AutoG: A Visual Query Autocompletion Framework for Graph Databases

Peipei Yi† Byron Choi† Sourav S Bhowmick§ Jianliang Xu†
†Department of Computer Science, Hong Kong Baptist University, Hong Kong
§School of Computer Science and Engineering, Nanyang Technological University, Singapore
{csppyi, bchoi,xujl}@comp.hkbu.edu.hk, assourav@ntu.edu.sg

ABSTRACT

Composing queries is evidently a tedious task. This is particularly true of graph queries as they are typically complex and prone to errors, compounded by the fact that graph schemas can be missing or too loose to be helpful for query formulation. Despite the great success of query formulation aids, in particular, *automatic query completion*, graph query autocompletion has received much less research attention. In this demonstration, we present a novel interactive visual subgraph query autocompletion framework called AUTOG which alleviates the potentially painstaking task of graph query formulation. Specifically, given a large collection of small or medium-sized graphs and a visual query fragment q formulated by a user, AUTOG returns top-k query suggestions Q' as output at interactive time. Users may choose a query from Q' and iteratively apply AUTOG to compose their queries. We demonstrate various features of AUTOG and its superior ability to generate high quality suggestions to aid visual subgraph query formulation.

1. INTRODUCTION

The prevalence of graph-structured data in modern real-world applications such as biological and chemical databases (*e.g.*, PUBCHEM), and co-purchase networks (*e.g.*, Amazon.com) has led to a rejuvenation of research on graph data management. Several database query languages have been proposed for textually querying graph databases (*e.g.,* SPARQL, Cypher). Unfortunately, formulating a graph query using any of these query languages often demands considerable cognitive effort and requires "programming" skill at least similar to programming in SQL. Yet, in a wide spectrum of graph applications users who need to query graph data are not proficient query writers. For example, chemists are not often expected to learn the complex syntax of a graph query language in order to formulate meaningful queries over a chemical compound database such as PUBCHEM[1] or *eMolecule*[2]. Hence, it is important to devise intuitive techniques that can alleviate the burden of query formulation and thus increase the usability of graph databases.

A popular approach to making query formulation user-friendly is to provide a visual query interface (GUI) for interactively constructing queries. In recent times, there has been increasing efforts to create such user-friendly GUIs from academia (*e.g.*, [3]) and industries (*e.g.,* PUBCHEM and *eMolecule*) to ease the burden on query formulation. During the construction (also refer to as composition) of visual queries, given a partially-composed query, it is always desirable to suggest top-k possible query fragments that the user may potentially add to his/her intermediate query in subsequent steps. Such suggestions can enhance user experience on graph databases and facilitate *exploratory search* [2], where non-expert users may learn, discover, and investigate information from a graph data source through a sequence of queries and answers.

In the literature, such suggestions that assist query formulation are often referred to as *query autocompletion*. Techniques for query autocompletion have been proposed for web search. Search engine companies use their proprietary algorithms for providing keyword suggestions during query formulation. A corresponding capability for graph query engine is in its infancy. In fact, to the best of our knowledge, except for a recent demo for edge suggestions [4], *the autocompletion of subgraph queries has not been studied before.*

There are two key challenges of autocompleting subgraph queries. Firstly, in web search, the natural logical increments (*i.e.*, tokens) of queries are keywords. However, the notion of "increments" of subgraph queries has not yet been defined. Furthermore, subgraph queries are structures and there are many ways to compose a query. Secondly, there can be potentially many candidate query suggestions. Consequently, it is paramount to return a small ranked list of query suggestions at interactive time.

In this demonstration, we present a novel autocompletion framework for subgraph queries called AUTOG. To tackle the first challenge, we propose a novel notion of logical increments of subgraph queries. We call them *c-prime features*, which are (frequent) subgraphs that can be constructed from small (frequent) subgraphs in *no more than c ways*. When possible increments are many, query autocompletion can be inefficient. The idea is to optimize query autocompletion time by omitting non-c-prime features because they may be formed from smaller features/queries anyway. To address the second challenge, we propose a novel *ranked subgraph query suggestion problem* (RSQ). The goal of the RSQ problem is to efficiently determine a candidate query suggestion set Q'. Specifically, AUTOG deploys a submodular *ranking function* that is in favor of query suggestions of high selectivities and structural diversity. A novel index for c-prime features, called *feature* DAG (FDAG) [9], is exploited to optimize RSQ. The key characteristics of FDAG are as follows: (i) it enumerates possible *query compositions* of c-prime feature pairs offline, (ii) it prunes redundant suggestions via graph automorphism, and (iii) it indexes some auxiliary structures

[1] https://pubchem.ncbi.nlm.nih.gov/
[2] https://www.emolecules.com/

Proceedings of the VLDB Endowment, Vol. 9, No. 13

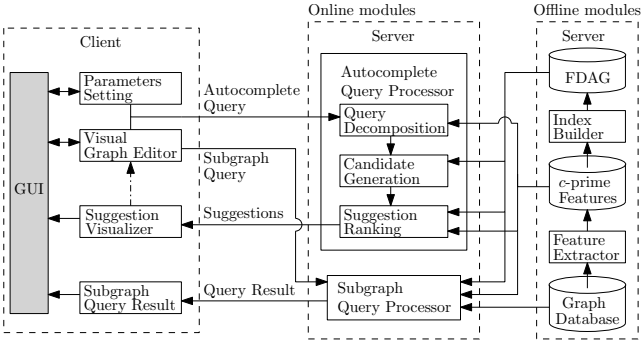

Figure 1: Architecture of AUTOG.

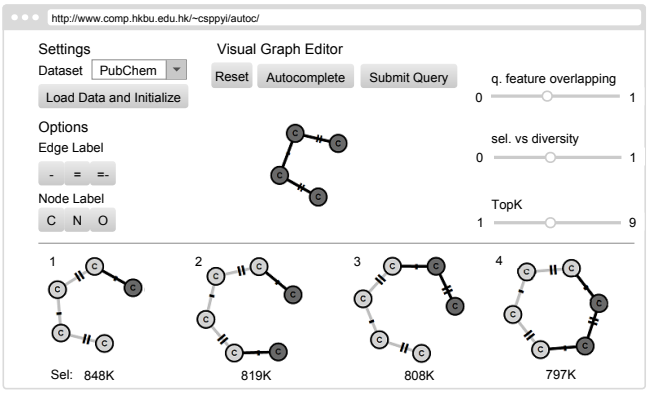

Figure 2: The AUTOG GUI (reorganized for clarity of printouts).

for computing structural similarity of suggestions online. To determine query suggestions efficiently, AUTOG leverages a greedy algorithm. In addition, it adopts a sampling approach to efficiently estimate the selectivity of query suggestions. Our demonstration focuses on these innovative features necessary to realize a query autocompletion framework for subgraph queries.

2. SYSTEM OVERVIEW

The system architecture of AUTOG is shown in Figure 1. It employs a client-server architecture and comprises of the following modules. The client-side mainly consists of an intuitive GUI whereas at the server side, the modules are organized into offline and online ones.

The GUI module at the client side. Figure 2 depicts the screenshot of the AUTOG visual interface. A user begins by choosing a target graph database using the left panel on which subgraph queries will be formulated visually. This panel also displays the unique labels of nodes and edges that appear in the dataset. Note that during the query formulation process, these labels are chosen for creating the nodes and edges in the query graph. The *Visual Graph Editor* panel in the middle is the area used to construct a query graph. *Users may click on the empty space of the editor to add a new node, and drag from one node close to another to add a new edge*. At any stage of the formulation process, one may retrieve query suggestions from the server by clicking on the *Autocomplete* button. Then, the bottom panel displays the relevant suggestions in real time. To highlight the incremental parts of the suggestions, the existing query is colored in light grey. To accept a suggestion, users may simply click on it. The right panel enables a user to tune relevant parameters related to the query suggestion process by interacting with the sliders. When the *Submit Query* button is clicked,

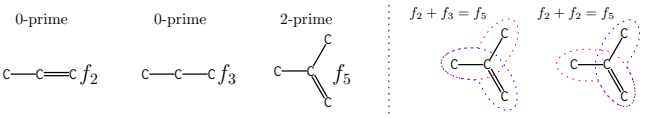

Figure 3: 0-prime feature f_2 and f_3 vs 2-prime feature f_5.

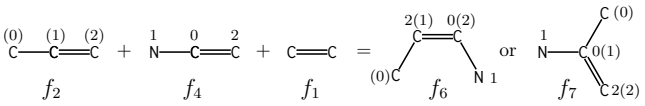

Figure 4: Example of different compositions of the same features.

the results of the current query graph are retrieved from the server and displayed on the GUI.

Offline modules at the server side. The offline component consists of modules that extract c-prime features from a graph database and build the FDAG index, that indexes these features, their compositions, and data graphs. While all indexes are built offline, we are able to build the index for PubChem data graphs in few days by using just a commodity machine. The *Feature Extractor* module extends an existing feature mining algorithm (*i.e.*, GSPAN [8]) with the algorithm to compute the *composability* of features. The output of this module is c-prime features of data graphs. The c-prime features are further indexed by our proposed index called Feature DAG index (FDAG). Consider Figure 3 as an example. Suppose $F = \{f_2, f_3, f_5\}$ is the set of frequent subgraphs of a graph database D. f_2 and f_3 are the smallest features in F. They cannot be constructed from other features and hence are 0-prime features. f_5 is a 2-prime feature because there are only two ways to construct f_5 from f_2 and f_3, as shown in the right hand side of the figure. The design rationales of c-prime features are that (i) some features are important to autocompletion because their absence leads to fewer possible suggestions; and (ii) some other features are less important because they can be constructed incrementally from small ones in numerous ways and can be suggested by query autocompletion anyway. We propose c-prime features – the first features being defined with their *composabilities*.

Observe that there are numerous possible ways to add query increments (c-prime features) to an intermediate user query. For example, consider f_2 in Figure 4 as an initial query (with the node IDs in parentheses). f_4 is a c-prime feature that is added to f_2 to form a new query via their common subgraph f_1. Figure 4 shows two possible queries (f_6 and f_7) that can be composed. Figure 5 shows that f_8 can be added to f_2 and there are many compositions that form the same query f_9. In our experiments, we mined 1.2K c-prime features from PUBCHEM, there are only 4.2M distinct non-empty subgraph queries that can be composed from the feature pairs.

As we shall see later, the *Candidate Generation* step in the online module generates candidates by determining possible compositions of features. Hence, FDAG indexes all possible compositions. Furthermore, Figure 5 highlights that structurally equivalent suggestions may be generated. Recall that incrementing automorphic f_8 to f_2 via different node configurations yields the same query. Similar argument can be applied to the current query and the common subgraph between the query and the increment. Hence, FDAG indexes automorphisms of c-prime features so that redundant candidates can be pruned.

Autocomplete Query Processor module at the server side. This module takes the current query graph and AUTOG's parameters as input and produces query suggestions as output at interactive time. First, it decomposes the query into a c-prime feature set with respect to the c-prime features F computed offline. Importantly, it

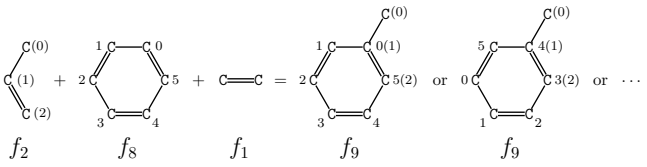

Figure 5: Example of redundant query compositions.

determines the *embeddings* (*i.e.*, the locations) of the features in the query, which show how they are connected and where query increments can be added to. Obviously, the decomposition of a query is not unique. Users may tune the *query decomposition* according to two competing objectives: firstly, the larger the features, the more structural semantics they preserve; secondly, the larger the features, the higher the chance the features overlap. However, overlapping features contain redundant information which affects the other parts of AUTOG.

It should be noted that queries may contain infrequent edges, which are not in F, and will not be handled by AUTOG. The analogy is that in web searches, infrequent keywords are not suggested; similarly, in AUTOG, infrequent logical units are not suggested. By definition, infrequent edges lead to small answer sets and consequently users may need less assistance from AUTOG.

The output of the above query decomposition step is a set of c-prime features of q and their embeddings in q. Next, in the *candidate generation* step, a set of candidate query suggestions are generated by utilizing these decompositions. A candidate query suggestion is formed by adding a query increment (which is a c-prime feature) to one of the c-prime features of q. Recall that query increments can be added to the current query in multiple ways. In the worst case, the number of possible suggestions being composed is exponential to the query and feature sizes.

In practice, some candidate queries may not make sense, as they may not retrieve any data graphs. We refer to them as *empty queries*. This step prunes empty queries. The unpruned queries are then returned as *candidate suggestions*. We elaborate on this pruning step further. Consider a graph $g = (V, E, \ell)$. Denote Σ to be the label set, where $\Sigma = \{\ell(v) \mid v \in V\}$. For each node $v \in V$, we determine a vector of the counts of its neighboring node's label $\vec{\ell_v}$, where

$$\vec{\ell_v}[l'] = |\{v' | (v, v') \in E, \ell(v') = l'\}|.$$

The nodes of the graphs can be represented by such vectors and hence, as data points in a Σ-dimensional space. Denote S to a skyline in the Σ-dimensional space of the data point representations of the nodes of the graphs. Given a query q, *q is non-empty if it does not contain a node whose vector dominates the nodes in S in some dimensions.*

To prune empty queries, each possible query is compared against the data points of the skyline, which is costly. Thus, we relax the check for efficiency. For each label l_1 in Σ, we determine the maximum number of each neighboring label l_2, d_{l_1,l_2} =

$$\mathsf{max}(|\{v_2 | \ell(v_1) = l_1, \ell(v_2) = l_2, (v_1, v_2) \in g.E, g \in D\}|).$$

The necessary condition for non-empty queries is then expressed in terms of d_{l_1,l_2}. A query q is non-empty *only if* $\nexists v_q \in q.V$, such that $\forall\ d_{l_q,l_2},\ d_{l_q,l_2} < |\{v_2 \mid \ell(v_q){=}l_q,\ \ell(v_2){=}l_2,\ (v_q,v_2) \in q.E\ \}|$, where $l_q \in \Sigma$.

Consider a small example database D that consists of g_1, g_2 and g_3, as shown in Figure 6. We illustrate that $d_{\mathtt{C},\mathtt{C}} = 2$, $d_{\mathtt{C},\mathtt{N}} = 1$, $d_{\mathtt{N},\mathtt{C}} = 1$ and $d_{\mathtt{C},\mathtt{O}} = 1$, etc. q_1' satisfies the necessary condition, and thus it is a candidate suggestion. In contrast, the C node in the center of q_2' has $d_{\mathtt{C},\mathtt{N}} = 2$, which violates the relaxed necessary condition. Thus, q_2 is pruned.

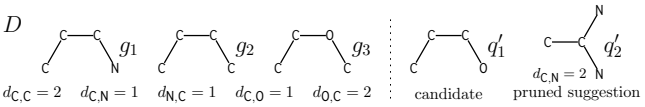

Figure 6: Example of pruning by the relaxed necessary condition.

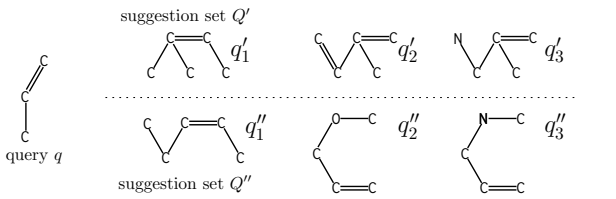

Figure 7: Similar suggestions Q' vs diverse suggestions Q''.

In the last step, *candidate suggestions are ranked* by the processor. To elaborate further, when users formulate their queries, they may have different needs for suggestions, under different query formulation scenarios. For example, expert users may use AUTOG to speed up their manual query formulation, whereas novice users may prefer diversified suggestions for exploring a database. We model the preferences between different criteria with a ranking function util and a user preference α. Moreover, since users may only be able to interpret a small subset of the candidate suggestions, AUTOG returns only top-k suggestions w.r.t. util and α. Here, we present a submodular util function, where greedy algorithms are its natural heuristics. It should be remarked that the ranking function util is for illustration purposes (*i.e.*, other function can be readily plugged into the AUTOG framework).

More specifically, we illustrate a ranking function for possibly novice users who prefer query suggestions that *(i) return more answer graphs and (ii) are structurally diversified*. These two preferences can be quantified by the following two functions:

- $\mathsf{sel}(q)$: the selectivity of q on D.
- $\mathsf{dist}(q_i, q_j)$: the "intra-dis-similarity" between a pair of suggestions, q_i and q_j. The total pairwise distance of suggestions reflects how diversified a set of suggestions is. For illustration purposes, we adopt the *maximum common edge subgraph* (mces) for dist. mces is adopted because adding edges (as opposed to nodes) to an existing query is an important logical step of composing queries.

Figure 7 shows two sets of example suggestions Q' and Q'' to the same query q. All suggestions add two edges to q. Clearly, Q'' is more diverse because its suggestions differ from each other by two edges, whereas those in Q' differ from each other by an edge.

Given a set of suggestions Q':$\{q_1', q_2', \ldots, q_k'\}$ and a user preference α, the *user intent value* of Q' (util) is defined as follows:

$$\mathsf{util}(Q') = \frac{\alpha}{k} \sum_{q' \in Q'} \mathsf{sel}(q') + \frac{1-\alpha}{k(k-1)} \sum_{q_i', q_j' \in Q', i \neq j} \mathsf{dist}(q_i', q_j'),$$

where $\alpha \in [0, 1]$.

The two objectives of util can be competing. It can be observed that in practice, the sel of smaller queries are often larger as more data graphs contain smaller queries; in contrast, smaller queries may have smaller structural differences between them and consequently, dist returns smaller values and their diversities are relatively low. Determining the top-k suggestions that have the optimum util value (called the RSQ problem) is an NP-hard problem. A greedy ranking algorithm is implemented to strike a balance between suggestion quality and efficiency.

Table 1: Query sizes vs quality metrics (PUBCHEM)

$\|q\|$	#AUTOG	TPM (%)
8	2.2	45%
12	3.3	44%
16	4.0	42%

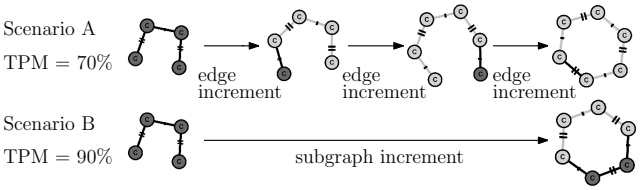

Figure 8: Illustration of edge and subgraph suggestion scenarios.

We have evaluated the qualities of the suggestions of random queries via user tests and large scale simulations [9]. We report some results in Table 1, where #AUTOG is the average number of suggestions accepted in the simulation and TPM is the *total profit metric (TPM)* adopted from [7], which quantifies the % of mouse clicks saved by AUTOG in visual graph query formulation: TPM $= \frac{\text{number of clicks saved by AUTOG}}{\text{number of clicks without AUTOG}}$.

We highlight that i) the suggestions were accepted multiple times during query formulation, and ii) AUTOG saved roughly 40% of users' mouse clicks in query formulation. We also carried out a user test with 10 volunteers to formulate 60 diverse queries [9]. The questionnaire can be found at `http://goo.gl/dFRdwj`. The results showed that TPM is a good indication of quality because the correlation coefficient of users' ratings and TPM is 0.96 and the p-value is 0.002. In addition, we observe that AUTOG returned suggestions shortly under a large variety of parameter settings.

3. RELATED SYSTEMS & NOVELTY

Mottin et al. [6] recently studied the problem of graph query reformulation. The reformulated queries maximally cover the results of the current query. Their approach assumes that all query results are relevant. When queries are small, the number of answers can be huge (*e.g.*, 30K graphs on average for small queries of size 8 over PUBCHEM). In practice, users may not be interested in all of them. In contrast, AUTOG ranks suggestions based on users' preferences such as selectivities and structural diversities.

More germane to this demonstration is the recent work in [4], which suggests *edge increments* to the current query graph. However, query formulation may then take many steps; and users can only express limited structural information in each step. In contrast, AUTOG suggests *subgraph increments*. Scenario A of Figure 8 shows the suggestions of AUTOG when edge increments are enforced, whereas Scenario B shows that AUTOG can suggest a larger subgraph to form the same query graph. As can be seen, AUTOG is more effective and user-friendly.

Exploratory search has known to be useful for enhancing interactions between users and search systems (*e.g.*, [5]). Graph query autocompletion is consistent to exploratory search as it allows users to construct their queries incrementally and interactively, and explore the intermediate query results.

4. DEMONSTRATION OVERVIEW

AUTOG was implemented in C++, using VF2 for subgraph test and the McGregor's algorithm (with minor adaptation) for determining mces. We adopted the GSPAN implementation from [8] for frequent subgraph mining. We will use two popular benchmarked real datasets, namely PUBCHEM (1 million graphs) and AIDS (10,000 graphs), for demonstration. The default dataset used in the live demonstration is PUBCHEM. We will use query sets provided by IGRAPH [1], and follow their default settings. A set of sample target queries will be presented (some are shown in Figure 9). Attendees may also formulate their own ad-hoc queries. A video of AUTOG can be found at `http://goo.gl/4KnJeq`.

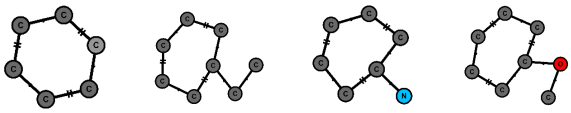

Figure 9: Example query graphs.

The key objective of the demonstration is to enable the attendees to interactively experience the following features through the AUTOG GUI (as shown in Figure 2).

Interactive experience of autocompletion during query formulation. Through the GUI (Figure 2), one will be able to select the relevant data source, visually formulate the subgraph query she would like to construct (some examples are shown in Figure 9), and view the suggestions generated by AUTOG during the construction process. Specifically, she will be able to automatically generate a subgraph query (especially large ones) with significantly fewer clicks without constructing each edge manually. She will also be able to experience the query suggestion time and the quality of top-k suggestions generated by AUTOG.

Interactive experience of the effect of different parameters. By varying α, users may tune their preference of selectivity over suggestion diversity. Moreover, AUTOG can be used by both expert users (for reducing query formulation steps) and novice users (for returning structurally diverse suggestions). By varying c, users may set their preference between quick and comprehensive suggestions. During the demonstration, an attendee will also be able to modify various parameters associated with the query suggestion generation process through the right panel of the GUI and interactively experience their impact on the query suggestions in the bottom panel.

5. REFERENCES

[1] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, pages 449–459, 2010.

[2] M. Herschel, Y. Tzitzikas, K. S. Candan, and A. Marian. Exploratory search: New name for an old hat? `http://wp.sigmod.org/?p=1183`, 2014.

[3] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou. QUBLE: blending visual subgraph query formulation with query processing on large networks. In *SIGMOD*, pages 1097–1100, 2013.

[4] N. Jayaram, S. Goyal, and C. Li. VIIQ: Auto-suggestion enabled visual interface for interactive graph query formulation. *PVLDB*, pages 1940–1951, 2015.

[5] G. Marchionini. Exploratory search: from finding to understanding. *Commun. ACM*, pages 41–46, 2006.

[6] D. Mottin, F. Bonchi, and F. Gullo. Graph query reformulation with diversity. In *KDD*, pages 825–834, 2015.

[7] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, pages 219–230, 2007.

[8] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[9] P. Yi, B. Choi, J. Xu, and S. S. Bhowmick. AutoG: A visual query autocompletion framework for graph databases. *http://goo.gl/4KnJeq*, 2016.