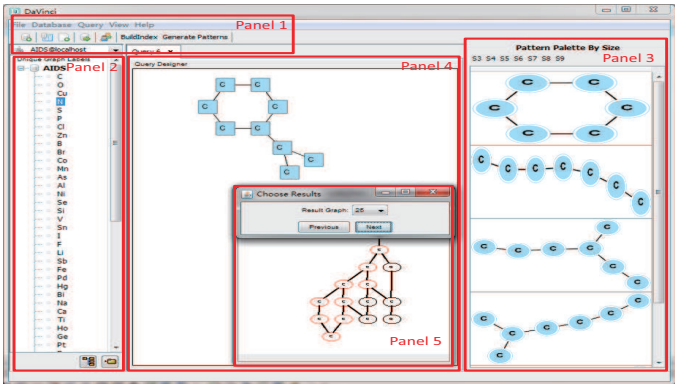

Fig. 3. The DAVINCI GUI.

in the architecture for the sake of completeness, they are orthogonal to this demonstration as any state-of-the-art graph query processing and results visualization techniques can be plugged into here. Hence, we do not elaborate on them further.

The GUI module: Figure 3 depicts the screenshot of the visual interface of DAVINCI. A user begins formulating a query by choosing a database as the query target and creating a new query canvas using Panel 1. Panel 2 displays the unique labels of nodes that appear in the dataset in lexicographic order (generated by the *Node Label Generator* module). In the query formulation process, the user may choose labels from Panel 2 to create the nodes in the query graph. Panel 3 displays the set of canned patterns grouped by their size (generated by the *Canned Pattern Generator* module). Panel 4 depicts the area for formulating subgraph queries by dragging and dropping labels and/or patterns from Panels 2 and 3. An edge between two nodes in a query graph can be created by left and right clicking on them. Panel 5 displays the results.

The Cluster Generator module: Given the set of data graphs in $\mathcal{D}$, this module partitions $\mathcal{D}$ into a set of clusters of data graphs. Let $SimScore(g_1, g_2)$ denotes the *similarity score* between a graph pair (g_1, g_2). Here, we use *maximum connected common subgraphs* (MCCS) [5] to compute similarity between a pair of graphs. First, it randomly chooses a data graph g_1 from $\mathcal{D}$ and then chooses another data graph g_2 that is least similar to g_1. The pair of (g_1, g_2) becomes the two *pivots* for partitioning. For all remaining data graphs $g_i \in \mathcal{D}$, it sorts them based on $Simscore(g_i, g_1) - Simscore(g_i, g_2)$ in ascending order. Then the data graphs in the first half of the list are associated with g_1 and the rest with g_2. This partitioning strategy is invoked recursively until the size of the cluster is below the

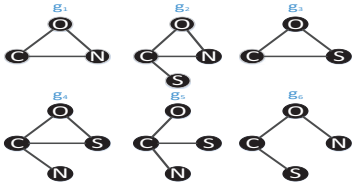

Fig. 4. A set of data graphs.

threshold *Cluster_size*, a parameter that is adjusted according to the characteristics of the underlying database. Note that computing similarity scores of all pairs of data graphs for cluster formation can be prohibitively expensive. Hence, here we exploit certain *topological feature-based similarity bounds* of the data graph pairs (wherever possible) to allocate them in the correct cluster without computing the scores.

The Closure Graph Set Computation module: Given the set of data graph clusters, the goal of this module is to generate a *closure graph* for each cluster that "summarizes" the content of each cluster. The intuition behind this step is that since each cluster represents topologically similar data graphs, it is convenient to generate a concise and accurate closure graph to represent them in contrast to attempting to find such closure graphs directly from $\mathcal{D}$. Observe that the closure graph set covers $\mathcal{D}$ effectively. We extend the idea of *graph closure* in [2] to compute them.

First, given a cluster C_i, it creates a *mapping* between a pair of data graphs (g_1, g_2) in C_i by *extending* each data graph with dummy vertices and edges such that each vertex and edge in g_1 has a corresponding mapping in g_2. This subsequently enables us to build the closure graph of data graphs even if they have different size. A dummy vertex or edge is assigned the label ε. Furthermore, each non-dummy vertex and edge is annotated with the identifier[4] of the original data graph it belongs to. For example, consider the data graphs g_1 and g_2 in Figure 4. The extended graph of g_1 is shown in Figure 5(a). Notice that the dummy vertex and edge are added to accommodate the vertex S in g_2.

A *mapping* between two extended data graphs $g'_1(V'_1, E'_1)$ and $g'_2(V'_2, E'_2)$ is given as $\phi : g'_1 \rightarrow g'_2$, where (a) $\forall v \in V'_1, \phi(v) \in V'_2$ and at least one of v and $\phi(v)$ is not dummy; furthermore, if both v and $\phi(v)$ are not dummy, then the labels of v and $\phi(v)$ should be the same and (b) $\forall e = (v_1, v_2) \in E'_1, \phi(e) = (\phi(v_1), \phi(v_2)) \in E'_2$ and at least one of e or $\phi(e)$ is not dummy. Figure 5(b) shows a mapping of the extended graphs of g_1 and g_2. It is easy to see that there are many ways to map a pair of data graphs. The one that uses least number of dummy vertices and edges is chosen here.

Given two extended graphs $g'_1(V'_1, E'_1)$ and $g'_2(V'_2, E'_2)$ and a mapping ϕ between them, the *closure graph* of g'_1 and g'_2 is a graph $g_c(V_c, E_c)$ where V_c is a set of *vertex closures* of V'_1 and V'_2 and E_c is a set of *edge closures* of E'_1 and E'_2. A *vertex* (*resp. edge*) *closure* in g_c is a vertex (*resp.* edge) whose attribute is a union of the attributes of the corresponding mapped vertices (*resp.* edges) of g'_1 and g'_2. Furthermore, each vertex (resp. edge) in g_c is annotated with an *idSet* which is the union

[4]We assume each data graph in $\mathcal{D}$ is assigned a unique identifier.

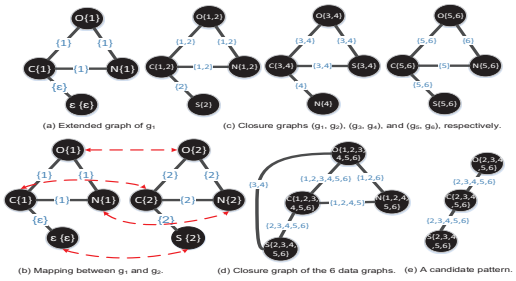

Fig. 5. An example of canned pattern generation.

of the graph identifiers of the corresponding mapped vertex (*resp.* edge) pairs. Note that the dummy labels are removed from the closure graph as well as some compression technique is employed to make the *idSet* space-efficient. For example, Figure 5(c) shows three closure graphs of data graph pairs (g_1, g_2), (g_3, g_4), and (g_5, g_6). Note that during construction of the closure graph of a pair of extended data graphs, all the matchings between the vertices and edges are established by computing the similarity between each pair of vertices using the *Neighbor Biased Mapping (NBM)* [2], which bias the matching towards neighbors of already matched vertices. The final closure graph to represent the set of data graphs in a cluster is built recursively from the data graphs and the closure graphs. For example, the final closure graph of all the six data graphs is shown in Figure 5(d), which is constructed recursively using the above closure graphs.

The Canned Pattern Set Generator module: Given the set of closure graphs, this module generates the canned patterns to be displayed on the GUI by traversing these closure graphs in a breath-first search fashion. It consists of two key steps, namely, *candidate pattern set generation* and *canned pattern set selection*. Let g be a subgraph in a closure graph where $|g| = |E|$ is the *size* of g and $Cov(g)$ be the *coverage* of g[5]. Then, in the first step, we use $|g|Cov(g)$ as the objective function and find candidate patterns (subgraphs) in the closure graph that maximize it. We illustrate this with an example. Consider Figure 5(d). The *idSet* of the edge (O, C) is largest, so the traversal begins from this edge. Specifically, since *idSet* $= \{1, 2, 3, 4, 5, 6\}$, so the value of the objective function is $1 * 6 = 6$. The edges that are adjacent to (O, C) along with the sizes of their *idSet*s $(((C, N), 4), ((C, S), 5), ((O, S), 2)$, and $((O, N), 3))$ are added into a priority queue which stores them in decreasing order of their *idSet* size. As (C, S) has the largest value, it is traversed next. Since the *idSet* of (C, S) is $\{2, 3, 4, 5, 6\}$, the current *idSet* is updated to $\{2, 3, 4, 5, 6\}$ (intersection of current *idSet* and the edge's *idSet*) and the value of the objective function becomes $2 * 5 = 10$. Similarly, (C, N) is traversed next and the *idSet* is updated (*idSet* $= \{2, 4, 5\}$). Since the value of the objective function is now reduced to $3 * 3 = 9$ (pattern size is 3), the pattern in Figure 5(e) is generated, which is added to the candidate pattern set. Next, the *idSet* of this pattern is removed from the closure graph as well as from the priority queue. Consequently, the value of the objective function will now be reset to 0. This process continues until

[5]The *coverage* of g is the number of data graphs that contain g.

no pattern can be generated or the closure graph is empty. The candidate pattern sets from all closure graphs are aggregated by removing duplicate patterns and aggregating their coverage.

Although the pattern set generated by the above step covers $\mathcal{D}$ maximally, it may be too large to fit in the limited space of the visual interface in its entirety. Hence, it is important to select a subset of these patterns as canned patterns for the GUI. Recall that one of the characteristics of the canned patterns is that they should be diverse (*i.e.,* the similarity between them should be minimal). Hence given a GUI *constraint* I[6], the *canned pattern set selection* step selects a subset of these candidate patterns that maximizes an objective function consisting of maximizing coverage and minimizing similarities among them. Note that this can be modeled as a *constrained optimization* problem which is NP-hard. Hence, we select the canned patterns greedily. First, the candidate patterns are grouped by their size and within each group the pattern p with the maximum coverage is selected. The coverage of each remaining candidate pattern p' in the group is updated by penalizing it by its similarity (computed using MCCS [5]) to p. This process is repeated until the selected pattern set satisfies I. Finally, these canned patterns are displayed on the GUI (grouped by size).

III. Related Systems and Novelty

There has been considerable research in the arena of visual query languages for relational databases, Web, XML databases, and graph databases (*e.g.,* [1], [4], [7]). These proposals typically focus on providing user-friendly strategies to increase expressiveness of visual queries and direct mappability to the textual query language. However, unlike DaVinci, they are not data-driven and do not generate the contents of various GUI components automatically.

In [3], we demonstrated a novel paradigm of blending subgraph query processing with visual query formulation. Specifically, it focused on the *Query Processor* and *Results Visualizer* modules in Figure 2. In contrast, DaVinci is built on top of these modules and its aforementioned components are orthogonal to these modules.

Lastly, it may seem that techniques deployed in DaVinci are related to the areas of frequent subgraph mining and graph summarization. Specifically, as some of the canned patterns may be frequent in the graph database, it can be generated using any frequent subgraph mining algorithm (*e.g., gSpan* [8]). However, data-driven visual interface construction cannot simply be realized by adopting a frequent mining algorithm due to the following reasons. First, the latter techniques may generate very large number of frequent patterns making it hard for users to locate a canned pattern for formulating queries. Second, it is not necessary for *all* canned patterns to be frequent. It is indeed possible that some patterns are frequently used by end users to formulate visual queries but are infrequent in the database. Similarly, graph summarization techniques (*e.g.,* [6]) focus on grouping nodes at different resolutions in a large network. In contrast, DaVinci generates a concise canned pattern set from a large collection of data graphs by maximizing coverage while minimizing redundancy.

[6]The GUI *constraint* can be expressed as follows: the GUI can only display canned patterns having size ranging from k_1 to k_2 (*e.g.,* 2 to 8) and the maximum number of patterns is M for each size.

IV. Demonstration Objectives

DaVinci is implemented in Java JDK 1.7 on top of the PRAGUE query engine [4]. Our demonstration will be loaded with a few real datasets (*e.g.,* AIDS Antiviral dataset containing 43K graphs, `PubChem`, `Protein Data Bank`) with different sizes. Example query graphs that can be constructed using the canned 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 to enable the audience to interactively experience the followings.

User-friendly construction of a data-driven visual subgraph querying interface. Through DaVinci's GUI (Figure 3), the audience will be able to select a graph database and click on the *Generate Pattern* button (Panel 1) to automatically construct the contents of Panels 2 and 3. One will also be able to interactively change the underlying graph database as well as the GUI constraint (through Panel 1) to appreciate the portable and data-driven nature of DaVinci. Specifically, as the graph database or GUI constraint changes, one will be able to view automatic changes to the contents of Panels 2 and 3. Consequently, one will be able to formulate visual subgraph queries effortlessly over different graph databases without requiring reconstruction of the visual interface.

Formulation of a visual query effortlessly. Given the data-driven construction of Panels 2 and 3, an audience can quickly and interactively formulate a large variety of queries by dragging and dropping canned patterns. Specifically, she may formulate the same query graph using the `PubChem` interface (Figure 1) and experience first-hand the tediousness in query construction due to the lack of availability of desired canned patterns to aid query formulation.

Acknowledgment

Sourav S Bhowmick was supported by the Singapore-MOE AcRF Tier-1 Grant RG24/12. Jinbo Zhang and Feida Zhu were supported by the NSF of China (No. 61170003) and Pinnacle Lab at SMU, respectively. Byron Choi was partially supported by the HKBU grant FRG2/12-13/079.

References

[1] D. H. Chau , C. Faloutsos, et al. GRAPHITE: A Visual Query System for Large Graphs. ICDM Workshop, 2008.
[2] H. He, A. K. Singh. Closure-tree: An Index Structure for Graph Queries. In ICDE, 2006.
[3] C. Jin, et al. Gblender: Visual Subgraph Query Formulation Meets Query Processing. In SIGMOD, 2011.
[4] C. Jin, et al. prague: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. In ICDE, 2012.
[5] H. Shang, et al. Connected Substructure Similarity Search. In SIGMOD, 2010.
[6] Y. Tian, et al. Efficient Aggregation for Graph Summarization. In SIGMOD, 2008.
[7] S. Yang et al. SLQ: A User-friendly Graph Querying System. In SIGMOD, 2014.
[8] X. Yan, et al. gSpan: Graph-based Substructure Pattern Mining. In ICDM, 2002.