The VLDB Journal (2017) 26:347–372
DOI 10.1007/s00778-017-0454-9

REGULAR PAPER

AutoG: a visual query autocompletion framework for graph databases

Peipei Yi[1] · **Byron Choi**[1] · **Sourav S. Bhowmick**[2] · **Jianliang Xu**[1]

Received: 11 March 2016 / Revised: 25 November 2016 / Accepted: 6 January 2017 / Published online: 27 January 2017

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 paper, we propose a novel framework for subgraph query autocompletion (called AUTOG). Given an initial query q and a user's preference as input, AUTOG returns ranked query suggestions Q' as output. Users may choose a query from Q' and iteratively apply AUTOG to compose their queries. The novelties of AUTOG are as follows: First, we formalize query composition. Second, we propose to increment a query with the logical units called *c-prime features* that are (i) frequent subgraphs and (ii) constructed from smaller c-prime features in no more than c ways. Third, we propose algorithms to rank candidate suggestions. Fourth, we propose a novel index called *feature* DAG (FDAG) to optimize the ranking. We study the query suggestion quality with simulations and real users and conduct an extensive performance evaluation. The results show that the query suggestions are useful (saved roughly 40% of users' mouse clicks), and AUTOG returns suggestions shortly under a large variety of parameter settings.

Keywords Subgraph query · Query autocompletion · Graphs · Database usability

✉ Peipei Yi
csppyi@comp.hkbu.edu.hk

Byron Choi
bchoi@comp.hkbu.edu.hk

Sourav S. Bhowmick
assourav@ntu.edu.sg

Jianliang Xu
xujl@comp.hkbu.edu.hk

[1] Hong Kong Baptist University, Kowloon, Hong Kong

[2] Nanyang Technological University, Singapore, Singapore

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 lead to a rejuvenation of research on graph data management and analytics. Several novel graph data management platforms have emerged from academia, industrial research laboratories and start-up companies. Several database query languages have been proposed for textually querying graph databases (*e.g.,* SPARQL and 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 consumers need to query graph data but 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 make query formulation user-friendly is to provide a visual query interface (GUI) for

[1] https://pubchem.ncbi.nlm.nih.gov/.

[2] https://www.emolecules.com/.

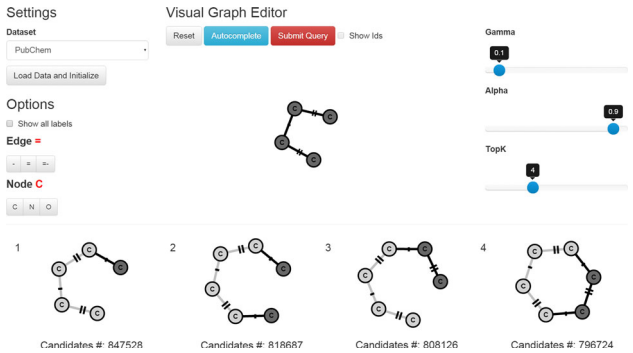

Fig. 1 Suggestions during visual subgraph query formulation

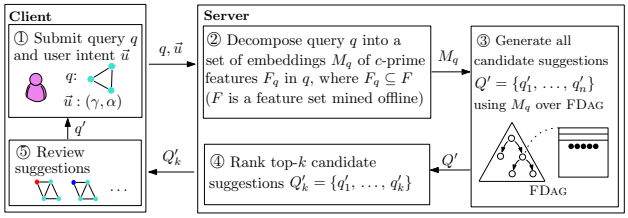

Fig. 2 Autocompletion framework for subgraph queries

interactively constructing queries. In recent times, there has been increasing efforts to create such user-friendly GUIs from academia [18] and industry (*e.g., PubChem* and *eMolecule*) to ease the burden of query formulation. Given a partially constructed visual subgraph query, it is always desirable to suggest top-k possible query fragments that the user may potentially add to his/her intermediate query in the subsequent steps. Such suggestions can enhance user experience on graph databases and facilitate *exploratory search* [17], where non-expert user may learn, discover and investigate information from a graph data source through a sequence of queries and answers.

Example 1 Consider the visual subgraph query interface in Fig. 1 for querying PUBCHEM. Suppose Mike wishes to search for compounds containing the chlorobenzene substructure. The partial subgraph query constructed by him is depicted in the *Visual Graph Editor* panel. It will be indeed helpful to Mike if the query system can suggest top-k possible query fragments (subgraphs) that he may add to his query in the next step.[3] An example of such top-4 suggestions is shown at the bottom panel. Observe that each suggestion is composed by adding small increments to the query graph in the middle panel (indicated in gray). Mike may select the fourth suggestion by clicking on it, thus saving his mouse clicks to manually formulate the new nodes and edges. He may then continue formulating the final query graph in subsequent steps by leveraging the query suggestion capability iteratively. A further animated example and a prototype system can be found at https://goo.gl/Xr9MRY. □

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 and XML search [14]. For instance, search engine companies use their proprietary algorithms for providing keyword suggestions during query formulation. However, 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 [19], 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, not a sequence of tokens. That is, there are many ways to compose the queries. Secondly, there can be potentially many candidate query suggestions. Consequently, it is paramount to return a ranked list of query suggestions at interactive time.

To address the aforementioned challenges, we propose a novel autocompletion framework for subgraph queries (namely AUTOG [44]), whose simplified workflow is shown in Fig. 2. In a nutshell, AUTOG allows users to submit an initial query q and a preference $\vec{u}$ and it returns ranked suggestions according to $\vec{u}$.

To tackle the first challenge, we propose a novel notion of c-prime *features* as logical increments of subgraph queries. In this paper, we illustrate c-prime features with *frequent subgraphs, but they can be any structural features that capture the structural characteristics of data graphs* (e.g., [10,13,42]) that users may be interested in. c-prime features are the first structural features defined with *feature composability*—the number of ways that a feature can be composed from other small features. In short, a c-prime feature is a feature whose composability is no more than c. When possible increments are many, query autocompletion can be inefficient. Our main idea is that to optimize query autocompletion time, AUTOG omits non-c-prime features because they may be formed from c-prime features/queries anyway. As shown in Fig. 2, in our proposed framework, a user submits an initial query q and his/her intent on ranking query suggestions. Then, the query graph is represented by c-prime features.

To overcome the second challenge mentioned above, we formalize the query autocompletion problem as a novel *ranked subgraph query suggestion problem* (RSQ). The goal of RSQ problem is to efficiently determine a candidate query

[3] Ideally, the user interface may automatically show useful suggestions to users. The current GUI (Fig. 1) provides an "Autocomplete" button for users to fetch the top-k suggestions to allow an explicit comparison of the experiences with and without suggestions for user tests.

suggestion set Q', where each suggestion specifies how a c-prime feature is added to the current query. It should be noted that *queries, as opposed to data graphs, are being ranked*. We prove that RSQ is an NP-hard problem. To optimize RSQ, we propose a novel index for c-prime features, called *feature* DAG (FDAG). The key features of FDAG are as follows: (i) it provides efficient support for determining the subgraphs or supergraphs of indexed features and their embeddings, (ii) it prunes redundant suggestions via graph automorphism, (iii) it enumerates possible query compositions of c-prime feature pairs offline, and (iv) it indexes some auxiliary structures for computing structural dissimilarity of suggestions online. To this end, we propose a ranking function that is in favor of query suggestions of high selectivities and structural diversity. It should be noted that *our framework is not tightly coupled with this ranking strategy. Interested users may plug into* AUTOG *their ranking functions that fit their potential applications as well*. To rank query suggestions efficiently, we propose a two-level greedy algorithm. In addition, we propose a necessary condition for non-empty suggestions that is used for pruning useless suggestions. We adopt a sampling approach to efficiently estimate the selectivity of query suggestions. We propose an algorithm to efficiently compute structural differences of suggestions.

In summary, the key contributions of this paper are as follows:

1. We propose c-prime features as logical units for autocompletion for subgraph queries.
2. We propose an autocompletion framework that takes a user's current query and preference as input and produces ranked query suggestions as output.
3. We propose FDAG to efficiently optimize the generation and ranking of query suggestions.
4. We study the query suggestion quality with simulations and real users and conduct an extensive experimental study on performance with both real and synthetic datasets. The result verifies the usefulness of the suggestions and the efficiency and effectiveness of our autocompletion framework.

The rest of the paper is organized as follows. Section 2 introduces preliminary concepts and formally states the autocompletion problem for subgraph queries. Section 3 proposes the c-prime features. The autocompletion framework is presented in Sect. 4. Section 5 presents an indexed approach to autocompletion. Section 6 proposes an automorphism-based pruning technique. Experimental study is reported in Sect. 7. Related work is discussed in Sect. 8. Section 9 concludes this paper. "Appendices" contain all proofs, additional experiments on performances and technical details of index construction.

2 Preliminaries

In this section, we first provide the preliminaries and describe the problem being studied. Then, we elaborate the query composition operation (or simply *composition*) assumed by this paper which combines two graphs to form another graph.

2.1 Subgraph queries and background

We consider a graph database D as a set of data graphs $\{g_1, g_2, \ldots, g_n\}$. Each graph is a 3-ary tuple $g = (V, E, l)$, where V and E are the vertex and edge sets of g, respectively, and l is the label function of g. The size of a graph is defined by $|E|$. The query formalism adopted by this paper is subgraph isomorphism, defined as follows.

Definition 1 *(Subgraph isomorphism)* Given two graphs $g = (V, E, l)$ and $g' = (V', E', l')$, g is a *subgraph* of g', denoted as $g \subseteq_\lambda g'$, *iff* there is an injective (or *embedding*) function $\lambda : V \mapsto V'$ such that

1. $\forall u \in V, \lambda(u) \in V'$ such that $l(u) = l'(\lambda(u))$; and
2. $\forall (u, v) \in E, (\lambda(u), \lambda(v)) \in E'$ and $l(u, v) = l'(\lambda(u), \lambda(v))$. □

Definition 2 (*Subgraph query*) Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a query graph q, the answer set of q is $D_q = \{g | q \subseteq_\lambda g, g \in D\}$. □

Multiple subgraph isomorphic embeddings of g may exist in g', denoted as $\lambda^0_{g,g'}, \lambda^1_{g,g'}, \ldots, \lambda^m_{g,g'}$. Therefore, subgraph isomorphism of g and g' can be viewed as a relation between g and g', where each record is an embedding of g in g'. Furthermore, in our technical discussions, we may use *graph isomorphism* of g_i and g_j, which is $g_i \subseteq_\lambda g_j$ and $|g_i.V| = |g_j.V|$, and *graph automorphism* of g, which is a graph isomorphism to itself. To keep the presentation intuitive, we may describe automorphism/graph isomorphism as a mapping of a node configuration (domain) to another node configuration (image).

2.2 Query composition

This subsection formalizes the query composition used in our technical discussions.[4] We recall that queries are complex structures, and larger queries may be constructed from smaller queries in many ways. To facilitate the discussions, we define how a large query is constructed from two smaller queries by specifying how a common subgraph connects them.

[4] Query composition refers to an intuitive step when users are composing their queries, which is obviously different from the compositions used in the functional programming literature.

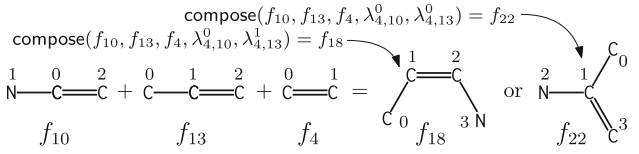

Fig. 3 Query graph composition

Definition 3 (*Common subgraphs* (CS)) Given two graphs g_1 and g_2, a *common subgraph of* g_1 *and* g_2 is a connected subgraph containing at least one edge and it is a subgraph of g_1 and g_2 (denoted as $\mathsf{cs}(g_1, g_2)$, or simply cs when g_1 and g_2 are clear from the context), *i.e.*, $\mathsf{cs} \subseteq_{\lambda_1} g_1$ and $\mathsf{cs} \subseteq_{\lambda_2} g_2$. We define $\mathsf{CS}(g_1, g_2)$ to be the set of common subgraphs of g_1 and g_2. □

A subtlety is that in the literature, *maximal* common subgraphs are extensively studied. However, we present *common subgraphs* because in query composition, large query graphs may not necessarily be formed via the maximal common subgraphs of small graphs.

Definition 4 *(Query composition)* compose is a function that takes two graphs, g_1 and g_2, and the corresponding embeddings, λ_1 and λ_2, of a common subgraph cs as input, returns the graph g that is composed by g_1 and g_2 via λ_1 and λ_2 of cs, respectively, denoted as $g = \mathsf{compose}(g_1, g_2, \mathsf{cs}, \lambda_1, \lambda_2)$. □

Example 2 Figure 3 shows an example of query composition by using PUBCHEM. The chemical elements are vertices and the numbers near them are their IDs. f_4 is a common subgraph of f_{10} and f_{13}. We have $f_4 \subseteq_\lambda f_{10}$ where $\lambda^0_{4,10} = \{0 \mapsto 0, 1 \mapsto 2\}$ or $\lambda^1_{4,10} = \{0 \mapsto 2, 1 \mapsto 0\}$, $f_4 \subseteq_\lambda f_{13}$ where $\lambda^0_{4,13} = \{0 \mapsto 1, 1 \mapsto 2\}$ or $\lambda^1_{4,13} = \{0 \mapsto 2, 1 \mapsto 1\}$. There are four ways to compose f_{10} and f_{13} via f_4:

1. $\mathsf{compose}(f_{10}, f_{13}, f_4, \lambda^0_{4,10}, \lambda^0_{4,13})$ results in f_{22};
2. $\mathsf{compose}(f_{10}, f_{13}, f_4, \lambda^0_{4,10}, \lambda^1_{4,13})$ results in f_{18};
3. $\mathsf{compose}(f_{10}, f_{13}, f_4, \lambda^1_{4,10}, \lambda^0_{4,13})$ results in f_{18}; and
4. $\mathsf{compose}(f_{10}, f_{13}, f_4, \lambda^1_{4,10}, \lambda^1_{4,13})$ results in f_{22}. □

2.3 Query composition modes

In this subsection, we present the possible query composition modes supported by AUTOG and the one that is assumed by our technical presentations.

Connected versus disconnected subgraphs In this paper, we assume that both the query graphs and the query suggestions (formed by a composition of the query graphs and increments) are connected graphs. Hence, the common subgraphs are connected graphs, too. In the case of disconnected query graphs, we may simply pass them *individually* to AUTOG. Similarly, the query suggestions presented in this paper are connected. A minor relaxation on the requirement of connected existing queries and increments will support disconnected query suggestions.

Edge increments versus subgraph increments A simple query autocompletion mode is to generate edge increments to the current query (e.g., [19]). However, edge suggestions have at least two drawbacks: (1) the query formulation process may take many steps, and (2) users can express limited structural information regarding their desired queries in each step. On the other hand, as shown in Sect. 1, we propose to increment the query through our proposed features, which can be subgraphs. In our experimental investigation on the query suggestions, the average number of edges of query increments is always more than one.

For simplicity of presentation, *we assume connected subgraphs suggestions* unless otherwise specified. Putting these together, the problem being studied can be described as follows.

Problem statement Given an existing query q, a ranking function util, a user preference $\vec{u}$ and a parameter k, compute a query suggestion set Q'_k: $\{q'_1, q'_2, \ldots, q'_k\}$ s.t. for $i \in [1, k]$, q'_i is composed by adding an increment to q and Q'_k is the top-k suggestions w.r.t. the ranking function util and the user preference $\vec{u}$. □

3 c-Prime features

Structural features of graphs have been extensively studied recently, for the purpose of optimizing the performances of structural queries, among other things. Their intuition is to determine a set of subgraphs that carry various structural characteristics of data graphs D (e.g., discriminative frequent subgraph [42] and action-aware frequent subgraph [21]). Data graphs are then indexed by the features. Given a query q, it is decomposed into a set of features F_q. A data graph that does not contain F_q cannot be an answer of q and hence can be pruned. Previous work shows that this approach can effectively prune non-answers. However, this approach does not consider *query autocompletion*.

In this section, we propose c-prime features. They are defined by how many ways they can be composed from smaller features. Intuitively, c-prime features are features that can be formed by smaller features in only a few ways. c-prime features have not been proposed before because they are designed for suggesting *query increments not filtering data graphs*. c-prime features are orthogonal to existing features, i.e., users may integrate their existing features with c-prime features for their specific applications.

The design rationales of c-prime features are that (i) some features are important to query autocompletion because their

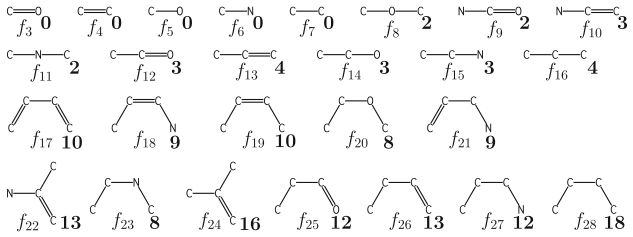

Fig. 4 Frequent features (partial) with their composabilities

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.

To discuss c-prime features, we start with *frequent features* [41]. Frequent features are adopted because, without a prior knowledge, we may assume each data graph in the database D has the same chance of being retrieved by users' queries. Hence, frequent subgraphs of D have higher chances that appear in users' queries.

We present the formal definition of frequent features in Definition 12 in "Appendix 1". We provide an example of frequent features below.

Example 3 Figure 4 shows a set of frequent features, extracted from PUBCHEM by GSPAN[41]. In the figure, the vertices C, N and O represent the chemical elements carbon, nitrogen and oxygen, respectively. The edges between the chemicals (i.e., C–C and C=C) signify the single and double bonds between two elements. □

Given a graph g and a frequent feature set F of a database D, we may decompose g into a set of features F_g: $\{f_1, ..., f_n\}$, where $\forall f_i \in F_g$ implies $f_i \in F$ and $f_i \subseteq_\lambda g$. Similarly, we may decompose a query into a set of features and their embeddings.

Definition 5 A query q of AUTOG is represented as a binary tuple (F_q, λ), where F_q is a set of features of q and λ takes a feature $f_q \in F_q$ as input and returns the embedding of f_q in q. □

Example 4 Suppose f_{18} is a query. A possible F_q is $\{f_4, f_6, f_7, f_{10}, f_{13}, f_{18}\}$. One may easily derive embeddings of the double bond C=C (f_4), the single bond C–C (f_7) and other features in f_{18}. □

From the example, we can see that queries can be considered as compositions of features.[5] However, how features are composed together to form queries requires some elaboration. Graph query composition here is structural.[6] Feature embeddings are required to specify how large structures are formed (see Definition 4). In Definition 6, we define feature composability as a measurement of the number of embeddings of feature pair compositions that form the feature f.

Definition 6 *(Feature composability)* The composability of a frequent feature f with respect to the feature set F, denoted as $\mathsf{c}(f,F)$, is

$$\sum \frac{|\{(f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j) \mid f = \mathsf{compose}(f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j)\}|}{|A(\mathsf{cs})|},$$

where $\mathsf{cs} \in \mathsf{CS}(f_i, f_j)$ and $f_i, f_j \in F$, the equality "=" denotes graph isomorphism and $A(\mathsf{cs})$ denotes the automorphism relation of cs. □

In Definition 6, the numerator of the composablity is the number of distinct feature embeddings that form the feature. The denominator $|A(\mathsf{cs})|$ is needed in Definition 6 because the queries that are constructed from features via automorphic common subgraphs are structurally equivalent.

Given this background, we are ready to propose c-prime features, which are features that have a composability smaller than or equal to c.

Definition 7 (c-prime *feature*) A feature f is a c-prime feature if and only if $\mathsf{c}(f, F) \leq c$. The feature is non-c-prime feature if and only if it is not a c-prime feature. □

Assuming that each composition is equally likely, non-c-prime features have a high chance of being formed from c-prime features. Therefore, non-c-prime features have a higher chance of being recovered from query autocompletion. On the other hand, c-prime features may not be suggested as they may not be composed from other features. Ignoring c-prime features leads to less comprehensive query suggestions.

Example 5 Consider the features shown in Fig. 4 again. We annotate each feature with its composability at its lower right-hand side. When the value of c was 4, the c-prime features (4-prime features) are $\{f_3, f_4, ..., f_{16}\}$. While larger features may still be constructed from 4-prime features, they may require multiple construction steps. When c is set to 16, only f_{28} is not c-prime feature. It is because f_{28} can be constructed from f_7 and f_{16} in many ways. In the case where the dataset has many features and their possible compositions are voluminous, query autocompletion may become too costly. Hence, the non-16-prime features may be omitted from query autocompletion as these features are more probably suggested. In other words, users may not lose query suggestion candidates even omitting non-16-prime features. □

In addition, c-prime features have the properties of anti-monotonicity and downward-closure properties, which are

[5] We assume the non-feature parts of the queries are inputted by users. That is, they are not composed from AUTOG.

[6] In contrast, for keyword search (of strings), key phrases are simply composed by a *union/concatenation* of keywords.

similar to those of frequent features. The details are provided in "Appendix 1".

Example 6 Consider Fig. 4 again. f_{13} can be constructed from f_4 and f_{13} via f_4. The cs is f_4. The cs has two embeddings in f_{13} and f_4, respectively. $A(\text{cs})$ is 2. The same counts are obtained from composing f_7 and f_{13} via f_7. The composability of f_{13} is 2×2 / 2 + 2×2 /2 equals 4. Thus, f_{13} is a 4-prime feature. f_{13} is a subgraph of f_{18}. After some counting, we note that f_{18} is a 9-prime feature. By the property of anti-monotonicity, f_{13} is certainly a 9-prime feature. Since f_{18} is not a 4-prime feature, by the downward-closure property, any supergraphs of f_{18} are not 4-prime features. □

Integration of c-prime features with other features We remark that this section illustrates c-prime features with frequent features (as their underlying features). However, depending on users' applications, AUTOG may integrate other features into c-prime features. For instance, the structure search of *PubChem*[7] provides *some templates of query structures*[8] for users to compose their queries, based on domain knowledge of chemical applications. In other applications such as web searches [32], the query templates can be automatically derived from query logs by using machine learning or data mining methods. *When* AUTOG *adopts such approaches, it adds the templates into the feature set of* D. AUTOG simply determines whether they are c-prime as before.

4 Autocompletion framework for subgraph queries—AutoG

This section presents the major steps in AUTOG, namely (i) decomposing users' queries, (ii) determining the candidate query suggestions and (iii) ranking the suggestions, with respect to users' preference. An overview of the framework is shown in Fig. 2.

4.1 Query decomposition

Following the subgraph query processing method in the literature (e.g., [10,42]), AUTOG assumes the (c-prime) features of data graphs are mined offline. At runtime, it decomposes a query into a feature set. However, AUTOG requires the embeddings (intuitively, the locations) of the features in the query which show how they are connected.

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

[8] Almost all query templates of *PubChem* on their user interface are also frequent subgraphs, returned by gSpan using its default parameters. This shows that the domain experts indeed set frequent subgraphs as query templates. Yet, we omit the advanced templates of *PubChem* after user's click on the UI.

Algorithm 1 Query Decomposition

Input: a query q, feature set F (determined by gSpan [41] offline) and user preference component γ
Output: a set of embeddings of the c-prime features M_q in q
1: Let F_q be the c-prime features of q
2: Let M_{F_q} be the embeddings of each $f \in F_q$ in q // e.g., using VF2
3: Let E be the edge set of q, and M_q be an empty set
4: Initialize $e.w = 1$ for $\forall e \in E$, and $m_f.unused = \texttt{true}$ for $\forall m_f \in M_{F_q}$
5: **while** $m_f = \text{FIND}(E, M_{F_q})$ **do**
6: $M_q \leftarrow M_q \cup m_f$
7: **return** M_q
8: **function** FIND(E, M_{F_q})
9: determine $m_f \in M_{F_q}$, where $m_f.\texttt{unused} = \texttt{true}$,
10: m_f covers at least one uncovered edge
11: $\nexists m'_f \in M_{F_q}$ and UTIL(m'_f) > UTIL(m_f)
12: $e.w \leftarrow e.w * \gamma$ for $\forall e \in m_f.E$
13: $m_f.\texttt{unused} \leftarrow \texttt{false}$
14: **return** m_f
15: **function** UTIL(m_f)
16: **return** $\sum_{e \in m_f.E} e.w$

It is evident that the decomposition of a query is not unique. Further, the design rationales of query decomposition in AUTOG are competing ones: Firstly, the larger the features, the more structural semantics they preserve; secondly, the larger the features, the higher the chance the features overlap. The overlapping features contain redundant information and hence should be avoided. Therefore, a user-specified parameter γ is introduced to specify the desirable degree of overlapping features. The larger the value of γ, the more likely the overlapping of decomposed features.

Example 7 Consider the graphs shown in Fig. 3. Suppose that f_{18} is a user query. A possible decomposition is $\{f_{10}$ $f_{13}\}$, of which the embeddings are overlapping. □

Algorithm 1 is a greedy algorithm for decomposing a query q with respect to the user-specified parameter γ. Initially, all the edges of q are not covered. Algorithm 1 iteratively determines an embedding of a c-prime feature to cover q until no more uncovered edges can be found. The output is a set of embeddings of the c-prime features M_q in q.

More specifically, first, we determine the c-prime features F_q of q by invoking a feature extraction program [41]. We determine the embeddings M_{F_q} (called feature-query embeddings) of the extracted features by invoking VF2 (Line 2). We initialize the result feature-query embeddings M_q to be an empty set and E to be the edge set of q to be covered (Line 3). We assume each edge e of q has a weight w, which denotes if e has been covered by some features in M_q. Each feature-query embedding m_f has a flag (called unused) to indicate whether the feature-query embeddings is in M_q. Second, in Lines 5–6, we iteratively add the next m_f with

the largest utility (defined by UTIL) to M_q. The weights of the edges of the added m_f are degraded by γ (Line 12).

We make two remarks on FIND (Lines 8–14). In general, there are multiple decompositions of a query. FIND enumerates larger features, where the feature size is determined by the sum of edge weights (implemented by UTIL in Lines 15–16). Larger features are used because they require more human effort to compose, i.e., they may preserve more of the user's intention. Hence, when a user has drawn exactly a large feature in his/her initial query, FIND leads AUTOG to consider it as a whole, as opposed to small feature(s). However, large features may overlap. Thus, if an edge is covered by an m_f and added to M_q, the weight of the covered edge is reduced by a factor of γ (Line 12).

It should be remarked 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. In this case, users may need less assistance from AUTOG.

4.1.1 Complexity analysis of query decomposition

The time complexity of Algorithm 1 is $O(|F_q| \times T_{subiso} + |E|^2 \times |M_{F_q}|)$, where (a) the first term is the time for determining the embeddings of F_q in q and T_{subiso} is the time for a subgraph isomorphism call, and (b) the second term is for scanning the $|M_{F_q}|$ embeddings to cover $O(|E|)$ edges in the FIND function, which is invoked $O(|E|)$ times.

We provide several observations on the two terms in the above-mentioned complexity. A worst-case exponential-time subgraph isomorphism algorithm is needed to determine the embeddings of F_q in q (Lines 1–2). In this paper, we use a practical subgraph isomorphism algorithm called VF2. Moreover, users typically draw small queries via a visual graph editor, e.g., a graph containing fewer than 24 edges. Hence, the size and the number of features of the query (F_q) are small. As a result, T_{subiso} is small in practice, and VF2 is invoked only few times. Regarding the complexity for scanning the embeddings, the terms $|E|$ and $|M_{F_q}|$ are small, again, due to small query sizes. Finally, the calculations of Lines 5–16 do not incur large constants in the asymptotic complexity. Hence, Algorithm 1 decomposes queries efficiently.

4.2 Generation of candidate suggestions

After the query decomposition step, the query q is represented by a set of c-prime features and their embeddings in q. The next step is to generate candidate query suggestions. In this subsection, we present connected feature increments, which are the most technically intriguing query composition mode, as discussed in Sect. 2.3.

Query increments can be added to the current query in multiple ways. Specifically, given a set of c-prime features, the number of compositions is, in the worst case, exponential to the query and feature sizes. However, in practice, many possible composed queries may not make sense that do not retrieve any data graphs. Such queries are also known as *empty queries*. Further, it is known that deciding the emptiness of a subgraph query is NP-hard.

This subsection formalizes a *necessary* condition for non-empty query compositions. We illustrate how to efficiently prune empty queries using the necessary condition and *the unpruned queries are considered candidate suggestions*.

4.2a Baseline We present the condition with node labels for presentation simplicity, which can be readily extended to support edge labels. 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, *if it is non-empty, q does not contain a node whose vector dominates the nodes in S in some dimensions.* This condition is formalized in Proposition 1. Its proof can be established by a proof by contradiction.

Proposition 1 *A query q is a* non-empty query *(also refers to candidate suggestion) only if $\nexists v_q \in q.V$ such that $\forall g \in D$, $\forall v \in g.V$, $\exists l \in \Sigma$, $l = \ell(v) = \ell(v_q)$, and $\vec{\ell}_v[l] < \vec{\ell}_{v_q}[l]$.* □

Suppose that there are $|S|$ data points on the skyline. The check of Proposition 1 requires $O(|q.V| \times |\Sigma| \times |S|)$ comparisons. As discussed, numerous possible queries may be generated and *they are checked with S at runtime*. Thus, we relax the check for efficiency.

4.2b Relaxed necessary condition For each label l_1 in Σ, we determine the maximum number of each neighboring label l_2 of D,

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

The necessary condition for non-empty queries is then expressed in terms of d_{l_1,l_2}, as shown in Proposition 2. The number comparisons are reduced to $O(|q.V| \times |\Sigma|)$. It has been validated by our experiments that this simplification is both efficient and effective.

Proposition 2 *A query q is non-empty* only if $\nexists v_q \in q.V$, *such that* $\forall d_{l_q,l_2}$, $d_{l_q,l_2} < |\{v_2|\ell(v_q) = l_q, \ell(v_2) = l_2, (v_q, v_2) \in q.E\}|$, *where* $l_q \in \Sigma$. □

4.3 Ranking candidate suggestions

Candidate suggestions can be many. Since users may only be able to interpret a small subset of them, AUTOG returns top-k suggestions w.r.t. a ranking function and a user preference component. When users formulate their queries, they may *rank* candidate suggestions differently, because of 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. In this section, we model the preferences between different criteria with a ranking function and a user preference component.

4.3.1 Ranking function and user preference component

This subsection presents a ranking function for possibly novice users who prefer query suggestions that *(i) return more answer graphs and (ii) are structurally diversified*. The first preference simply reflects users' intent to retrieve more answers, whereas the second one recognizes the importance of avoiding similar suggestions (e.g.,[15,34,37]). These two preferences can be quantified as the following objective functions:

1. $\mathsf{sel}(q)$: the selectivity of q on D, is $|D_q|/|D|$.
2. $\mathsf{dist}(q_i, q_j)$: 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 are. For illustration purposes, we adopt the *maximum common edge subgraph* (mces) for dist (see Definition 8 [9,22]). mces is adopted because adding edges (as opposed to nodes) to an existing query appears a natural logical step of composing queries. The distance definition is denoted as BS.

Definition 8 *(BS)* Given two graphs g_1 and g_2, the graph distance based on the maximum common edge subgraph (mces) is defined as follows:

$$\mathsf{dist}(g_1, g_2) = 1 - \frac{|\mathsf{mces}(g_1, g_2)|}{\mathsf{max}\{|g_1|, |g_2|\}},$$

where $\mathsf{mces}(g_1, g_2)$ is a subgraph of g_1 with as many edges as possible that is isomorphic to a subgraph of g_2. □

Example 8 Take f_{18} and f_{22} in Fig. 4 as an example, $\mathsf{dist}(f_{18}, f_{22}) = 1 - \frac{2}{3} = \frac{1}{3}$, according to Definition 8. □

The dist function has a few nice properties [9,22]. It is a metric. It is a reflexive and symmetric function, which can be observed from its definition. Other graph distance functions can be adopted to implement dist. We study the suggestion quality with other popular graph distance metrics [22,36,38] in "Appendix 3" .

Definition 9 *(User intent value of query suggestions)* Given a set of query suggestions $Q' : \{q'_1, q'_2, ..., q'_k\}$ and a user preference component 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 overall ranking function (util) is presented in Definition 9, which is the normalized weighted sum of the two objective functions mentioned above. The two objective functions can be competing: for example, 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. With the util function, we are ready to formulate the ranking problem of query suggestions and analyze it hardness.

Definition 10 Given a query q, a set of query suggestions Q', the ranking function util, a user preference component α and a user- specified constraint k, the *ranked subgraph query suggestion* problem [RSQ] is to determine a subset Q'', $\mathsf{util}(Q'')$ is maximized. □

Theorem 1 *The* RSQ *problem is NP-hard.* □

The RSQ problem is an NP-hard problem, which can be established by a reduction from the maximum independent set (MIS) problem. The proof is presented in "Appendix 2".

Template values of user preference α The user preference α in Definition 9 expresses the relative importance of selectivities and suggestion diversities. To help users to set α, we may derive a set of templates of predefined preference components with intuitive semantics (such as *selectivity-oriented* and *diversity-oriented* suggestions) from the underlying dataset. Users may start with a predefined template and subsequently refine α after reviewing some query suggestions returned.

Alternatively, AUTOG starts with a template of user preference. Based on the suggestions adopted by a user, AUTOG

may learn whether he/she prefers selectivities or diversities. The details of learning parameters from users' feedbacks, however, are beyond the scope of this paper.

Integration of other ranking functions It should be remarked that the ranking function util presented in this subsection is for illustration purposes. That is, other objective functions can be readily plugged into the AUTOG framework. Take the structure search of *PubChem* as a concrete example. We may include application-specific semantics in an additional objective function. Suppose F_T is the set of query templates provided by *PubChem* and the templates in F_T are c-prime features, as discussed at the end of Sect. 3. Suppose users are favorable to suggestions that contain the query templates F_T. This can be achieved by introducing a function app(q') that returns the number of features in F_T contained in q'. AUTOG adds app to util and sets its preference as other objective functions.

4.3.2 *Efficient selectivity and diversity computation*

Next, we present efficient algorithms for determining sel and dist, which enable efficient ranking candidate suggestions.

1. Candidate answer selectivity estimation We first recall some standard notations. We denote $D_q = \mathsf{eval}(q, D)$ as the query evaluation of q on D and D_q is the query result. The selectivity of q (denoted as $\mathsf{sel}(q, D)$) is $|\mathsf{eval}(q, D)|/|D|$.

Recall that $\mathsf{eval}(q, D)$ is NP-hard due to subgraph isomorphism tests. Hence, we propose to leverage feature-based query processing to efficiently estimate eval. The benefits of this approach are twofold. 1) c-prime features can be seamlessly integrated into existing feature-based approaches. This estimation does not incur much overhead. 2) It has been known that the feature-based approach (e.g., [42]) can efficiently determine candidate answer sets of subgraph queries, which are close to the actual answer sets.

In a nutshell, each feature is associated with a set of IDs of the graphs that contain the feature. Given the features of a query F_q, the candidate set is obtained by the *intersection* of the sets of IDs associated with each feature in F_q. The numerous intersections of large ID sets may be costly, especially when ranking suggestions online. Hence, we estimate the selectivity by adopting a systematic sampling after a uniformly random permutation of the *graph IDs* [8].

W.l.o.g, we assume two sets A and B, and $|A| < |B|$. Then, $|A|$ is the population size. The real selectivity, $|A \cap B|$, is the number of success states in the population. $S = |A|/m$ is the number of draws, where m is *the user-specified sampling interval*. The number of observed successes is denoted as k. The probability that the observed successes exactly equals to k is given by

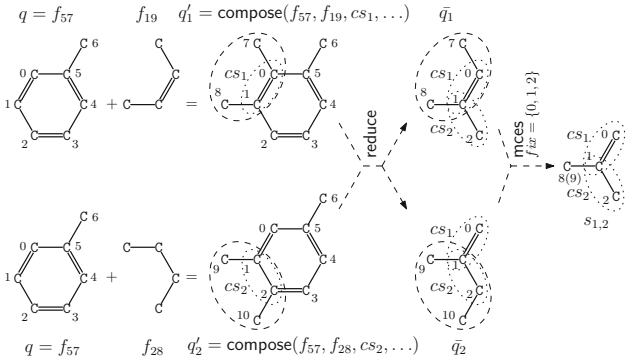

Fig. 5 Triming compositions for mces computation

$$P(X = k) = \frac{\binom{|A \cap B|}{k}\binom{|A| - |A \cap B|}{S - k}}{\binom{|A|}{S}}$$

The error ϵ of our estimation method can be analyzed based on hypergeometric distribution, that describes the probability of k successes in n draws, *without replacement*.

$$|A \cap B|\frac{S}{|A|} - \epsilon \leq k \leq |A \cap B|\frac{S}{|A|} + \epsilon$$

2. Diversity approximation The second component of util is the structural diversity of query suggestions in Q'. dist makes the overall ranking function util submodular, so that greedy algorithms are its natural heuristic. (Please refer to "Appendix 3" for the experiments on ranking using different graph distance functions).

To efficiently implement the dist function of two suggestions, our main idea is to trim the common parts from them before calling the exponential-time algorithm for mces. This can be efficiently computed because (i) the query suggestions are composed by adding different increments on the *same* existing query graph, and (ii) some auxiliary structures between possible composable features can be computed *offline*. For brevity, we omit the tedious pseudo-code but illustrate the major steps with the following example.

Example 9 Given a current query q which is simply a feature f_{57}. Consider two possible compositions (shown in Fig. 5) that construct query suggestions from f_{57} by adding either f_{19} (denoted as q'_1) or f_{28} (denoted as q'_2). Note that the existing query q and increments f_{19} and f_{28}, in this example, are features and their compositions can be enumerated offline. Suppose we compute mces of q'_1 and q'_2. Some parts of q'_1 and q'_2 are trivially common and are not necessary to perform the costly mces computation. Thus, we reduce q'_1 and q'_2 to the trimmed subgraphs $\bar{q}_1$ and $\bar{q}_2$ for computing the non-trivial mces of q'_1 and q'_2. Further, some intermedi-

ate results are indexed offline. Specifically, the major offline steps are presented as follows.

1. Denote cs_1 (resp. cs_2) to be the common subgraph between f_{57} and f_{19} (resp. f_{57} and f_{28}).
2. The subgraph s computed by f_{57} - cs_1 - cs_2 is trivially a part of the mces of q_1' and q_2';
3. $\bar{q_1}$ is obtained by q_1' - s. Similarly, $\bar{q_2}$ is q_2' - s;
4. The embedding of cs_1 (resp. cs_2) in f_{57} is computed offline. It also specifies its location in $\bar{q_1}$ (resp. $\bar{q_2}$), which minimizes the search of mces. In particular, the nodes 0, 1 and 2 of $\bar{q_1}$ must map to the nodes 0, 1 and 2 of $\bar{q_2}$;
5. An mces algorithm determines the mces of $\bar{q_1}$ and $\bar{q_2}$ offline, and it returns $s_{1,2}$.
6. In $s_{1,2}$, the nodes {0,1,2} are from cs_1 and cs_2 (i.e., the existing query q). The non-trivial mces is C-C (1, 8 (or 9)), which contains one edge.

When candidate suggestions are ranked *online*, the size of mces between q_1' and q_2' is obtained by simply adding the sizes of the existing query q (i.e., f_{57}) and the non-trivial mces: $|f_{57}| + 1 = 8$. □

We provide two remarks on the above mces computations. (i) While the query is provided by users online, features and their compositions can be enumerated offline. Therefore, sizes of the non-trivial mces between compositions can be indexed offline. (ii) The above optimization significantly speeds up the online mces computation because query suggestions contain the same existing query graph.

4.3.3 Greedy ranking algorithm

In this subsection, we propose a ranking algorithm for a set of candidate suggestions with respect to the ranking function util and the user preference α. Note that the util function has two monotone submodular components. It can be easily established that the util function is also submodular. A decent property of a submodular function is that greedy algorithms work and guarantees a 1/2OPT approximation ratio [6].

We propose a two-level greedy algorithm (Algorithm 2) to rank the candidate suggestions. A two-level algorithm is proposed because the candidate suggestions can be many and computing diversity between every possible pairs of them involves dist, which is time-consuming. In a nutshell, at the first level, for each feature f embedded in a specific location λ of the query, the algorithm greedily determines its top-k suggestions (denoted as $Q_k^{f,\lambda}$) that increment f in location λ. At the second level, it greedily determines the overall top-k suggestions from $Q_k^{f,\lambda}$s, for all (f, λ)s. Hence, it avoids computing dist of all possible pairs of candidate suggestions. The pseudo-code of the greedy algorithm is presented in Algorithm 2. We elaborate its details below.

Greedy_local: First, GREEDY_LOCAL (Lines 14–15) determines *the possible suggestions composed by adding a feature to the feature-query embedding* (f, λ) *of* q, *denoted as* Q_C. For efficiency purposes, we further restrict q' is increased by at most δ edges (Line 15). Line 16 computes the pairwise dist between suggestions in Q_C. In each iteration step (Lines 17–20), it adds the composed suggestion q_c that makes the util function of $Q_k^{f,\lambda}$ the largest. This step is repeated until it obtains k suggestions in $Q_k^{f,\lambda}$.

Greedy_global: In GREEDY_GLOBAL, GREEDY_LOCAL is invoked $O(|M_q|)$ times (Line 2). In each GREEDY_LOCAL call (Line 3), we obtain candidate suggestions (Line 4) and take the union with the top-k suggestions from each possible (f, λ) obtained so far (denoted as Q'). Line 5 computes the pairwise dist between suggestions in Q' *online*. In each iteration step (Lines 6–10), it adds the composed suggestion q' that makes the util function of Q_k' the largest. This step is repeated until it obtains k suggestions in Q_k'.

Remarks. The greedy algorithm involves a trade-off between the efficiency and suggestion quality. GREEDY_LOCAL obtains 1/2OPT suggestions w.r.t a specific (f, λ), whose computation can be further optimized offline. Since M_q is only available online and the time for computing the user intent value of a query set, T_{util}, can be potentially long, GREEDY_GLOBAL is run only on Q' instead of Q_Cs of all possible (f, λ)s. Alternatively, when the query suggestion time is already acceptable, one may tune AUTOG to produce more suggestions for ranking as follows: (i) One may include more features, e.g., by lowering the minimum support and/or increasing the composabiliy of c-prime features. (ii) AUTOG can be tuned to allow more overlapping features in query decomposition by using the parameter γ. (iii) One may set the maximum query increment size δ to a large value.

4.3.4 Complexity analysis of the greedy ranking algorithm

This section analyzes the asymptotic complexity of the greedy ranking algorithm (Algorithm 2), presented in Sect. 4.3.3. We remark that when determining the user intent value of query suggestions for ranking, the sampling method of selectivity estimation (presented in Sect. 4.3.2) is significantly more efficient when compared to the worst-case exponential-time computation of the query suggestions' diversity, which involves computing structural difference between graphs (dist) multiple times. For a succinct analysis, we assume that there is an oracle that efficiently provides selectivity estimation of a query and omit it in the analysis.

Greedy_local: The time complexity of the local greedy algorithm is simply $O(|Q_C| \times T_{compose} + k \times |Q_C| \times T_{util} + k \times |Q_k^{f,\lambda}|)$.

1. The first term is due to the addition of a feature to (f, λ), where λ is the embedding of f in the current query. $|Q_C|$ is the number of possible suggestions at (f, λ), which is of modest value. $T_{compose}$ is the time for adding a feature in F to (f, λ) of q. Such an addition requires to check whether the feature contains a common subgraph with f. This requires subgraph isomorphism calls.
2. The second term is due to the time for computing the composed suggestion q_c that makes the util of $Q_k^{f,\lambda}$ the largest. Denote T_{util} to be the time for computing the user intent value of a set of suggestions. Recall that in each iteration of the greedy algorithm, we augment $Q_k^{f,\lambda}$ with the suggestion from Q_C that gives the largest user intent value, which takes $O(|Q_C| \times T_{util})$ to find the largest one. The iteration is repeated k times. We remark that to compute the user intent value of $Q_k^{f,\lambda}$, it requires to compute the structural diversity of the $Q_k^{f,\lambda}$ and it involves the dist function. We adopt the technique presented at the end of Sect. 4.3.2 to optimize this step.
3. The last term is the time for outputting the top-k suggestions $Q_k^{f,\lambda}$ of the feature embedding (f, λ).

Greedy_global: The time complexity of the overall greedy algorithm is $O(|M_q| \times T_{local} + k \times |Q'| \times T_{util} + k \times |Q'_k|)$.

1. The first term is due to the calls of local greedy algorithm of feature embeddings. GREEDY_GLOBAL calls GREEDY_LOCAL once for *each* embedding. There are $|M_q|$ embeddings in total. Recall humans often draw small queries. Hence, $|M_q|$ is small in practice.
2. The second term is due to the ranking of Q' to obtain the overall top-k suggestions Q'_k. We remark that each GREEDY_LOCAL call returns k suggestions. Hence, the number of query suggestions ranked by GREEDY_GLOBAL $|Q'|$ is $k \times |M_q|$, which is of modest number. In each iteration, the suggestion from $|Q'|$ that makes the value of util of Q'_k the largest (computed in T_{util} time) is added to Q'_k. The iteration is repeated k times.
3. Finally, the last term is the time for outputting the top-k suggestions $|Q'_k|$.

Based on the above analysis, we observe that the algorithm calls two worst-case exponential-time subroutines with graphs of small or modest sizes. In particular, they are (i) the subgraph isomorphism calls when determining the possible common subgraphs of feature pairs, and (ii) the structural difference between query suggestions for computing their diversities. In Sect. 4.3.2, we have proposed an optimization for computing suggestion diversities. Moreover, in the next section, we propose an index to further optimize them.

Algorithm 2 Ranking Candidate Suggestions

Input: a query q represented by M_q, user preference component α, number of suggestions requested k and max. increment size δ
Output: the top-k query suggestions Q'_k

```
1: function GREEDY_GLOBAL(q, α, k)
2:     for all (f, λ) ∈ M_q do
3:         Q_k^{f,λ} = GREEDY_LOCAL(f, λ, q)        //local ranking w.r.t (f, λ)
4:         Q' = Q' ∪ Q_k^{f,λ}
5:     compute dist between each pair of suggestions in Q'
6:     Q'_k = {}
7:     for all i = 1 ... k do                        //global ranking
8:         q_max = argmax(q', util(q', Q'_k)), where q' ∈ Q'
9:         Q'_k = Q'_k ∪ q_max
10:        Q' = Q' \ q_max
11:    return Q'_k
12: function GREEDY_LOCAL(f, λ, q)                   // local ranking
13:    Q_k^{f,λ} = {}
14:    Q_C is the possible suggestions composed by adding a feature to
       the feature-query embedding (f, λ) of q, where
15:        q_c ∈ Q_C implies |q_c| − |q| ≤ δ
16:    compute dist between each pair of suggestions in Q_C
17:    for all i = 1 ... k do
18:        q_max=argmax(q_c, util(q_c, Q_k^{f,λ})), where q_c ∈ Q_C
19:        Q_k^{f,λ} = Q_k^{f,λ} ∪ q_max
20:        Q_C = Q_C \ q_max
21:    return Q_k^{f,λ}
```

5 Indexed autocompletion for subgraph queries—AutoGI

To optimize the autocompletion framework presented in Sect. 4, we present a novel index, called feature DAG (FDAG) index, *and* its associated algorithms. It is the first structure that indexes features and records their structural information for query autocompletion, including *subgraph isomorphisms between features*, *features' automorphisms* and *auxiliary structural differences between query compositions*. We present the definition of FDAG and its operations, whereas we postpone the details of the index construction to "Appendix 4".

5.1 Feature DAG (FDAG) index

Prior to the definition of FDAG, we present the design rationales of FDAG:

1. GREEDY_LOCAL (Algorithm 2, Lines 14–15) involves adding a feature $f_j \in F$ to a query q, via a feature f_i embedded in q, where f_j and f_i have a common subgraph cs, i.e., cs $\subseteq_{\lambda_i} f_i$ and cs $\subseteq_{\lambda_j} f_j$. Numerous suggestions may be potentially generated (i.e., $|Q_C|$ can be large). Determining all possible common subgraphs of two features is costly. Hence, FDAG indexes all subgraph isomorphic embeddings between features.

2. Suggestions that are formed via common subgraphs which are automorphic to each other are structurally equivalent. FDAG indexes automorphic embeddings of each feature, so that automorphic suggestions are generated only once.
3. All possible pairwise feature compositions are enumerated and indexed so that the time for adding an increment to a feature ($T_{compose}$) is done in an FDAG lookup.
4. As motivated in Sect. 4.3.2, it has been known that determining structural differences between graphs is potentially costly (i.e., T_{util} can be large). Thus, FDAG indexes the auxiliary structure for determining structural differences between compositions (illustrated with Example 9) so that T_{util} is significant reduced.
5. The ranking function presented in Sect. 4.3.1 involves selectivity estimations. FDAG indexes the graph IDs of a feature with a predefined sampling interval m.

The feature DAG (FDAG) index is then formally presented in Definition 11. We provide the algorithmic details for constructing FDAG in "Appendix 4".

Definition 11 FDAG is a DAG (V, E, M, anc, des, A, ζ, η, D), where

1. V is a set of index nodes. Each node v represents a feature denoted as f_v. For presentation simplicity, we may often use f_v to refer to the index node;
2. $E : V \times V$ is a set of edges, $(v_i, v_j) \in E$ *iff* $f_{v_i} \subseteq_\lambda f_{v_j}$; further, M is a function that takes an edge (v_i, v_j) as input, i.e., $f_{v_i} \subseteq_\lambda f_{v_j}$, and returns the subgraph isomorphism embeddings of f_{v_i} in f_{v_j}, often denoted as M_{f_i,f_j}; anc and des are functions that take an index node v as input and return its ancestor and descendant nodes, respectively.
3. A takes a feature f_v as input and returns the automorphism embeddings of f_v, often denoted as A_{f_v};
4. ζ is a function that takes a c-prime feature f_v as input and returns a set of composition records C as output, where each record in C is a 6-ary tuple (f_v, f_{v_j}, cs, λ_v, λ_{v_j}, F_l), where cs is the common subgraph of f_v and f_{v_j}, λ_v (resp. λ_{v_j}) specifies the embedding of cs in f_v (resp. f_{v_j}) and F_l is a set of features embedded in the composed graph;
5. η is a function that takes a pair of compositions as input and returns the auxiliary structural difference between the pair; and
6. D takes an index node f_v as input and outputs a sample of the IDs of the graphs that contain f_v. We often denote the graph IDs of f_v as D_{f_v}. □

FDAGs can be large in worst case. However, in practice, their sizes are far from the worst-case ones. (i) The number of edges $|E|$ is $O(|F|^2)$, where F is the feature set. From our experiments, the $|E|$ values were only $18\times|F|$ on average (see Table 8). (ii) A feature pair may have exponentially many subgraph isomorphism embeddings between them. In practice, the average number of such embeddings per feature pair is around 2. (iii) Regarding automorphisms, our experiments show that about 12% of the features have multiple automorphisms. (iv) Each feature pair has around 120 possible compositions on average (indexed in ζ).

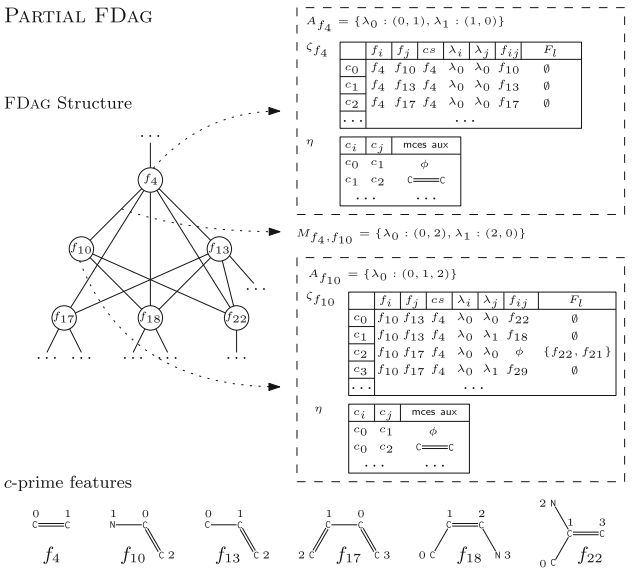

Fig. 6 Major structures of the (partial) FDAG of PUBCHEM

Example 10 We constructed the FDAG for PUBCHEM and illustrated its partial sketch in Fig. 6. Suppose f_{10} is the current query. f_{13} is a possible increment to f_{10} that forms f_{22}, via f_4. Query suggestion f_{22} can be efficiently determined by using FDAG as follows.

f_4 is located from anc(f_{10}). f_{13} is in des(f_4). f_{22} is retrieved from the compositions of f_{10} (i.e., $\zeta_{f_{10}}$) via an FDAG lookup. A trivial query composition is needed when the composed suggestion is not a feature. The composed suggestion (i.e., f_{22}) is ranked against other possible candidate suggestions (formed by other compositions of f_{10}). This is efficient because the intermediate results of the structural differences between compositions of f_{10} are recorded in η. □

5.2 Autocompletion by using FDAG

We end this section by highlighting how FDAG optimizes the online ranking (Algorithm 2). Determining the set of possible candidate suggestions Q_C (Line 14) that have large util values (Line 18) is computationally costly. In Line 14 of GREEDY_LOCAL, the query increments that can be composed with the feature f (at the location λ of q) are fixed. Therefore, the FDAG indexes all possible compositions C of

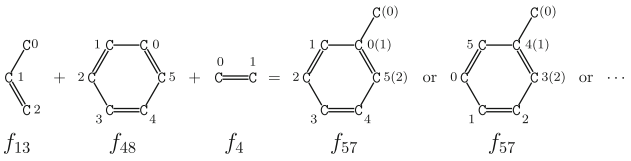

Fig. 7 Example of redundant compositions

f in ζ offline. Then, they can be efficiently retrieved online. Given a composition $c = (f, f', \mathsf{cs}, \lambda_1, \lambda_2)$ and a suggestion q', the constraint $|q'| - |q| \leq \delta$ can be then easily checked (Line 15). Then, the util function has the sel and dist components (Lines 16–18). sel can be efficiently estimated from D of FDAG. Regarding dists between q' and other suggestions constructed from (f, λ), some intermediate results had been indexed in η of FDAG. Further, since q' differs from q by at most δ edges, dist can be efficiently derived from η. Hence, T_{util} is reduced.

6 Pruning redundant compositions via graph automorphism

A unique problem of subgraph query composition is that compositions can be *structurally identical* and therefore redundant. These redundant compositions adversely affect the performance of AUTOG in two ways. First, the generation of these redundant compositions is useless. Second, these redundant compositions are mixed with the useful ones, and thus, subsequent ranking of suggestions is required to eliminate them. In this section, we detail an automorphism-based optimization to prune them, by indexing automorphisms of features A in FDAG.

Example 11 (Redundant compositions) Figure 7 shows an example of redundant compositions. Here, the cs is f_4. Clearly, f_4 has 2 and 6 embeddings in f_{13} and f_{48}, respectively. f_4 has two automorphic node configurations: $(0 \mapsto 0,\ 1 \mapsto 1)$ and $(0 \mapsto 1,\ 1 \mapsto 0)$). According to Definition 6, we have 6 (i.e., $2 \times 6 / 2$) possible ways to form graph f_{57} by combining f_{13} and f_{48} via f_4. These 6 suggestions result in the same q', i.e., 5 of them are redundant. □

The intuition of the redundant compositions is that the graph increment (e.g., the feature f_{48} in Example 11) may "rotate" and are combined to form the same query suggestion. Such "rotations" can be captured by the automorphism relation of the graph increment. Recall that the automorphism relation A_G of a graph G is an isomorphism of G to itself. Determining A_G is one of the NP-hard problems. However, when the vertex degrees are bounded by a constant, there is a polynomial time algorithm [25] for solving the graph automorphism problem since the sizes of features are often small (e.g., the largest feature of PUBCHEM contains 14 vertices when we set the minimum support of features to 10%, and its degree must be no larger than 14). Thus, determining the automorphisms of each feature A is efficient offline. They are indexed FDAG (see Definition 11).

We elaborate the major steps of the pruning with (f_i, f_j, cs), where cs is a common subgraph of f_i and f_j. cs may have multiple embeddings in f_i and f_j, respectively. Denote such embeddings as M_{cs,f_i}, which are all the possible subgraph isomorphic embeddings of cs in f_i. Without any background knowledge of the existing query q other than $f_i \subseteq_\lambda q$, we perform two pruning steps exploiting the automorphism of cs and f_j.[9]

1. We prune redundant compositions resulted from the "rotation" of cs. Suppose there is an automorphic embedding $\lambda \in A_{\mathsf{cs}}$ that relates two *node configurations* of cs, V_0 and V_1 : $(V_0.v_0 \mapsto V_1.v_1, ..., V_0.v_{|V(\mathsf{cs})|} \mapsto V_1.v_{(|V(\mathsf{cs})|)}$.[10] Then, cs can be embedded in a subgraph s_i of f_i in multiple ways M_{cs,f_i}. Suppose the two embeddings λ^0_{cs,f_i} and λ^2_{cs,f_i} of M_{cs,f_i} map two node configurations of cs (V_0 and V_1) to the same node configuration of f_i. Then, we keep only one of such embeddings.
   This step is to reduce the number of compositions from $|M_{\mathsf{cs},f_i}| \times |M_{\mathsf{cs},f_j}|$ to $|M_{\mathsf{cs},f_i}| \times |M_{\mathsf{cs},f_j}|/|A_{\mathsf{cs}}|$.
2. We prune the redundant compositions resulted from the "rotation" of f_j. Suppose there are two embeddings λ_0 and λ_1 of cs into the subgraph s_j of f_j. Denote V_0 and V_1 are the node configuration of f_j that the nodes of cs mapped to, specified by λ_0 and λ_1. If V_0 and V_1 are automorphic, we keep only one such embedding and prune the other.

Example 12 Following up Example 11, the pruning is illustrated as follows. The cs is f_4. The automorphism of f_4 is $A_{f_4} = \{\lambda_0 : (0 \mapsto 0,\ 1 \mapsto 1), \lambda_1 : (0 \mapsto 1,\ 1 \mapsto 0)\}$. f_4 is embedded into f_{13} in two ways: $M_{f_4,f_{13}} = \{\lambda_0 : (0 \mapsto 1,\ 1 \mapsto 2), \lambda_1 : (0 \mapsto 2,\ 1 \mapsto 1)\}$. The two node configurations of f_4 are V_0:(0,1) and V_1:(1,0), and they are automorphic because of λ_1 from A_{f_4}. Thus, we prune λ_1 from $M_{f_4,f_{13}}$.

f_4 has 6 embeddings in f_{48}, e.g., $M_{f_4,f_{48}} = \{\lambda_0 : (0 \mapsto 0,\ 1 \mapsto 5), \lambda_1 : (0 \mapsto 5,\ 1 \mapsto 0), \ldots\}$. λ_0 and λ_1 map the nodes of cs to a subgraph of f_j, where the node configurations are V_0:(0,5,...) and V_1:(5,0,...). Consider the automorphism of f_{48}, where $A_{f_{48}} = \{\ldots, \lambda_5 : (0 \mapsto 5,\ 1 \mapsto 4, 2 \mapsto 3,\ 3 \mapsto 2,\ 4 \mapsto 1,\ 5 \mapsto 0)\}$. V_0 and V_1 are automophric because of λ_5. Therefore, λ_1 of $M_{f_4,f_{48}}$ can be pruned. □

[9] The automorphism of f_i is not used for pruning because f_i is embedded in q, and the automorphism of q is unknown offline.

[10] The automorphism relation A_{cs} of cs can be retrieved from FDAG.

Table 1 Some characteristics of the datasets

Dataset	$\|D\|$	$avg(\|V\|)$	$avg(\|E\|)$	$\|l(V)\|$	$\|l(E)\|$
AIDS	10 K	25.42	27.40	51	4
SYN- 1	10 K	11.02	30.53	20	20
SYN- 2	10 K	11.02	30.53	57	80
PUBCHEM	1 M	23.98	25.76	81	3

Table 2 Statistics of the feature sets

Dataset	*minSup*	$\|F\|$	$avg(\|V\|)$	$avg(\|E\|)$
AIDS	0.10	460	6.15	5.20
SYN- 1	0.05	1860	3.81	3.15
SYN- 2	0.05	1453	4.40	4.04
PUBCHEM	0.10	1206	7.44	6.47

The correctness of the automorphism-based pruning can be established by a simple proof of contradiction.

7 Experimental evaluation

This section presents an experimental evaluation of the proposed AUTOG framework. We first investigated the suggestion quality via user tests and simulations and then conducted an extensive performance evaluation on popular real and synthetic datasets. In particular, we studied the usefulness of AUTOG, the overall performance of AUTOG, the effectiveness of the optimizations and the effect of the parameters of AUTOG.

Software We implemented a prototype of AUTOG. The interface is shown in Fig. 1. The prototype was mainly implemented in C++, using VF2 [12] for subgraph test and the McGregor's algorithm [27] (with minor adaptation) for determining mces. We used the GSPAN implementation from [41] for frequent subgraph mining.

Hardware We conducted all the experiments on a machine with a 2.67GHz processor and 64GB memory running the Linux OS. All the indexes were built offline once and loaded from a hard disk and were then fully memory resident for online query suggestions.

Datasets We used the datasets and query sets provided by IGRAPH [16] and followed their default settings. We used two popular benchmarked real datasets: (i) PUBCHEM [31], a real chemical compound dataset consisting of 1 million graphs, and (ii) AIDS [30] (the AIDS Antiviral dataset), which consists of 10,000 graphs. For the synthetic datasets, we used synthetic.10K.E30.D5.L20 and synthetic.10K.E30.D5.L80 (hereafter referred to as SYN- 1 and SYN- 2), both of which consist of 10,000 graphs. Table 1 shows some characteristics of the datasets: the number of graphs ($|D|$), the average number of vertices and edges ($avg(|V|)$ and $avg(|E|)$), and the number of vertex and edge labels ($|l(V)|$ and $|l(E)|$).

Query sets The query sets were taken from [16], with the query size ranging from 4 to 24. Each query set of a particular size contained 100 queries.

For time measurements, we reported the elapsed wall-clock times. It is known that there are large variations in subgraph query times [40]. To avoid the reported times being governed by few long (or short) queries, we discarded the runtimes that were beyond two standard deviations from the mean. The reported runtime was the average of the remaining runtimes.

Mining of c**-prime features** We ran GSPAN to obtain sufficient features for AUTOG to compose suggestions. In particular, we set the default minimum support value ($minSup$) for the real datasets to 10%. We set $minSup$ to 5% for synthetic datasets simply because their frequent subgraphs are relatively scarce. The maximum feature size $maxL$ was set to 10 for all datasets. Some statistics of the features are summarized in Table 2.

Default AUTOG settings The default maximum increment size (i.e., δ) was set to 5, which is large enough to provide a large number of candidate suggestions. We set the default composability c to infinity unless we specifically studied its effects. There are five parameters (i.e., m, γ, α, k and $|q|$) of online AUTOG processing. In the default setting, we set m to 4, γ and α to 0.5, k to 10 and $|q|$ to 8, unless otherwise specified.

7.1 Suggestion quality

Simulations We investigated the qualities of the suggestions via simulations under a large variety of parameter settings. For each target query, we started with a random 2-edge subgraph. In each step, we called AUTOG. Then, we chose a suggestion with the largest size. If no suggestion was useful, we augmented the query by a random edge toward the target query. Each target query set contained 100 queries, which are publicly available[43]. To study suggestion qualities, we employ several popular metrics, listed in Table 3.

We report some representative results from PUBCHEM in Tables 4, 5 and 6. Table 4 shows the quality metrics of Q8 with various δ. We remark that same trends were observed on Q12 and Q16. Table 5 shows the quality metrics of various target query sizes. 1 Hit shows that AUTOG suggestions were almost always used, and #AUTOG shows that the suggestions were used in multiple iterations of query formulation; the numbers of edges added by AUTOG were around 30%; and TPM shows that AUTOG saved roughly 42% mouse clicks in query formulation.

Table 3 Quality metrics and their meanings

Metric	Meaning
1 Hit	The % of the queries that AUTOG returned at least one useful suggestion
#AUTOG	The average number of suggestions accepted in the simulation
AUTOG $\|E\|$	The average number of edges obtained from suggestions
TPM	The *total profit metric (TPM)* adopted from [29], which quantifies the % of mouse clicks saved by AUTOG in visual graph query formulation: $TPM = \frac{number\ of\ clicks\ saved\ by\ AutoG}{number\ of\ clicks\ without\ AutoG}$

Table 4 Quality metrics versus δ (PUBCHEM)

δ	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
1	99	4.7	4.7	59
2	99	2.7	3.8	52
3	98	2.2	3.3	45
4	99	2.2	3.0	42
5	98	2.3	3.1	43

Table 5 Quality metrics versus $|q|$ (PUBCHEM when $\delta = 3$)

$\|q\|$	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
8	98	2.2	3.3	45
12	99	3.3	5.1	44
16	100	4.0	6.7	42

Table 6 Quality metrics versus k (PUBCHEM when $\delta = 3$)

k	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
4	86	1.5	2.0	26
6	92	1.8	2.5	34
8	94	2.0	2.9	40
10	98	2.2	3.3	45

Table 6 shows the suggestion quality when we varied k. Table 6 shows the qualities increased with k. It is not surprising because as more suggestions are returned, the higher chance some of them are adopted.

Table 7 shows the quality metrics of Q8 with various α. The results showed that the suggestion qualities increased as the value of α increased and were stable when $\alpha \geq 0.1$. That is, when the factor of the selectivities of suggestions (e.g., $\alpha \geq 0.1$) was adequately significant in the ranking, AUTOG produced high quality suggestions. To obtain helpful suggestions, α may be set to some values greater than 0.1 so that both selectivities and diversities involve in suggestion ranking.

Table 7 Quality metrics versus α (PUBCHEM when $\delta = 3$)

α	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
0.00	63	0.8	1.2	17
0.02	86	1.3	2.2	31
0.04	83	1.2	2.0	29
0.06	86	1.3	2.4	34
0.08	90	1.6	2.6	36
0.10	93	1.6	2.7	39
0.20	98	1.8	2.9	41
0.40	97	2.1	3.1	44
0.60	98	2.2	3.2	44
0.80	98	2.2	3.2	45
1.00	99	2.1	3.1	44

User test Next, we conducted a user test with 10 volunteers. Each user was given 2 queries with high, medium and low TPM values, respectively, from the simulation. We randomly shuffled these 6 queries.[11] The users were asked to formulate the target queries via the visual aid shown in Fig. 1. They expressed their level of agreement to the statement "AUTOG *is useful when I draw the query.*" via a symmetric 5 level agree-disagree Likert scale, where 1 means "*strongly disagree*" and 5 means "*strongly agree*").[12]

Our result showed that the correlation coefficient between TPMs and users' points is 0.96 and the p value is 0.002. Therefore, TPM is a good quality indication of AUTOG. The average ratings of the queries with high, medium and low TPM values are 4.55 (between "*strongly agree*" and "*agree*"), 2.95 ("*neither agree nor disagree*") and 1.65 (between "*disagree*" and "*strongly disagree*"), respectively.

7.2 Index construction performance

Next, we report the performance of building FDAG in Tables 8, 9 and 10. The major steps are (i) to build the structure of the FDAG, (ii) to enumerate all feature pair compositions and (iii) to precompute for the mces distances. We elaborate the results below.

Construction of the FDAG structure The columns $|V_{FDag}|$ and $|E_{FDag}|$ in Table 8 report the number of vertices and edges of FDAG. It can be observed that the FDAG structures are sparse. Such construction times were small.

[11] Readers may find the full list of queries for investigating the suggestion quality from the project site https://goo.gl/Xr9MRY. Further, a short video shows how users may interact with the AUTOG prototype.

[12] The questionnaire used in the tests can be found at http://goo.gl/dFRdwj.

Table 8 FDAG structure construction

Dataset	$\|V_{FDag}\|$	$\|E_{FDag}\|$	*Time* (s)
AIDS	460	6965	0.4
SYN- 1	1860	21,822	1.4
SYN- 2	1453	25,569	1.1
PUBCHEM	1206	31,610	2.6

Table 10 Graph distance precomputation

Dataset	Composition pairs	*Time* (s)
AIDS	5370 M	30,799
SYN- 1	562 M	20,719
SYN- 2	17,813 M	144,334
PUBCHEM	22,636 M	824,825

Optimization for composition enumeration We enumerated all the possible feature pair compositions as detailed in "Appendix 4". The results are reported in Table 9. A fact from "larger fs" is that a large portion of graphs formed by small features contained new large features.

"Before opt" reports the total numbers of possible feature pair compositions. It can be observed that while the numbers of features $|V_{FDag}|$ were modest, the numbers of possible compositions generated (using the default δ value) were large (e.g., millions of compositions for AIDS, PUBCHEM and SYN-2). "After autom. opt" shows the numbers of compositions optimized after applying the automorphism-based optimization presented in Sect. 6. The optimization determined that on average, over 10% of the compositions were *redundant* and pruned, and in particular, 30% of the compositions for the AIDS datasets were pruned. "After nec. cond." reports the numbers of the *useless* compositions pruned by the necessary condition introduced in Sect. 4.2. They reduced 13 and 45% of the compositions after the automorphism-based optimization for the AIDS and PUBCHEM datasets, respectively. This optimization prunes few compositions of the synthetic datasets since the graphs were randomly generated, and thus, such patterns could not be found.

Graph distance precomputation The precomputation of some auxiliary structures for the graph distance between each pair of compositions for each node in FDAG is reported in Table 10. We report the numbers of composition pairs and the precomputation times with the proposed technique in Sect. 4.3.2. The precomputed results are used in online processing.

Varying δ The δ value determines the number of candidate suggestions to be ranked; the larger δ, the more suggestions to be ranked. In Fig. 8, we show the effectiveness of the automorphism-based optimization when δ was varied. The results show that the effectiveness was stable as δ was varied on all datasets.

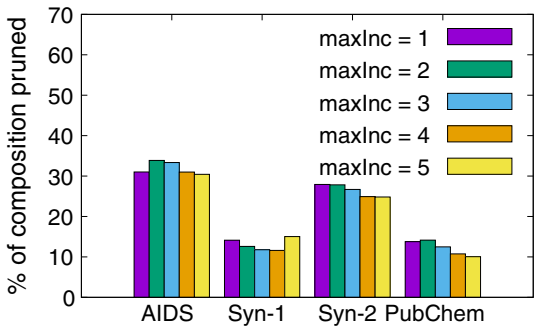

Fig. 8 Pruning of the automorphism optimization

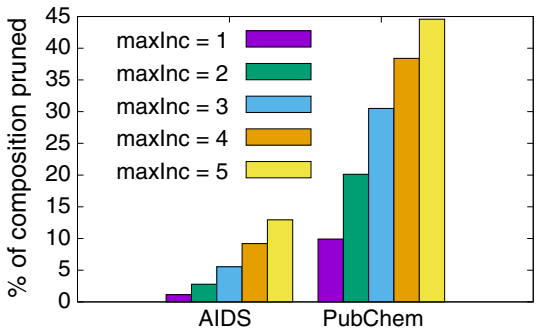

Fig. 9 Pruning of the necessary condition

Next, we further investigated the effects of δ on the pruning of empty suggestions using the necessary condition. Figure 9 shows that this optimization on real datasets was more effective when the δ values were larger. This reflects that when one uses large increments, the resulting suggestions deviate more from those that could retrieve some data graphs. Synthetic datasets were skipped as this optimization was mainly effective on our real datasets.

Table 9 Composition enumeration

Dataset	Larger fs	Before opt	After autom. opt	After nec. cond.	*Time* (s)
AIDS	92%	2021 K	1406 K	1224 K	1601
SYN- 1	77%	945 K	804 K	804 K	489
SYN- 2	90%	4375 K	3289 K	3289 K	4293
PUBCHEM	95%	8372K	7530 K	4173 K	28,517

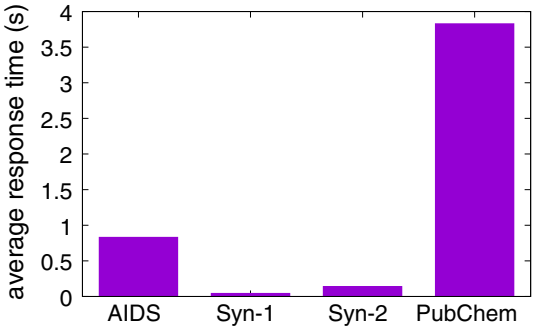

Fig. 10 Overall—default

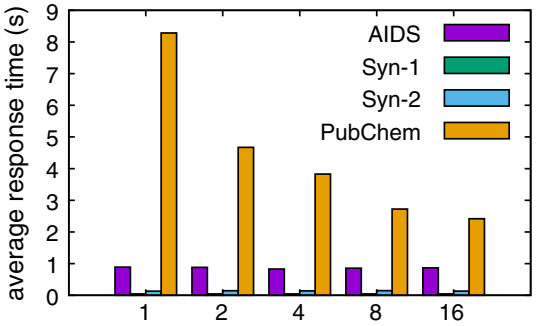

Fig. 11 Overall—ART versus m

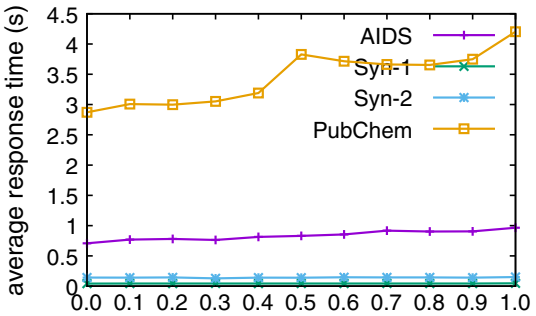

Fig. 12 Overall—ART versus γ

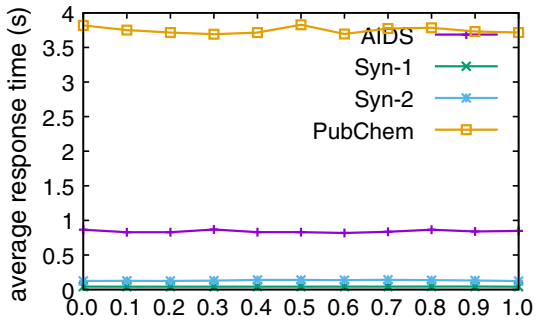

Fig. 13 Overall—ART versus α

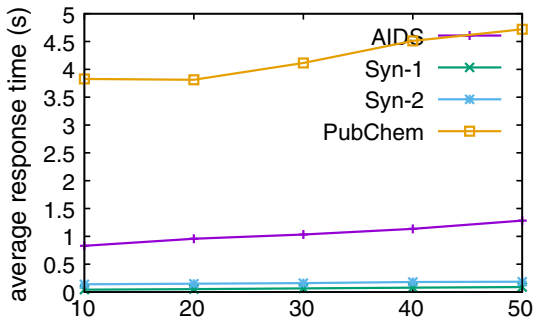

Fig. 14 Overall—ART versus k

7.3 Online autocomplete performance

We conducted a detailed evaluation of the online AUTOG processing. We report the *average response time* (ART) of AUTOG under the *default setting* in Fig. 10. For the synthetic datasets, we obtained short ARTs as their feature pair compositions were relatively few. The ARTs of AIDS and PUBCHEM were slightly shorter than 1s and 4s, respectively. Thus, the AUTOG time is generally short. We remark that the default value of c is set to infinity leading to the longest ARTs. As c decreases, the ART decreases, too.

Varying m We varied m from 1 to 16, to study its impact on the overall ART. The larger m, the smaller the sample sizes for estimating the selectivities of the ranking function. The result is presented in Fig. 11. m has negligible effects on AIDS, SYN- 1 and SYN- 2 datasets. The reason is that selectivity estimation was not the performance bottleneck. However, the candidate answer sets of queries on PUBCHEM were large and the selectivity estimation contributed notable computation times to query suggestions. Hence, as m decreased, the ART of PUBCHEM decreased.

Varying γ and α We ranged γ from 0 to 1. Figure 12 shows the effects of γ on ARTs. The ART was always less than 4.2s. We also noticed that the ART increased slightly when γ was approaching to 1. The higher the value of γ, the more overlapping was allowed and the more features were returned by query decomposition. With more features of queries, AUTOG took more time to rank possible query suggestions. We verified that since α was only a weight in the util function, it did not have a noticeable impact on ART, as shown in Fig. 13.

Varying k We varied k from 10 to 50 and reported the ARTs for each dataset in Fig. 14. The largest value of k tested was 50. The results show that the times were less than 1.5s for AIDS, SYN- 1 and SYN- 2. The time for PUBCHEM was shorter than 4.7s. As expected, the overall ART increased as k increased. The reason is that AUTOG determined and compared more possible suggestions, when the values of k were large.

Varying query sizes Figure 15 shows the ART as the query size increased. For the queries of sizes smaller than 16, AUTOG finished within 10s. For the queries of the sizes 24, the ART can be as long as 40s (e.g., for PUBCHEM). This is due to the NP-hardness of mces in global diversification (i.e.,

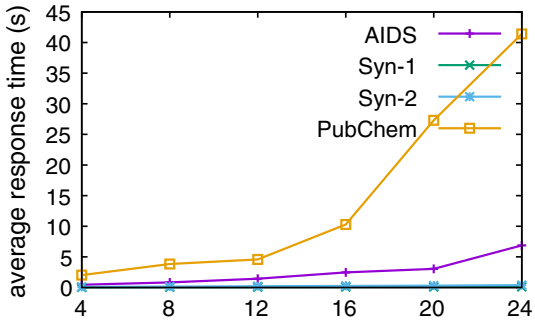

Fig. 15 Overall—ART versus $|q|$

Table 11 Number of c-prime features by varying c

c	AIDS	SYN- 1	SYN- 2	PUBCHEM
16	84	1180	545	88
32	136	1597	746	164
64	243	1793	1058	414
128	404	1857	1290	951
256	457	1860	1428	1178
512	460	1860	1451	1206

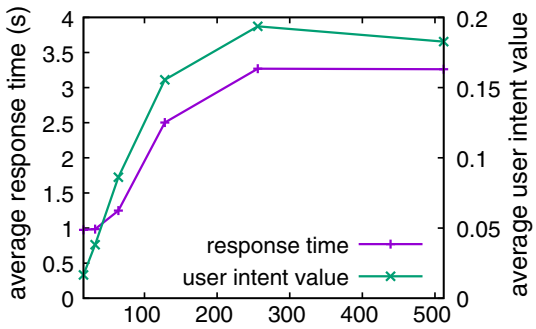

Fig. 16 Effects of c-prime(PUBCHEM)—vary c

their runtimes were exponential to query sizes). However, it should be remarked that the result sets of such large-sized queries are almost always small (e.g., roughly 25 on average for Q24 of PUBCHEM), which are often human manageable, and thus, the needs of AUTOG were arguably less when compared to those of smaller sizes.

Varying c When the values of c were decreased, the numbers of features in FDAG were reduced, too. The numbers of c-prime features by varying c are reported in Table 11. Figures 16 and 17 show the effects on the suggestions in terms of ART, user intent value and their selectivities and diversities on PUBCHEM, respectively. As expected, when the values of c were small, both selectivites and diversities were low. c was easy to set because these quantities were saturated when c reached a few hundreds. In default settings, c was infinity. However, if we sought smaller ART, we could set c to a smaller value.

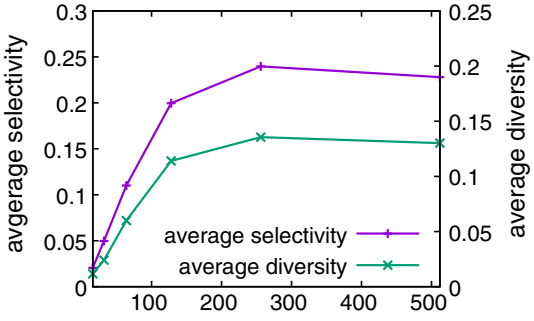

Fig. 17 Effects of c-prime (PUBCHEM)—vary c

In summary, we observed from Figs. 10 to 15 that the proposed AUTOG framework can interactively determine suggestions under a large variety of parameter settings. We have conducted additional experiments to investigate the detailed performance of AUTOG under different settings and other implementations. We have presented the results in "Appendix 3".

8 Related work

There have been some innovative works on query autocompletion for keywords (e.g., [2,29,39]). For brevity, we could not include them. This section focuses on some representative works related to graph databases and their usability.

Graph features Various graph features have been proposed to indexing graphs (e.g., [10,13,35,42,45]), to enhance query processing, by filtering non-answers efficiently. In comparison, c-prime features are defined with composability and how they are connected to form larger graphs are indexed. Hence, they assist users to compose their queries.

Query reformulation Mottin et al. [28] studied the problem of graph query reformulation. The reformulated queries maximally cover the results of the current query. This approach assumes that all query results are relevant. When queries are small (e.g., queries of the size 8 for PUBCHEM), the number of answers is 30K graphs on average, according to our simulation. Users may not interested in all of them. In contrast, AUTOG ranks suggestions based on their selectivities and the diversities.

XML query autocompletion Li et al. [14] proposed to extend keyword search autocompletion to XML queries. In [23], structures are associated with the query keywords. However, keyword searches are inherently difficult (if possible at all) to express structural queries. LotusX provides position-aware autocompletion capability for XML[24]. Autocompletion learning editor for XML provides intelligence autocompletion [1]. In contrast, this paper focuses on subgraph queries (structural search) for graphs.

Visual query composition To alleviate the burdens of structural query composition, visual aids (or GUIs) have been studied, especially in the context of XML queries [7,11,33]. For example, graphical constructs of XML queries (XML-GL) have been proposed [11]. QURSED provides a query editor for building reports [33]. The XML Query By Example (XQBE) provides tools to express graphical constructs of complicated XML queries [7]. One possible reason why GUIs have received significant research attention is that XML data are structures. Their queries are tedious to compose, and they are naturally visualized as pictures. The same arguments can be applied to graph databases[4,18], but their data and query languages are even more complex. GQBE presents a system that allows user to query knowledge graphs by example [20]. The support of interactive simple feedback [3] at opportunity times [5] via a GUI differs from query autocompletion.

Exploratory search Exploratory search has known to be useful for enhancing interactions between users and search systems (e.g., [26]). Obviously, the idea of exploration search can be applied to graph data. The query autocompletion is consistent to exploratory search that it allows users to construct their queries incrementally and explore their intermediate query results.

9 Conclusion

This paper presents a subgraph query autocompletion framework, namely AutoG, that provides query suggestions to users as they are formulating their queries. The logical units of query increments are c-prime features, which can be composed from smaller features in no more than c ways. Existing structural features of graphs can be adopted to the concept of c-prime features. We have proposed query decomposition, candidate suggestion generations and ranking. We have proposed an index called FDag and optimization methods. We conducted extensive experiments on both real and synthetic datasets. The results showed that AutoG saves about 40% of users' mouse clicks in query formulation, the response time of suggestions is short, and the optimizations are effective under a large variety of settings. In future work, we are investigating AutoG for users with different domain knowledge and adopting machine learning techniques on query logs to determine query templates and parameters for AutoG.

Acknowledgements Peipei Yi and Byron Choi are partially supported by the HK-RGC GRF 12201315 and 12232716. Sourav S Bhowmick is supported by the Singapore MOE AcRF Tier-1 Grant RG24/12 and MOE AcRF Tier-2 Grant 2015-T2-1-040. Jianliang Xu is partially supported by the HK-RGC GRF 12244916 and 12200114.

Appendix 1: Properties of c-prime features

This appendix presents the definition of frequent features and the anti-monotonicity and downward-closure properties of c-prime features.

Definition 12 *(Frequent feature)* Given a graph database D, a *frequent feature* f is a subgraph of the graphs in D and $|D_f| \geq minSup$, where $minSup$ is the minimum support of features. F is the frequent feature set of D with respect to $minSup$. □

Proposition 3 (Anti-monotonicity property) *If f is a c-prime feature and f' is a subgraph of f (i.e., $f' \subseteq_\lambda f$), then f' is a c-prime feature.* □

Proof sketch It is obvious that f' is a frequent subgraph. For any f'_i and f'_j such that $f' = \mathsf{compose}(f'_i, f'_j, \mathsf{cs}(f'_i, f'_j), \lambda'_i, \lambda'_j)$. It is possible because by the a priori property of frequent subgraph, f'_i, f'_j and $\mathsf{cs}_k(f'_i, f'_j)$ are frequent. Thus, we can use these subgraphs to compose queries. Since f is connected, we can always find connected supergraphs of f'_i and f'_j, called them f_i and f_j such that $f = \mathsf{compose}(f_i, f_j, \mathsf{cs}(f_i, f_j), \lambda_i, \lambda_j)$. Again, by the a priori property of frequent subgraph, f_i and f_j are frequent. Therefore, for each way of constructing f', we can always determine a corresponding way to construct f. The composability of f is larger than or equal to that of f'. Since f is c-prime, f' is c-prime. □

In the context of c-prime features, the downward-closure property is also just a synonym of anti-monotonic property.

Proposition 4 (Downward-closure property) *If f is a non-c-prime feature, and f'' is a frequent feature and a supergraph of f (i.e., $f \subseteq_\lambda f''$), then f'' is a non-c-prime feature.* □

Proof sketch We can apply the same argument in the proof sketch of Proposition 3. Since f'' is a supergraph of f, the composability of f'' is larger than or equal to that of f. Since f is not c-prime f'' is not c-prime. □

Appendix 2: Analysis of the ranked subgraph query suggestion (RSQ) problem

Theorem 2 *The* RSQ *problem is NP-hard.* □

Proof sketch [MIS] Given a graph G=(V, E), where V and E are the sets of vertices and edges, respectively, an independent set (IS) is a set of vertices V', such that $V' \subseteq V$, there does not exist $v_i, v_j \in V'$ and $(v_i, v_j) \in E$. The MIS problem is to determine an IS V' such that there does not exist an IS whose size is larger than V'.

Reduction We start with an instance of MIS problem G=(V, E). We construct a query suggestion set Q_V such that for

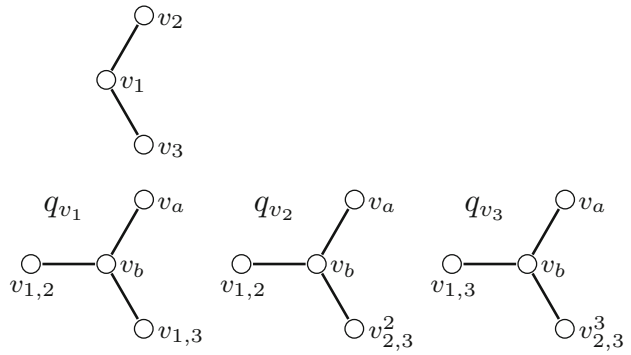

Fig. 18 An illustration of the query suggestions generated from an MIS instance

each vertex v in V, we have a query suggestion q_v corresponding to each v in V. We construct Q_V such that each query q_v in Q_V has exactly $|Q_V|$ edges. The structure of each query is a star, with a common first edge (v_a, v_b) and the other edges encode the following (also illustrated with Fig. 18):

1. If $(v_i, v_j) \in E$, then, (v_b, vi, j) is an edge of q_{v_i} and q_{v_j}. That is, mces between q_{v_i} and q_{v_j} is $\{(v_a, v_b), (v_b, vi, j)\}$.
2. Otherwise, an edge $(v_b, v^i i, j)$ is introduced to q_{v_i} and $(v_b, v^j i, j)$ is introduced to q_{v_j}. Then, the mces between q_{v_i} and q_{v_j} is $\{(v_a, v_b)\}$ only.

The maximum independent set is at most of the size $|V|/2$. Therefore, we invoke RSQ on Q_V, where k is ranged from 1 to $|V|/2$ and α is set to 0. That is, only the diversity component of the ranking function is considered.

Case (1) Suppose Q'_v is a solution of RSQ. If for some $q_i, q_j \in Q'_v$ and mces(q_i, q_j) is not just (v_a, v_b), then it guarantees there does not exists Q''_v such that $|Q'_v| = |Q''_v|$ and *for all* $q'_i, q'_j \in Q''_v$, such that mces(q'_i, q'_j) is just (v_a, v_b). This is because Q''_v would have been ranked higher, according to util (i.e., Q''_v is more diversified than Q'_v). And by the reduction above, there is an edge (v_i, v_j) in E. The corresponding V' is not an IS.

Case (2) Suppose Q'_v is a solution of RSQ and for all $q_i, q_j \in Q'_v$, mces(q_i, q_j) is (v_a, v_b). By the reduction above, there is no edge between v_i and v_j, for all v_i, v_j. Thus, the corresponding V' is an IS.

Putting these together, let Q'_v is the largest set returned by invoking RSQ's for k ranging from 1 to $|V|/2$, whose corresponding V' is an IS. Suppose Q''_V is returned by RSQ and larger than Q'_v. Then, Q''_V belongs to Case (1). According to Case (1), there is no IS of the size $|Q''_V|$ can be obtained. Therefore, V' is the maximum independent set. □

Appendix 3: Additional experiments

Suggestion qualities with different underlying definitions in AUTOG

Quality metrics under other mces distance metrics For illustration purposes, the paper adopts the *maximum common edge subgraph* (mces) for dist (see BS in Definition 8). It is possible to plug in other edge based distance metrics into the AUTOG framework (without modifications) to represent the "intra-dis- similarity" between a pair of suggestions. In this experiment, we report the quality metrics of the suggestions when AUTOG uses two other mces distance metrics.

The two distance metrics are presented in [22,36,38], denoted as WSKR and FV. The first distance metric (WSKR) uses the size of the union instead of the size of the larger graphs to distinguish variations in the graph sizes [38]:

$$\text{dist}_{WSKR}(g_1, g_2) = 1 - \frac{|\text{mces}(g_1, g_2)|}{|g_1| + |g_2| - |\text{mces}(g_1, g_2)|} \quad (1)$$

The second distance metric (FV) is based on the maximum common subgraph and minimum common supergraph of the two graphs. Therefore, it takes into consideration of both superfluous and missing structure information of the two graphs [36]. We normalize FV by dividing it by $|g_1| + |g_2|$.

$$\text{dist}_{FV}(g_1, g_2) = \frac{|g_1| + |g_2| - 2|\text{mces}(g_1, g_2)|}{|g_1| + |g_2|} \quad (2)$$

AUTOG achieved similar stable qualities under these different distance metrics and parameter settings. For presentation brevity, we reported the results of suggestion qualities in terms of #AUTOG and TPM only. Tables 12, 13 and 14 show that regardless the distance adopted, the suggestions were used in multiple iterations of query formulations. BS

Table 12 #AUTOG versus δ (PUBCHEM)

δ	#AUTOG (BS)	#AUTOG (WSKR)	#AUTOG (FV)
1	4.7	4.8	4.7
2	2.7	2.6	2.8
3	2.2	2.0	2.1
4	2.2	2.0	1.9
5	2.3	1.9	2.0

Table 13 #AUTOG versus $|q|$ (PUBCHEM when δ=3)

$\|q\|$	#AUTOG (BS)	#AUTOG (WSKR)	#AUTOG (FV)
8	2.2	2.0	2.1
12	3.3	2.8	3.0
16	4.0	3.4	3.4

Table 14 #AUTOG versus k (PUBCHEM when $\delta = 3$)

k	#AUTOG (BS)	#AUTOG (WSKR)	#AUTOG (FV)
4	1.5	1.6	1.5
6	1.8	1.7	1.9
8	2.0	1.9	2.0
10	2.2	2.0	2.1

Table 15 TPM versus δ (PUBCHEM)

δ	TPM (%) (BS)	TPM (%) (WSKR)	TPM (%) (FV)
1	59	60	59
2	52	51	52
3	45	41	44
4	42	41	40
5	43	39	42

Table 16 TPM versus $|q|$ (PUBCHEM when δ=3)

$\|q\|$	TPM (%) (BS)	TPM (%) (WSKR)	TPM (%) (FV)
8	45	41	44
12	44	42	42
16	42	38	39

Table 17 TPM versus k (PUBCHEM when $\delta = 3$)

k	TPM (%) (BS)	TPM (%) (WSKR)	TPM (%) (FV)
4	26	28	29
6	34	33	35
8	40	40	40
10	45	41	44

performed slightly better than FV and WSKR. Tables 15, 16 and 17 show similar trends from BS, FV and WSKR, while BS almost always performed the best. Therefore, users may pick the distance metric that is intuitive to their applications.

Suggestion qualities of the *c*-prime features on top of GINDEX In the paper, c-prime features are defined (Definition 7) with frequent features (Definition 12). As discussed, c-prime features are orthogonal to other features. As a proof of concept for integrating other features to AUTOG, we implemented c-prime features on top of discriminative features proposed in GINDEX–the seminal work of using features for subgraph query performance.

In a nutshell, we counted the composability of the discriminative features and index those that are c-prime. We used the implementation from IGRAPH [16]. We adopted the default parameter values for GINDEX. In particular, the support threshold was set to 10%, the maximum feature size $maxL$ was 10, and the discriminative ratio γ_{min} was 2. We

Table 18 Quality metrics versus δ (PUBCHEM, using GINDEX)

δ	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
1	11	0.15	0.15	2
2	56	0.74	1.43	21
3	58	0.62	1.63	25
4	55	0.56	1.49	23
5	51	0.54	1.46	22

Table 19 Quality metrics versus $|q|$ (PUBCHEM when δ=3, using GINDEX)

$\|q\|$	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
8	58	0.62	1.63	25
12	79	1.17	3.11	29
16	91	1.62	4.24	28

Table 20 Quality metrics versus k (PUBCHEM when $\delta = 3$, using GINDEX)

k	1 Hit (%)	#AUTOG	AUTOG $\|E\|$	TPM (%)
4	47	0.49	1.27	19
6	42	0.45	1.19	18
8	51	0.53	1.41	21
10	58	0.62	1.63	25

implemented the same size-increasing function as in [42]. Under this setting, we obtained 2370 discriminative features from the *PubChem* dataset. We then constructed the FDAG index as before.

We compared the suggestion qualities of AUTOG using the proposed c-prime features and c-prime features on top of discriminative features, simply denoted as GINDEX. We report the suggestion qualities from simulations in Tables 18, 19 and 20. The results showed that such suggestions were still somewhat useful. As expected, when they are compared to the results from Tables 4, 5 and 6, the proposed c-prime features gave clearly higher quality suggestions than those by using discriminative features. More specifically, when we varied δ, the average values of 1 Hit (%), #AUTOG, AUTOG $|E|$ and TPM (%) of our proposed c-prime features were 99%, 2.8, 3.6, and 48%. whereas those of GINDEX were 46%, 0.5, 1.2, and 19%. When we varied $|q|$, those quality metrics of the proposed features were 99%, 3.2, 5.0, and 44%, whereas those of GINDEX were 76%, 1.1, 3.0, and 27%. Similarly, when we varied k, those quality metrics of the proposed features were 93%, 1.9, 2.7, and 36%, whereas those of GINDEX were 49%, 0.5, 1.4, and 21%. The reason is simple. The design goal of GINDEX is to efficiently prune non-answer graphs of a query, not query autocompletion.

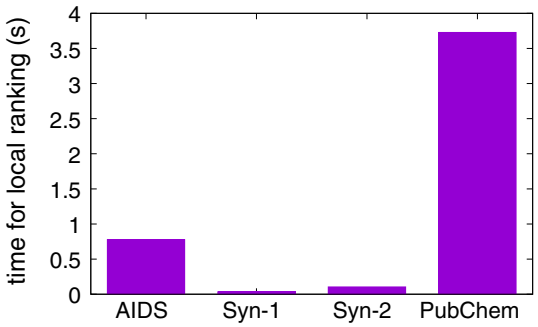

Fig. 19 Local—default

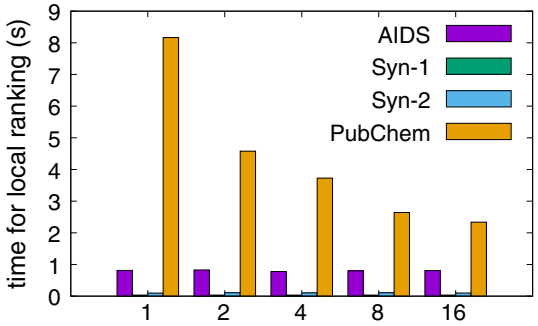

Fig. 20 Local—time versus m

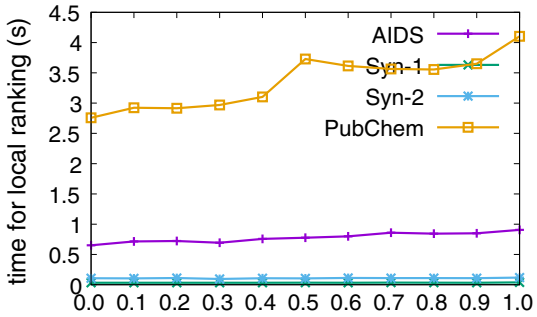

Fig. 21 Local—time versus γ

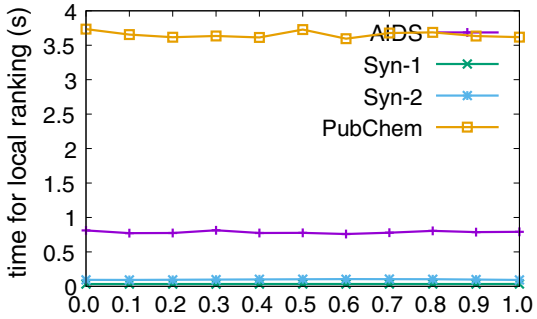

Fig. 22 Local—time versus α

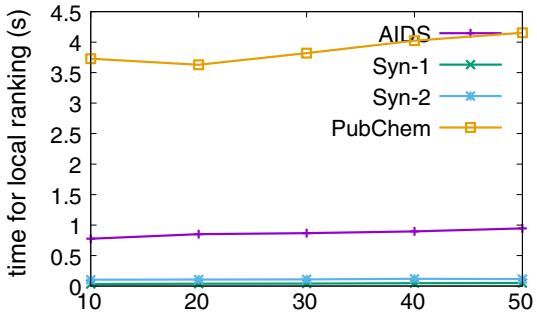

Fig. 23 Local—time versus k

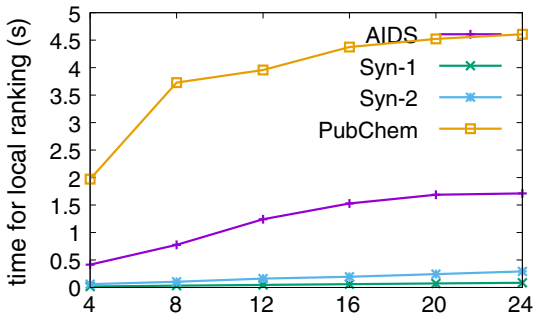

Fig. 24 Local—time versus $|q|$

Online performance breakdowns

Next, we present a detailed performance study of each major step of the online processing.

Query decomposition We report that the query decomposition phase always took less than a few milliseconds, for all queries and datasets under all the aforementioned parameter settings.

Local ranking The runtimes of local ranking phase under various parameter settings are presented in Figs. 19, 20, 21, 22, 23 and 24. When comparing Fig. 19 to Fig. 10, we observed that the local ranking was the bottleneck of online processing. The reason is that local ranking compared many pairs of candidate suggestions, even they were indexed by FDAG. Consistent results can be observed from experiments with the parameters m, γ, α and top-k were varied (e.g., Figs. 20, 11), except with the following situation. Under the default setting, the runtimes of the local part increased slightly with k as shown in Fig. 23. We also noted from Fig. 24 that the ART of the online processing for local ranking increased sub-linearly with $|q|$.

Global ranking The runtimes of the global ranking phase under a large variety of parameter settings are reported in Figs. 25, 26, 27, 28, 29 and 30. When varying m, γ and α, the times were quite stable. We noted from the experimental results that the global diversification took <60 ms under the default setting. Fig. 29 shows that its runtimes were roughly linear to the number of suggestions k. Fig. 30 however shows that the runtimes increased sharply as the query size $|q|$ increased. The two main reasons are: (1) When $|q|$

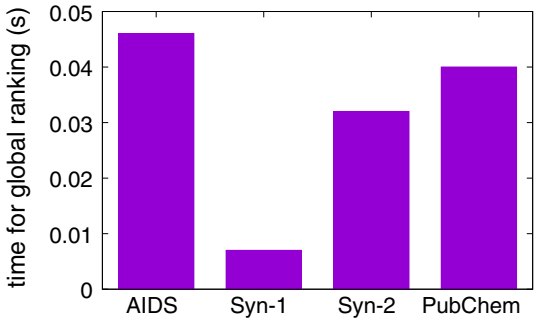

Fig. 25 Global—default

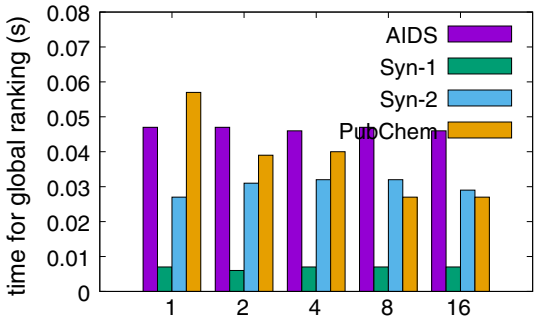

Fig. 26 Global—time versus m

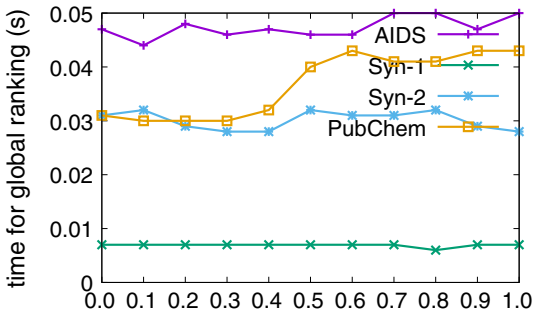

Fig. 27 Global—time versus γ

increased, the queries may be decomposed into *large* features. Computing mces of large features online is known to be costly, and (2) Large queries may be decomposed into *more* features, which in turn resulted in more mces calls.

From the last experiment, we found that the ART of online processing was determined by either local or global rankings, which are in turn dependent to $|q|$. We illustrate further the relations between the two by reporting performance breakdown on all datasets datasets (see Figs. 31, 32, 33, 34). For AIDS and PUBCHEM, we observed that the runtimes of global ranking increased much faster than those of the local one, due to the costly online mces computations. For SYN- 1 and SYN- 2, we observed the ARTs of the two phases exhibited linear trends.

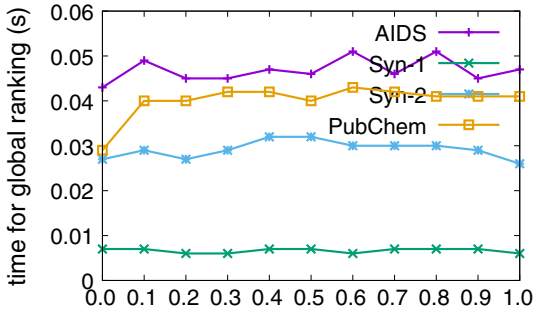

Fig. 28 Global—time versus α

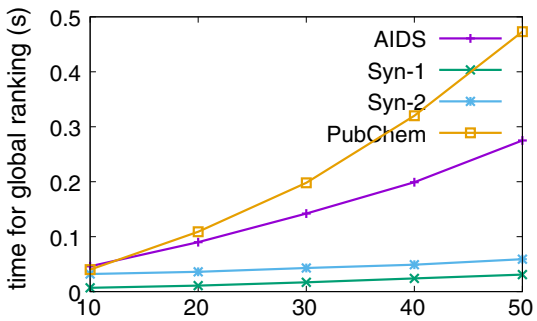

Fig. 29 Global—time versus k

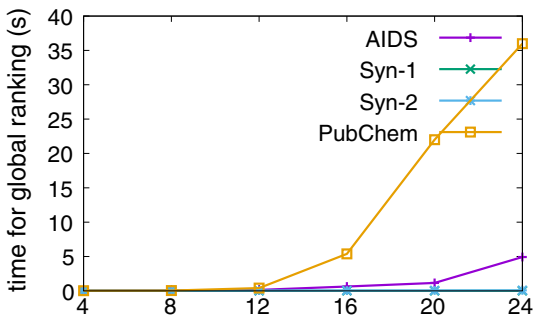

Fig. 30 Global—time versus $|q|$

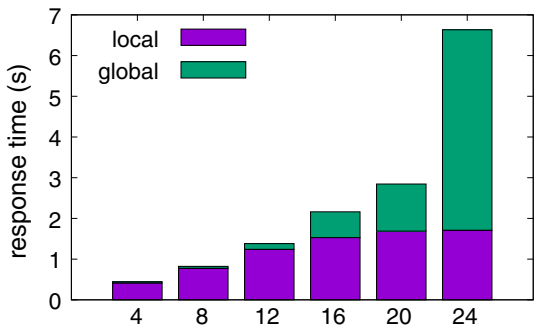

Fig. 31 Performance breakdown (AIDS)—vary $|q|$

Selectivity estimation We evaluated various facets of selectivity estimation in AUTOG. The results are summarized in Figs. 35, 36. We noted that the estimation results clearly formed three distinct categories: "*mis-estimated*" denotes

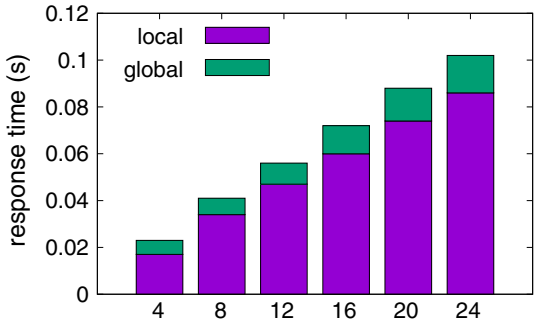

Fig. 32 Performance breakdown (SYN- 1)—vary $|q|$

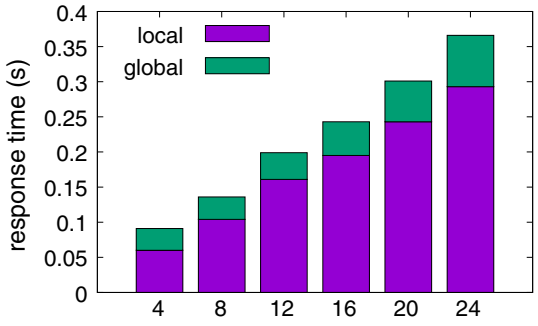

Fig. 33 Performance breakdown (SYN- 2)—vary $|q|$

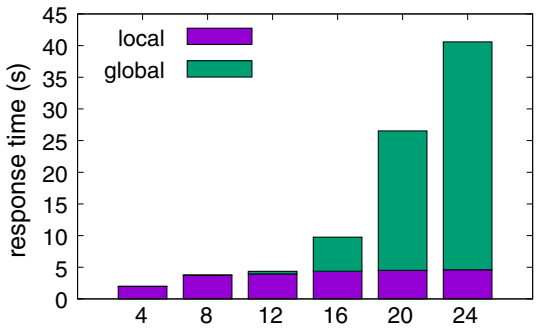

Fig. 34 Performance breakdown (PUBCHEM)—vary $|q|$

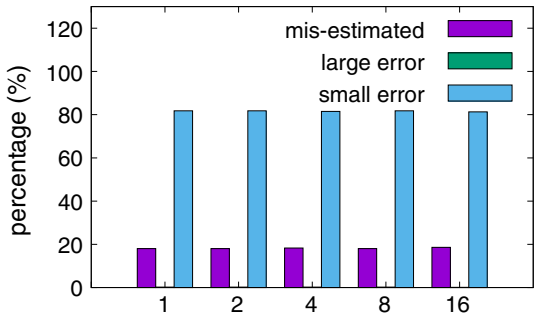

Fig. 35 Percentages of estimation presented in error categories (PUBCHEM)—vary m

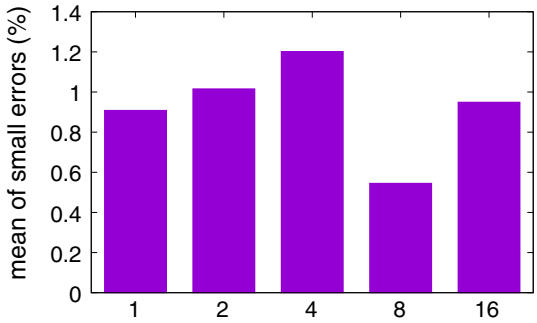

Fig. 36 The errors of the estimated in the "small errors" category (PUBCHEM)—vary m

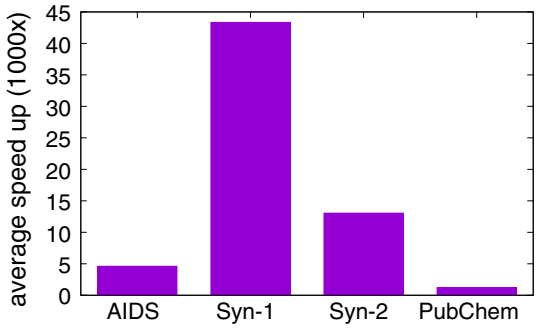

Fig. 37 Effects of the trimming technique on mces

queries that were estimated to be non-empty, but are in fact empty; "*large error*" denotes estimated queries whose errors were larger than 100%; and "*small error*" denotes the remaining queries. Fig. 35 shows the percentages for each of the three categories under different values of sampling step m. About 80% of the queries were estimated correctly. Fig. 36 reports that the mean estimation errors of "*small error*" were all below 1.2%. Thus, the selectivity estimation was accurate.

Effectiveness of structural trimming for mces Fig. 37 reports the average speedups due to the trimming technique introduced in Sect. 4.3.2 on all datasets. There were at least three orders of magnitudes of speedup. We remark that there were some large queries that could not be finished without trimming; these queries are excluded. Recall that mces was a performance bottleneck. With trimming techniques, we report in Fig. 38 that the costs of mces were around 30% (respectively, 60%) of global ranking costs for the synthetic datasets (respectively, the real datasets).

10 Appendix 4: The FDAG construction

In this appendix, we present the construction algorithm of FDAG (shown in Algorithm 3). The details of Algorithm 3 can be presented as follows. First, we sort F in ascending order of their edge numbers (Line 2). Then, we process the features one by one (Line 3). We create a node v_f in FDAG, for each $f \in F$, and compute the automorphism relation of f (A_f). For index nodes v_f and $v_{f'}$, we perform a subgraph

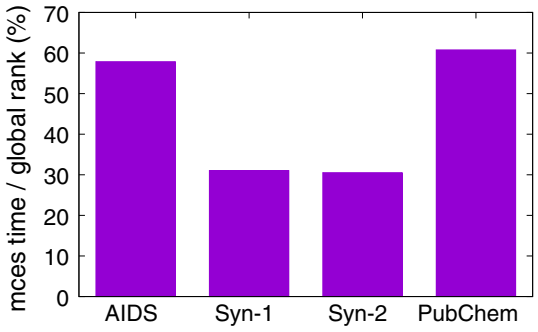

Fig. 38 mces cost in global ranking (with trimming)

Algorithm 3 Construct FDAG

Input: a graph database D, a feature set F, the max. increment size δ
Output: FDAG I

1: Initialize I as an empty directed graph
2: $[f_1, ..., f_n]$ = sort F by the no. of their edges, in ascending order
3: **for all** $f \in [f_1, ..., f_n]$ **do**
4: create an index node v for f
5: $I.V = I.V \cup \{v\}$
6: $I.A(f)$ = automorphism(f)
7: $I.D_f$ = eval(f, D)
8: **for all** $v' \in I.V$ **do**
9: **if** $f_{v'} \subseteq_\lambda f_v$ **then**
10: add $(f_{v'}, f_v)$ to $I.E$
11: add all the embeddings of $f_{v'}$ in f_v to $I.M_{v',v}$
12: construct the functions anc and des of I
13: ENUM_COMPOSITIONS(I, δ)
14: **return** I

isomorphism test between f and f' (Line 9). If subgraph isomorphism relations exist, we add the edge $(v_f, v_{f'})$ to FDAG, and associate the subgraph embeddings to the edge (Lines 10–11). Finally, we generate anc and des from the FDAG structure (Line 12). Next, Line 13 enumerates the possible compositions of a feature pair and determines intermediate results for computing structural difference between compositions (as presented in Algorithm 4).

Feature pair composition enumeration Algorithm 4 requires further elaboration. It takes FDAG and the maximum increment size as input, and outputs a set of feature pair compositions and some auxiliary data (i.e., f_{ij} in Line 5 and F_l in Line 12). We highlight some important steps before the details. (i) In Line 5, it shows that a query composition $(f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j)$ forms f_{ij}, and f_{ij} *itself can be a feature*. We record it in ζ; we count the occurrences of f_{ij} in all ζs and obtain f_{ij}'s composability. (ii) While f_{ij} may not be a feature, it may *contain* some features other than f_i and f_j. We record such features in F_l. It is known that in feature-based query processing, the more features (F_q) the query has, the more accurate the candidate query answer set (D_q) is, because $D_q = \cap_{f \in F_q} D_f$. (iii) The parameter δ is the threshold on the size increment in a query suggestion. The number of candidate suggestions increases exponentially with δ. Therefore,

Algorithm 4 Enumerate Feature Pair Compositions

Input: FDAG I and the max. increment size δ
Output: the ζ function of I

1: **function** ENUM_COMPOSITIONS(I, δ)
2: **for all** cs $\in F$ **do**
3: **for all** $f_i, f_j \in$ des(cs) and CHECKINCEMENT(f_i, f_j, cs, δ) **do**
4: **for all** λ_i, λ_j where cs $\subseteq_{\lambda_i} f_i$ and cs $\subseteq_{\lambda_j} f_j$ **do**
5: $f_{ij} \leftarrow$ compose(f_i, f_j, cs, λ_i, λ_j)
6: **if** ISEMPTY(f_{ij}, I) **then**
7: **continue**
8: **else if** ISFEATURE(f_{ij}, I) **then**
9: $\zeta = \zeta \cup (f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j, f_{ij}, \emptyset)$
10: **else**
11: F_l = FEATUREFORMED(f_{ij}, f_i, f_j, cs)
12: $\zeta = \zeta \cup (f_i, f_j, \mathsf{cs}, \lambda_i, \lambda_j, \phi, F_l)$
13: **for all** $c_i, c_j \in \zeta$ where $c_i.f_i = c_j.f_i$ **do**
14: $\eta(c_i, c_j)$ = mces_aux(c_i, c_j) //refer to Sec. 4.3.2
15: **return** ζ
16: **function** CHECKINCREMENT(f_i, f_j, cs, δ)
17: **return** $|f_i.E| - |\mathsf{cs}.E| \leq \delta$ and $|f_j.E| - |\mathsf{cs}.E| \leq \delta$

to ensure an interactive response, we may set a modest δ (e.g., its default is 5 in our experiments).

The details of Algorithm 4 can be described as follows. It starts the enumeration from a feature (cs), and composes large query graphs from two descendants (f_i and f_j) of cs (Lines 1–2). Line 3 checks if the increment is smaller than δ. In Line 4, we iterate through each embedding of cs in f_i and f_j, respectively. In Line 5, we compose a larger graph (denoted as f_{ij}) from f_i and f_j via cs. If f_{ij} is a feature, then Algorithm 4 just detected a possible way to compose f_{ij}. In Lines 6–7, we employ the techniques in Sect. 4.2 to prune the empty queries.

Lines 8–15 compute further information to optimize the ranking procedure (Sect. 4.3). Lines 8–9 check if f_{ij} is a feature. If yes, f_{ij} is recorded, for determining the composability of f_{ij}. Otherwise, Lines 10–12 determine if there are any features (other than f_i and f_j) embedded in f_{ij}, specifically, $F_l = \{f | \mathsf{cs} \subseteq_\lambda f \wedge f \subseteq_\lambda f_{ij}\}$, where $f \in F$. F_l are the set of features that contains in f_{ij}. As presented in Sect. 4.3.2, the selectivity of a query is estimated by the intersection of the candidate answers of the features of the queries. When F_l and f_{ij} are used in such estimation, it is more accurate than the estimation with only f_i and f_j. Lines 13–15 compute the auxiliary structures that optimize the online mces distance computation (Sect. 4.3.2).

References

1. Abiteboul, S., Amsterdamer, Y., Milo, T., Senellart, P.: Autocompletion learning for xml. In: SIGMOD (2012)
2. Bast, H., Weber, I.: Type less, find more: fast autocompletion search with a succinct index. In: SIGIR (2006)

3. Bhowmick, S.S., Choi, B., Zhou, S.: VOGUE: towards a visual interaction-aware graph query processing framework. In: CIDR (2013)
4. Bhowmick, S.S., Chua, H.-E., Thian, B., Choi, B.: ViSual: An hci-inspired simulator for blending visual subgraph query construction and processing. In: ICDE (2015)
5. Bhowmick, S.S., Dyreson, C.E., Choi, B., Ang, M.-H.: Interruption-sensitive empty result feedback: Rethinking the visual query feedback paradigm for semistructured data. In: CIKM (2015)
6. Borodin, A., Lee, H.C., Ye, Y.: Max-sum diversification, monotone submodular functions and dynamic updates. In: PODS (2012)
7. Braga, D., Campi, A., Ceri, S.: XQBE (XQuery By Example), A visual interface to the standard xml query language. In: TODS (2005)
8. Broder, A.Z.: On the resemblance and containment of documents. In: Compression and complexity of sequences (1997)
9. Bunke, H., Shearer, K.: A graph distance metric based on the maximal common subgraph. Pattern Recognit. Lett. **19**(3), 255–259 (1998)
10. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: towards verification-free query processing on graph databases. In: SIGMOD (2007)
11. Comai, S., Damiani, E., Fraternali, P.: Computing graphical queries over xml data. In: TOIS (2001)
12. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. In: PAMI (2004)
13. Fan, Z., Peng, Y., Choi, B., Xu, J., Bhowmick, S.S.: Towards efficient authenticated subgraph query service in outsourced graph databases. In: TSC (2014)
14. Feng, J., Li, G.: Efficient fuzzy type-ahead search in xml data. In: TKDE, pp. 882–895 (2012)
15. Gollapudi, S., Sharma, A.: An axiomatic approach for result diversification. In: WWW (2009)
16. Han, W.-S., Lee, J., Pham, M.-D., Yu, J.X.: iGraph: a framework for comparisons of disk-based graph indexing techniques. In: PVLDB, pp. 449–459 (2010)
17. Herschel, M., Tzitzikas, Y., Candan, K.S., Marian, A.: Exploratory search: New name for an old hat? http://wp.sigmod.org/?p=1183 (2014)
18. Hung, H.H., Bhowmick, S.S., Truong, B.Q., Choi, B., Zhou, S.: QUBLE: blending visual subgraph query formulation with query processing on large networks. In: SIGMOD, pp. 1097–1100 (2013)
19. Jayaram, N., Goyal, S., Li, C.: VIIQ: auto-suggestion enabled visual interface for interactive graph query formulation. In: PVLDB, pp. 1940–1951 (2015)
20. Jayaram, N., Gupta, M., Khan, A., Li, C., Yan, X., Elmasri, R.: GQBE: querying knowledge graphs by example entity tuples. In: ICDE (2014)
21. Jin, C., Bhowmick, S.S., Xiao, X., Cheng, J., Choi, B.: GBLENDER: towards blending visual query formulation and query processing in graph databases. In: SIGMOD (2010)
22. Kriege, N., Mutzel, P., Schäfer, T.: Practical sahn clustering for very large data sets and expensive distance metrics. J. Graph Algorithms Appl. **18**, 577–602 (2014)
23. Li, Y., Yu, C., Jagadish, H.V.: Enabling schema-free xquery with meaningful query focus. VLDB J. **17**, 355–377 (2008)
24. Lin, C., Lu, J., Ling, T.W., Cautis, B.: LotusX: a position-aware xml graphical search system with auto-completion. In: ICDE (2012)
25. Luks, E.M.: Isomorphism of graphs of bounded valence can be tested in polynomial time. J. Comput. Syst. Sci. **25**, 42–65 (1982)
26. Marchionini, G.: Exploratory search: from finding to understanding. Commun. ACM **49**, 41–46 (2006)
27. McGregor, J.J.: Backtrack search algorithms and the maximal common subgraph problem. Softw. Pract. Exp. **12**, 23–34 (1982)
28. Mottin, D., Bonchi, F., Gullo, F.: Graph query reformulation with diversity. In: KDD, pp. 825–834 (2015)
29. Nandi, A., Jagadish, H.V.: Effective phrase prediction. In: VLDB, pp. 219–230 (2007)
30. NCI. AIDS. https://dtp.cancer.gov/databases_tools/bulk_data.htm
31. NLM. PubChem. ftp://ftp.ncbi.nlm.nih.gov/pubchem/
32. Pandey, S., Punera, K.: Unsupervised extraction of template structure in web search queries. In: WWW, pp. 409–418 (2012)
33. Papakonstantinou, Y., Petropoulos, M., Vassalos, V.: QURSED: querying and reporting semistructured data. In: SIGMOD (2002)
34. Qin, L., Yu, J.X., Chang, L.: Diversifying top-k results. CoRR, arXiv:1208.0076 (2012)
35. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS (2002)
36. Venero, M.L.F., Valiente, G.: A graph distance metric combining maximum common subgraph and minimum common supergraph. Pattern Recognit. Lett. **22**, 753–758 (2001)
37. Vieira, M.R., Razente, H.L., Barioni, M.C.N., Hadjieleftheriou, M., Srivastava, D., Traina, C., Tsotras, V.J.: On query result diversification. In: ICDE (2011)
38. Wallis, W.D., Shoubridge, P., Kraetzl, M., Ray, D.: Graph distances using graph union. Pattern Recognit. Lett. **22**, 701–704 (2001)
39. Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., Sadakane, K.: Efficient error-tolerant query autocompletion. In: PVLDB (2013)
40. Xie, X., Fan, Z., Choi, B., Yi, P., Bhowmick, S.S., Zhou, S.: PIGEON: Progress indicator for subgraph queries. In: ICDE (2015)
41. Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining. In: ICDM, pp. 721–724, (2002)
42. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD (2004)
43. Yi, P., Choi, B., Bhowmick, S.S., Xu, J.: AutoG: A visual query autocompletion framework for graph databases. https://goo.gl/Xr9MRY (2016)
44. Yi, P., Choi, B., Bhowmick, S.S., Xu, J.: AutoG: a visual query autocompletion framework for graph databases [demo]. PVLDB **9**, 1505–1508 (2016)
45. Yuan, D., Mitra, P.: Lindex: a lattice-based index for graph databases. VLDB J. **22**, 229–252 (2013)