Data Science and Engineering (2022) 7:175–191
https://doi.org/10.1007/s41019-022-00182-8

RESEARCH PAPERS

# FLAG: Towards Graph Query Autocompletion for Large Graphs

Peipei Yi[1,2] · Jianping Li[2] · Byron Choi[2] · Sourav S. Bhowmick[3] · Jianliang Xu[2]

Received: 26 April 2021 / Revised: 29 January 2022 / Accepted: 12 February 2022 / Published online: 16 April 2022

**Abstract**
Graph query autocompletion (GQAC) takes a user's graph query as input and generates top-$k$ query suggestions as output, to help alleviate the verbose and error-prone graph query formulation process in a visual interface. To compose a target query with GQAC, the user may iteratively adopt suggestions or manually add edges to augment the existing query. The current state-of-the-art of GQAC, however, focuses on a large collection of small- or medium-sized graphs only. The subgraph features exploited by existing GQAC are either too small or too scarce in large graphs. In this paper, we present *Flexible* graph query autocompletion for LArge Graphs, called FLAG. We are the first to propose *wildcard labels* in the context of GQAC, which summarizes query structures that have different labels. FLAG allows augmenting users' queries with subgraph increments with wildcard labels to form suggestions. To support wildcard-enabled suggestions, a new suggestion ranking function is proposed. We propose an efficient ranking algorithm and extend an index to further optimize the online suggestion ranking. We have conducted a user study and a set of large-scale simulations to verify both the effectiveness and efficiency of FLAG. The results show that the query suggestions saved roughly 50% of mouse clicks and FLAG returns suggestions in few seconds.

**Keywords** Subgraph query · Query autocompletion · Large graphs · Database usability

## 1 Introduction

Researchers and practitioners perform different types of queries on large graphs [30]. Formulating subgraph matching query, among others, requires significant users' effort. A popular approach to provide query formulation aids for users is to build visual query interfaces (a.k.a GUIs) that facilitate the drawing of query graphs in an easy and intuitive manner. Real-world visual query interfaces (*e.g.*, PubChem[1], ChemSpider[2], and Scaffold Hunter [3]) have already been offered. However, composing graph queries in a visual environment may still be cumbersome. To alleviate the burden of visual graph formulation, *graph query autocompletion* (GQAC) [36, 37] has been proposed. Consider a scenario that a user formulates a target query graph $q_t$ iteratively via the GUI.

*Given an existing partially formulated query graph $q$, GQAC aims to suggest a subgraph increment $\Delta q$ to $q$ to form a query suggestion, such that the suggestion is closer to the target query $q_t$.*

Since users' intention is hard to predict, GQAC typically returns $k$ suggestions on a visual interface for users to choose from. An example of GUI, the user's current query, and suggestions of GQAC are shown in Fig. 1. We mimicked the example figure style of a related work [37] for presentation consistency.

Existing studies only consider the GQAC problem for large collections of small graphs, *e.g.*, chemical databases[4], and

✉ Peipei Yi
pyi@lenovo.com

✉ Byron Choi
bchoi@comp.hkbu.edu.hk

Jianping Li
csjpli@comp.hkbu.edu.hk

Sourav S. Bhowmick
assourav@ntu.edu.sg

Jianliang Xu
xujl@comp.hkbu.edu.hk

1 Machine Intelligence Center, Lenovo, Hong Kong, Hong Kong

2 Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong

3 School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

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

[4] http://autog.comp.hkbu.edu.hk:8000/autog/ http://autog.comp.hkbu.edu.hk:8000/gfocus/.

[2] http://www.chemspider.com/.

[3] http://scaffoldhunter.sourceforge.net/.

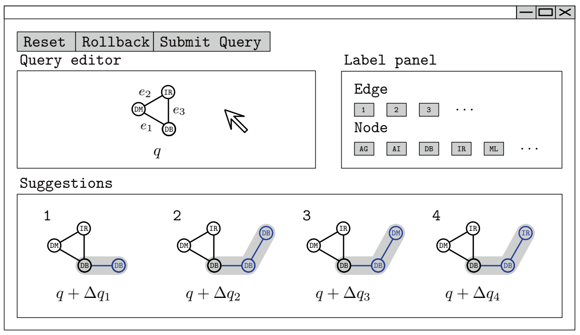


**Fig. 1** A typical GUI and query suggestions of GQAC

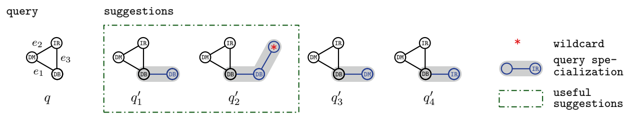


**Fig. 2** Example of query and the suggestions with wildcards

cannot be directly applied to large graphs. In particular, previous studies construct query suggestions based on some popular substructures (a.k.a *features*) of the graph data.[5] This assumes users want to construct queries to retrieve some graphs. For instance, we may set the minimum support of frequent substructures to 10% of the dataset size for PubChem and we obtained approximately a thousand features for GQAC. However, in large graphs, such features are smaller in size and more scarce in quantities. For example, such frequent subgraphs are very few in CiteSeer, reported fewer than 10 frequent subgraphs for various support threshold values [8]. This phenomenon leads to *two main challenges* of GQAC for large graphs. First, there are a large number of distinct subgraphs and each of them has small supports from the graph. Candidate suggestions generated from them are many but rare in the graph data. Further, the visual interface shows only $k$ suggestions, not to mention humans may interpret a small set of suggestions in practice. Such $k$ suggestions may not be useful. We illustrate the second challenge with Example 1.

***Example 1*** Suppose the current query is $q$. The first suggestion in Fig. 1 ($q + \Delta q_1$) increments $q$ by one edge, which may not save effort from query formulation. Then, consider the last three suggestions increment $q$ by two edges. They have however become *overly specific*, each of which appears only few times in the data graph. The three suggestions occupied relatively much area of the GUI. It is desirable to efficiently *summarize* the specific suggestions and rank the generalized one high and leave room for others.

To address the aforementioned challenges, we propose *Flexible graph query autocompletion for LArge Graphs (*FLAG*)*. To tackle the first challenge, we propose *wildcard label* to GQAC. A wildcard label represents any label of the data graph. It is suitable for GQAC for a large graph for two reasons. First, FLAG can then provide suggestions that contain wildcard labels. An example is shown as $q'_2$ of Fig. 2. $q'_2$ summarizes suggestions $q + \Delta q_2$, $q + \Delta q_3$, and $q + \Delta q_4$ of Fig. 1. It is evident that $q'_2$ summarizes (or generalizes) the three suggestions, each of which has only few support from the graph data, and spare some space of the top-$k$ suggestions for others. Second, wildcards can be naturally used when users are not sure about the labels of the nodes/edges of the query graph, but FLAG still suggests new edges. To avoid having wildcards appearing in arbitrary places of query suggestions, we propose well-formed suggestions.

To address the second challenge, we introduce *query generalization* and *query specialization* of suggestions for GQAC. *Query specialization* is an operator for augmenting an existing query to one that is closer to the target query. It also quantifies how much a suggestion augments the existing query. In each specialization, the user either i) add a wildcard edge or ii) change a wildcard label to an exact label. The introduction of wildcard label does not alter the asymptotic complexity of the query process (*e.g.*, subgraph matching) or graph autocompletion (*e.g.*, suggestions ranking). Next, we propose *query generalization* which is the opposite of query specialization. Recall from Fig. 2, three suggestions are generalized into one so that the support of the generalized suggestion becomes higher. $q'_2$ is more specific than $q + \Delta q_1$ but more generalized than $q + \Delta q_2$, $q + \Delta q_3$ and $q + \Delta q_4$.

Wildcard-enabled GQAC may generate numerous candidate suggestions and their ranking can be inefficient. We propose a novel linear submodular ranking function that involves not only query suggestion's specialization to the current query but also the summarization of the possible candidate suggestions. Specifically, we propose *specialization value* (SP) to quantify how much a suggestion augments the existing query; and *summarization value* (SM) to quantify how many candidate suggestions a suggestion summarizes. The approximation of the ranking function is differentiable. Hence, we can adopt a stochastic gradient descent algorithm to learn the parameters of the ranking function. It is also not surprising that the ranking problem is NP-hard. Since the ranking function is submodular, we propose an efficient greedy algorithm for computing the top-$k$ suggestions. To further optimize efficiency, we extend an existing index with the support of wildcards for ranking.

[5] When query logs (which are also graphs) are available, GQAC may generate suggestions from them also.

**Table 1** Frequently used notations

| Symbol | Meaning |
|---|---|
| $q$ | The current query or existing query |
| $q'$ | A query suggestion or simply suggestion |
| $\Delta q$ | Query increment (adding $\Delta q$ to $q$ yields $q'$) |
| $Q'$ | Query suggestions |
| $q \subseteq_{\lambda} q'$ | $q$ is a subgraph of $q'$ and $\lambda$ is the embedding of $q$ in $q'$ |
| $f$ | A (proper) connected subgraph of the query $q$ |

In conclusion, this paper makes the following contributions.

1. We propose *wildcard labels* for query graph and query suggestions. We propose a notion of well-formed wildcard graph for GQAC.
2. We propose *specialization value* (SP) and *summarization value* (SM) to measure how much a suggestion specializes an existing query and summarizes other candidate suggestions.
3. We propose a *ranking function* based on SP and SM.
4. To optimize the efficiency of *online* ranking of query suggestions, we present the techniques that are needed to extend an existing index for the wildcard-enabled GQAC.
5. We use a stochastic gradient descent algorithm to learn the parameters of the ranking functions in experiments. We investigate the usefulness and efficiency of FLAG via a user study and extensive simulations. The results show that FLAG saves about 50% of mouse clicks in query formulations and the suggestions are returned in several seconds under a large variety of settings.

The rest of the paper is organized as follows. Section 2 provides the background of GQAC. Section 3 proposes wildcard labels for GQAC. Section 4 proposes specialization value (SP) and summarization value (SM) for query suggestions. Section 5 provides details of the efficient online suggestion ranking. We present a performance study in Sect. 6. We discuss the related work in Sect. 7. Section 8 concludes the paper and presents some future work.

## 2 Preliminaries

This section provides the preliminaries of graph query autocompletion (GQAC) and presents the problem being studied. Some frequently used notations are listed in Table 1.

### 2.1 Background on Graph Query Autocompletion (GQAC)

#### 2.1.1 Graph Data

We consider a single large graph $G = (V, E, l)$, consists of a set of nodes $V$, a set of edges $E$ and a labeling function $l$ that assigns labels to nodes and edges. The size of a graph is defined by $|E|$. deg is a function that returns the degree of a vertex. For example, Fig. 1 shows the CiteSeer graph. Node labels represent the area of the publication (*e.g.*, `DB`, `DM`, `IR`) and edge labels represent the distance between the pair of publications. This dataset will be used for subsequent examples. For presentation simplicity, all examples illustrate undirected graphs with a single label for each node and edge.

#### 2.1.2 Query Formalism

This paper adopts subgraph isomorphism, a popular and fundamental query formalism, for the technical discussions. The subgraph isomorphism is recalled below.

**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 function (or *embedding*[6]) $\lambda : V \mapsto V'$ such that

1. $\forall u \in V, \lambda(u) \in V'$ such that $\mathsf{match}(l(u), l'(\lambda(u))) = \mathtt{true}$; and
2. $\forall (u, v) \in E$, $(\lambda(u),\lambda(v)) \in E'$ such that $\mathsf{match}\ (l(u, v), l'(\lambda(u), \lambda(v))) = \mathtt{true}$,

where $\mathsf{match}(l_1, l_2) = \mathtt{true}$ *iff* $l_1 = l_2$

Multiple subgraph isomorphic embeddings of $g$ may exist in $g'$, denoted as $\lambda^0_{g,g'}, \lambda^1_{g,g'}, \ldots, \lambda^m_{g,g'}$. For succinct presentation, we refer to each $\lambda^i_{g,g'}$ as an *embedding* $\lambda$, when the subscripts and superscripts are clear from or irrelevant to the context.

**Definition 2** (Subgraph query) Given a single large graph $G$ and a query graph $q$, the answer (or result set) of $q$ is $G_q = \{\lambda \mid q \subseteq_{\lambda} G\}$.

#### 2.1.3 Visual Graph Query Construction

Graphs and their query graphs can be intuitively displayed and drawn in a visual environment (*e.g.*, a GUI Fig. 1). In the process of visual query construction, the user draws the current query $q$ on the query editor and has the target query $q_t$ in

[6] Note that this is different from the embedding that maps a graph into a $d$-dimensional space, in the context of machine learning.

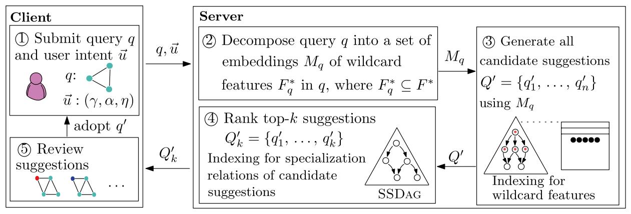

Client
① Submit query $q$ and user intent $\vec{u}$
$q$:
$\vec{u}: (\gamma, \alpha, \eta)$
adopt $q'$
⑤ Review suggestions
$q, \vec{u}$
$Q'_k$
Server
② Decompose query $q$ into a set of embeddings $M_q$ of wildcard features $F^*_q$ in $q$, where $F^*_q \subseteq F^*$
$M_q$
③ Generate all candidate suggestions $Q' = \{q'_1, \ldots, q'_n\}$ using $M_q$
④ Rank top-$k$ suggestions $Q'_k = \{q'_1, \ldots, q'_k\}$
Indexing for specialization relations of candidate suggestions
SSDAG
$Q'$
Indexing for wildcard features

Fig. 3 FLAG: graph query autocompletion for large graphs

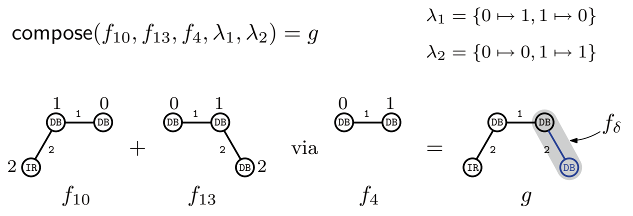

$\text{compose}(f_{10}, f_{13}, f_4, \lambda_1, \lambda_2) = g$
$\lambda_1 = \{0 \mapsto 1, 1 \mapsto 0\}$
$\lambda_2 = \{0 \mapsto 0, 1 \mapsto 1\}$
via
$f_\delta$
$f_{10}$
$f_{13}$
$f_4$
$g$

Fig. 4 An illustration of query composition—forming a large query graph from small graphs

mind; and he/she performs an action (*e.g.*, adding an edge or subgraph to $q$) to make the current query closer to the target. Performing this process manually can be error-prone.

### 2.1.4 Graph Query Autocompletion (GQAC)

A visual environment often provides visual aids for query construction, in addition to the basic constructs (*e.g.*, as shown node and edge labels in the label panel of Fig. 1). Recently, Yi et al. [36] propose GQAC that aims at alleviating users from the cumbersome actions by providing useful subgraph suggestions. The process of GQAC is sketched in Fig. 3. Here, we present its major steps. The details related to FLAG are postponed to later sections.

1. GQAC takes the user's current query $q$ and the user preference of suggestions as input. Voluminous candidate query suggestions are generated and ranked. A query suggestion is a graph that augments the current query with structure and/or labels. Note that the increments to the query can be subgraphs. A small set of ranked query suggestions are efficiently generated for the user's review.
2. The user may compose the query by either adopting a suggestion or manually adding other edges.
3. The above steps are repeated until the target query is constructed.

### 2.1.5 Formalizing GQAC

Recall that query suggestions are formed by incrementing the current query with a subgraph. The current state-of-the-art of GQAC [36] exploits the concepts of *graph features* (or simply features). Graph features are generally understood as subgraphs that carry important characteristics of graph data. Features have also been considered the *tokens* of GQAC. For example, an existing work of GQAC[36] decomposes the current query into a set of features and augments the current query with another feature to form a query suggestion. The intuition is that users may want to specify some characteristics of the graph in their target queries. While existing work uses $c$-prime features as the features for GQAC, other features can be plugged into GQAC, depending on the users' applications.[7]

The composition of two subgraphs (incrementing a subgraph with another) can be intuitively understood as a one-step construction of a query suggestion, which can be formally defined as a compose function. We recall some relevant definitions below.

**Definition 3** (Common subgraph (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 $\text{cs}(g_1, g_2)$, or simply cs when $g_1$ and $g_2$ are clear from the context), *i.e.*, $\text{cs} \subseteq_{\lambda_1} g_1$ and $\text{cs} \subseteq_{\lambda_2} g_2$, for some $\lambda_1$ and $\lambda_2$.

**Definition 4** (compose for query composition) compose [36] is a function that takes two graphs, $g_1$ and $g_2$, and the corresponding embeddings ($\lambda_1$ and $\lambda_2$) of a common subgraph cs as input, and returns the graph $g$ that is composed by $g_1$ and $g_2$ via $\lambda_1$ and $\lambda_2$ of cs, respectively, denoted by $g = \text{compose}(g_1, g_2, \text{cs}, \lambda_1, \lambda_2)$.

***Example 2*** An example of query composition is shown in Fig. 4. Assume that $f_{10}$ is the current query and $f_{13}$ is the graph feature used to increment $f_{10}$. Then, $g$ is the *query graph* formed by adding $f_{13}$ to $f_{10}$ via the common subgraph $f_4$, *i.e.*, $g = \text{compose}(f_{10}, f_{13}, f_4, \lambda_1, \lambda_2)$. The *increment* is highlighted in blue with the gray background. The embeddings $\lambda_1$ and $\lambda_2$ specify the locations of $f_4$ in $f_{10}$ and $f_{13}$, respectively.

**Definition 5** (Useful suggestion) Given a target query $q_t$ and existing (or current) query $q$, a query suggestion $q'$ is *useful if and only if* $q \subset_{\lambda_1} q'$ and $q' \subseteq_{\lambda_2} q_t$, for some $\lambda_1$ and $\lambda_2$.

[7] We remark that existing GQAC systems do not rank *infrequent* items of a database high, if at all. In the context of web search, some infrequent phrases are also not suggested. A possible reason is that users can simply run those queries without further constructing them and manually examine the few results to obtain their desired information.

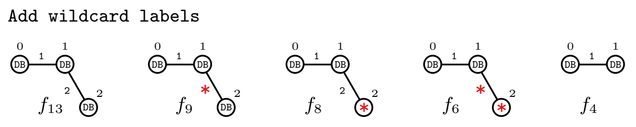


**Fig. 5** Adding wildcard labels to features

As motivated, users' target queries are hard to predict. Recently, GQAC systems have proposed various *ranking* mechanisms (according to users' preferences and a ranking function util) to efficiently compute a small list of suggestions with the hope that they are useful. Some ranking factors include the result counts of the suggested queries and the structural diversity of the suggestions. It is not surprising that the suggestion ranking problems are generally intractable and hence, greedy algorithms have been proposed to efficiently rank the useful query suggestions.

#### 2.1.6 Problem Statement

Given a large graph $G$, an existing query $q$, a ranking function util, a user preference $\mathbf{u}$ and a parameter $k$, the paper investigates to return query suggestions $Q'_k : \{q'_1, q'_2, \ldots, q'_k\}$ 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 $\mathbf{u}$.

*To the best of our knowledge, this paper is the first work that computes query suggestions for querying a single large graph and wildcards for* GQAC *have not been proposed before.*

## 3 Wildcard Labels for GQAC

In this section, we propose wildcard labels to generalize similar substructures into a summary structure. We further discuss how to introduce wildcard labels to the process of GQAC (*e.g.*, graph features and query compositions).

### 3.1 Wildcard Labels and Graphs

A wildcard label (or simply *wildcard*) represents any possible labels of nodes/edges and is assigned to new unlabeled nodes and edges by default, meaning that the labels are not yet specified. Figure 5 shows an ordinary feature ($f_{13}$) and features having a wildcard label of an edge ($f_8$ and $f_9$). The query formalism of subgraph isomorphism can be readily extended with wildcards by simply replacing the matching function match of Def. 1 with match$^*$, where match$^*$($l_1$,$l_2$)=true *iff* $l_1$="*" or $l_1$=$l_2$.

**Definition 6** (Wildcard graph) A graph with wildcard labels "*", denoted as $G^*$,[8] is defined as a 3-ary tuple ($V$,$E$,$\ell^*$), where $V$ and $E$ are node and edge sets and the label function $\ell^*$ that assigns ordinary labels or "*" to a node or edges.

A wildcard can be introduced to query graphs manually by users or suggested by GQAC. When introducing the wildcards to GQAC, the features to be added to an existing query must allow wildcards. However, this leads to an exponential blowup in the number of features used in existing GQAC for constructing query suggestions. Having too many wildcards in queries or suggestions is not only computationally costly to generate and rank but also confuses the users. Furthermore, the suggestions having wildcards can be neither too generic nor too specific with respect to the closest ordinary suggestion. To this end, we restrict the wildcards only occur at the leaf nodes/edges only (see Def. 7). Hence, users may often expand their query graphs at the boundaries.

**Definition 7** (Well-formed wildcard graph) A graph $G^*$ is a *well-formed* wildcard graph if it is a wildcard graph and all wildcard labels are on one leaf edge and the incident leaf node.

### 3.2 Wildcard Features for GQAC

While wildcards may still significantly increase the number of features, and hence, query suggestions, not every wildcard feature is useful. Consider an extreme case, where two frequent features $f$ and $f^*$ of the same size have the same result set, *i.e.*, $f^* \subseteq_\lambda f$, $|f^*| = |f|$ and $D_{f^*} = D_f$, it is not necessary to consider $f^*$ in GQAC. Among wildcard features and ordinary features with the same result set, it is sufficient to increment the existing query with the ordinary feature. Thus, such $f^*$ can be omitted from GQAC. Recall that GQAC generates query suggestions by adding a feature from a feature set to the existing query. We propose *independent wildcard features* such that the features retrieve different results from data.

**Definition 8** (Independent Wildcard Feature (IWF)) A wildcard feature $f^*$ is independent w.r.t a feature set $F$ if

1. There exists $F_1 \subseteq F$ and for $f_1 \in F_1$ such that $f_1 \subseteq_\lambda f^*$ and $|f^*| = |f_1| + 1$;
2. There exists $F_2 \subseteq F$ and for $f_2 \in F_2$ such that $f^* \subseteq_\lambda f_2$ and $|f^*| = |f_2|$; and

[8] We use $G^*$ to denote a graph having wildcards but may omit "*" when they are not relevant to the discussion.

3. $\frac{D_{f_1}}{D_{f^*}} \geq \phi$ and $\frac{D_{f^*}}{D_{f_2}} \geq \phi$ for $\forall f_1 \in F_1$ and $f_2 \in F_2$, where $\phi$ is a constant called *independent ratio*.

The independent ratio has the following properties: i) $\phi \geq 1$; and ii) the larger value of $\phi$, the more difference in the feature result sets and intuitively, more independent the wildcard features with respect to their closest ordinary features $F_1$ and $F_2$. A feature $f^*$ is introduced to GQAC if it is dependent enough from $F$.

The detailed process of generating independent well-formed wildcard features for graph query autocompletion is presented in Algo. 1. We adopt existing studies of feature mining [35] to obtain a set of features $F = \{f_1, f_2, \ldots, f_n\}$. Then, we add wildcard labels one by one to $f$ to obtain wildcard features $F^*$, that are both independent and well-formed. Applying the concepts introduced in Def. 8 and 7, we iteratively generate all wildcard features by substituting labels on one leaf edge with wildcards. Meanwhile, we eliminate the wildcard features that are dependent to existing ordinary features.

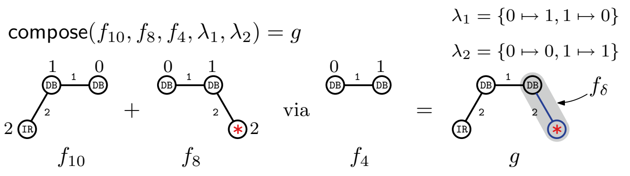
**Algorithm 1** Generating independent wildcard features

```
Input: Feature set $F$ (determined by existing algo, e.g., GSPAN [35])
Output: Wildcard feature set $F^*$
 1: Initialize $F^*$ as $F$
 2: for all $f \in F$ do
 3:     for all $v \in f.V$, where $\deg(v, f) = 1$ do    // $v$ is a leaf node
 4:         Let $e$ be the leaf edge incident to $v$, i.e., $e = (u, v)$, $v, u \in f.V$
 5:         Let $WF^*$ be the well-formed wildcard features by substituting the labels of $v$ or $e$ with wildcards
 6:         for all $f^* \in WF^*$ do
 7:             Let $F_1$ and $F_2$ be the ordinary features that satisfy 1) and 2) of Def. 8
 8:             if $f^*$ is independent with respect to $F_1$ and $F_2$ then
 9:                 $F^* \leftarrow F^* \cup \{f^*\}$
10: return $F^*$
```

***Example 3*** We illustrate the process of adding wildcard labels to features with Fig. 5. Given an ordinary feature $f_{13}$, and edge DB-DB connects leaf node DB. We replace the labels on edge DB-DB with wildcard labels to obtain wildcard features $f_9$, $f_8$, and $f_6$, which can be regarded as generalizing the labels on the edge DB-DB with wildcard labels.

### 3.3 Composition of Well-Formed Wildcard Features

The features discussed earlier can be the tokens for query autocompletion. Suggestions with wildcards are constructed by adding a feature to an existing query graph. The query composition (a one-step query suggestion construction Def. 4) can be readily extended. Given a query composition compose($g_1$, $g_2$, cs, $\lambda_1$, $\lambda_2$), $g_1$ and $g_2$ could be wildcard features and compose is restricted to return a well-formed suggestion.



$\mathsf{compose}(f_{10}, f_8, f_4, \lambda_1, \lambda_2) = g$
$\lambda_1 = \{0 \mapsto 1, 1 \mapsto 0\}$
$\lambda_2 = \{0 \mapsto 0, 1 \mapsto 1\}$
DB
IR
+
via
=
$f_{10}$
$f_8$
$f_4$
$g$
$f_\delta$

**Fig. 6** An illustration of wildcard composition

***Example 4*** Recall the query composition in Example 2 with compose($f_{10}$, $f_{13}$, $f_4$, $\lambda_1$, $\lambda_2$). We add wildcard labels into $f_{13}$ in Example 3 and obtain wildcard features $\{f_9, f_8, f_6\}$. A wildcard composition can be obtained by simply substituting $f_{13}$ of the composition with any of the wildcard features. One of the wildcard compositions, *i.e.*, compose($f_{10}$, $f_8$, $f_4$, $\lambda_1$, $\lambda_2$), is illustrated in Fig. 6.

## 4 Query Specialization and Query Summarization

The previous sections presented the features and their composition. In this section, we formalize query specialization for modeling the whole query suggestion construction process. We propose the specialization value (SP) to quantify how a query graph is specialized from an empty graph, and summarization value (SM) to quantify how one wildcard query suggestion summarizes other suggestions.

### 4.1 Query Specialization

#### 4.1.1 Specialization Order ($\prec$)

Specialization order is a partial order defined between two query graphs. The intuition is that a more specialized query is closer to the target query. It also models one query is constructed from the other. We formally define the specialization operators and specialization order as follows.

The *specialization operators* are the following two:

1. add($q$, $e$:($u$, $v$)): add a new edge $e$, where $\ell(e)$ is a "*" label, if $u$ and $v$ are existing nodes; and $\ell(v)$ is a "*" label, if $v$ is a new node.
2. replace($q$, $e$): replace a "*" label with a specific label of the edge or node of $e$.

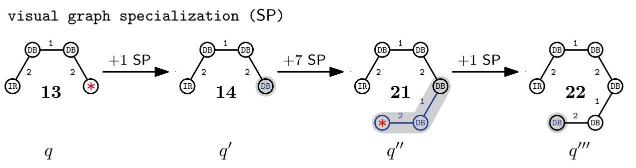


Fig. 7 An illustration of specialization orders and values

**Definition 9** (Specialization order ($\prec$)) Given two query graphs $q = (V, E, l)$ and $q' = (V', E', l')$, $q'$ specializes $q$, denoted as $q \prec q'$ *iff* there is an injective (or embedding) function $\lambda : V \rightarrow V'$ such that $q \subseteq_\lambda q'$.

#### 4.1.2 Specialization Value (SP)

To further measure the different degree of specialization of query graphs, we propose specialization value based on the specialization operators, in Def. 10. In addition, given a suggestion to an existing query, the difference of their specialization values captures how much does the suggestion augment the query. For simplicity, Def. 10 assumes that all operators are equal.

**Definition 10** (Specialization value (SP)) The specialization value of a query graph $q$ is the number of specialization operators needed to formulate $q$ from an empty graph $q_\emptyset$, denoted as $\mathsf{SP}(q)$.

***Example 5*** We illustrate specialization order and specialization value with Fig. 7. The specialization order of the query graphs is $q \prec q' \prec q'' \prec q'''$. The existing query is $q$ (leftmost) with a wildcard. The specialization value of $q$ is 13, indicated in bold at the center of $q$. After specializing the wildcard of $q$ into the label DB, the user obtains $q'$ with a specialization value increased by 1. Then, the user adopts a suggestion with a wildcard to get $q''$, with specialization value increased by 7. At last, the user specializes the wildcard to a specific label and obtains the target query $q'''$.

### 4.2 Query Summarization

#### 4.2.1 Summarization Set (SM)

To model how likely a query $q$ is useful to the user, we compute how many suggestions can be specialized from $q$. We formally define the summarization set to denote such suggestions.

**Definition 11** (Summarization set (SM)) The summarization set of a query $q$, denoted as $\mathsf{SM}(q)$, contains all query graphs that specialize $q$.

$$\mathsf{SM}(q) = \{q' | q \prec q', G_{q'} \neq \emptyset\},$$

where $G_{q'}$ is the subgraph query results set of $q'$. In other words, $q$ summarizes all the query graphs that specialize $q$. The summarization of a set of graphs $Q$ is as follows.

$$\mathsf{SM}(Q) = \bigcup_{q \in Q} \mathsf{SM}(q).$$

***Example 6*** Continuing with Fig. 7, given four query graphs, the specialization order of the query graphs is $q \prec q' \prec q'' \prec q'''$. Then, $\mathsf{SM}(q) = \{q, q', q'', q'''\}$, and $\mathsf{SM}(q') = \{q', q'', q'''\}$.

When the user formulates the query graph, both the number of query results and the possible suggestions are decreasing. This property (see Prop. 1) can be used to reduce the number of candidate suggestions for efficient GQAC. In particular, if $g$ is an answer for query $q'$, then $g$ is an answer for every query $q$ that summarizes $q'$. On the other hand, if $g$ is not an answer for $q$, then $g$ is not an answer for every query $q'$ that specializes $q$. This is formally described as follows.

**Proposition 1** *Given two query graphs $q$ and $q'$, where $q'$ specializes $q$ (i.e., $q \prec q'$) via a series of specialization operators. Then, $q \prec q' \Rightarrow \forall\, g' \in G_{q'}, \exists\, g \in G_q$ s.t. $g \subseteq_\lambda g'$.*

## 5 Autocompletion Framework for Large Graphs

The overall query autocompletion is presented in Algo 3 and illustrated with Fig. 3. FLAG assumes ① the user submits a query and an intent, and ② the query is decomposed into a set of embeddings of wildcard features of the data graph. FLAG then supports wildcards in two main steps of GQAC. First, in the candidate generation step, ③ we determine possible candidate suggestions, *i.e.*, the well-formed wildcard features to attach to the current query to form suggestions that may yield non-empty answers. In Sect. 5.1, we propose pruning techniques for large graphs and sampling techniques. Second, in Sect. 5.2, ④ we present a new ranking function that combines the specialization value and summarization set size.

### 5.1 Candidate Suggestions Generation

#### 5.1.1 Query decomposition

During the online autocomplete, the query decomposition procedure (Algo. 1 from AutoG[36]) is adopted. The query graph $q$ is decomposed into a feature set $F_q^*$, along with the embeddings of the features in the query. The detailed process is presented in Algo. 2.

To generate well-form query suggestions, where the wildcards appear in the leaf nodes/edges, the non-leaf wildcards (if any) in $F^*_q$ need to be specialized before generating candidate suggestions (Lines 3-9).

**Algorithm 2** Query decomposition

```
Input: a wildcard query $q$, wildcard feature set $F^*$, feature set $F$ and
       user preference component $\gamma$
Output: a set of embeddings $M_q$ of features $F^*_q$ in $q$
 1: $M^*_q$ = greedy_decomp($q$, $F^*$, $\gamma$)   // decompose $q$ with wildcard features [36]
 2: Let $M_q$ be an empty set   //initialize the embeddings of $F^*_q$ as $M_q$
 3: for all $(f^*, \lambda) \in M^*_q$ do   // for each embedding of $f^*$ in $q$:
 4:     if $* \in l(f^*)$ then   // $f^*$ is a wildcard feature
 5:         for all $f \in F$ do
 6:             if $|f^*| = |f|$ and $f^* \prec f$ then
 7:                 $M_q \leftarrow M_q \cup \{(f, \lambda)\}$   // materialize $f*$
 8:     else   // $f^*$ is an ordinary feature
 9:         $M_q \leftarrow M_q \cup \{(f^*, \lambda)\}$
10: return $M_q$
```

### 5.1.2 Non-empty candidate suggestions

Candidate suggestions can specialize the existing query in multiple ways. First, suggestions can replace wildcards in the query with specific labels. Second, candidates can increment the query with (wildcard) features. Specifically, given a set of features, the number of possible candidates is, in the worst case, exponential to the query and feature sizes. However, many of the composed queries may not make sense, when the composed queries do not retrieve any results from the underlying data graph. Such queries are known as *empty queries*. Furthermore, the problem of deciding the emptiness of a subgraph matching query is NP-hard.

Existing work [36] has proposed a *necessary* condition for compositions of non-empty query candidates. It has been reported that the condition reduced 13% and 45% of query compositions for AIDS and PUBCHEM, which consist of a large collection of modest-sized graphs. When directly applied, [36] prunes only 0.1% of the possible compositions of the CITESEER dataset. Therefore, in Prop 2, we propose a necessary condition for non-empty query compositions based on the large graph and sampling techniques.

We illustrate how to efficiently prune empty compositions using the embedding information. *The queries that are not pruned are considered candidate suggestions.*

Consider a large graph $G$, a set of sampled graphs $D$ obtained from $G$ using existing graph sampling techniques (Sect. 6), and the set of frequent features $F$ extracted from $D$ using existing frequent subgraph mining techniques offline. The embeddings $M_f$ of the features in the sampled graphs are and the embeddings $M_g$ of sampled graphs in the large graph can be computed offline. For a composition compose($f_1$,$f_2$, cs, $\lambda_1$, $\lambda_2$), the embeddings of $f_1$ and $f_2$ in the large graph $\lambda_{f_1,G}$, $\lambda_{f_2,G}$ are obtained using $M_f$ and $M_g$.

**Proposition 2** *A query $q$ is a non-empty query of the sampled graphs only if for each query composition* compose( $f_1$, $f_2$, cs, $\lambda_1$, $\lambda_2$) *of $q$ satisfied that* $\exists \lambda_{f_1,G}, \lambda_{f_2,G}$, *such that* $\lambda_{f_1,G}[\lambda_1] = \lambda_{f_2,G}[\lambda_2]$

Proposition 2 verifies whether each composition of the query can find at least one instance in the large graph from the sampled portion. There could be false negatives simply because the sampled graphs may not cover all possible compositions of the large graph, even one may increase the sampling size for higher accuracy. Prop. 2 is used in both online candidate generation and indexing of query compositions offline.

## 5.2 Suggestion Ranking

From our preliminary experiments, we observed that the number of candidate suggestions can be thousands. Considering the users may only be able to interpret a small subset of them, FLAG returns top-k suggestions w.r.t. a ranking function and a user preference. Suggestion ranking criteria of existing studies [36] are either infeasible to obtain from large graphs for efficient online autocomplete (*e.g.*, sel) or indistinguishable among the candidate suggestions (*e.g.*, dist) because the increment parts share no common subgraphs (*i.e.*, mces) and yield the same dist value. As the first attempt on GQAC for large graphs, we present a ranking function prefers query suggestions that *i) augments the existing query more* and *ii) summarizes more candidate suggestions.* The first preference simply reflects the user's intent to adopt larger useful increments, whereas the second one recognizes the importance of summarizing more suggestions that can be useful to the user. These two preferences can be quantified as *specialization power* and *summarization power*. We then combine these two criteria to measure the utility of a set of query suggestions.

**Definition 12** (Specialization power (special)) Given a set of candidate suggestions $\mathcal{U}$ to an existing query $q$, the specialization power of a suggestion $q' \in \mathcal{U}$ w.r.t. $q$ is defined as

$$\mathsf{special}(q', q) = \frac{\mathsf{SP}(q') - \mathsf{SP}(q)}{\max(\{\mathsf{SP}(q'') - \mathsf{SP}(q) | q'' \in \mathcal{U}\})}.$$

The specialization power of a suggestion $q'$ is defined as the increment of the specialization value SP if the user

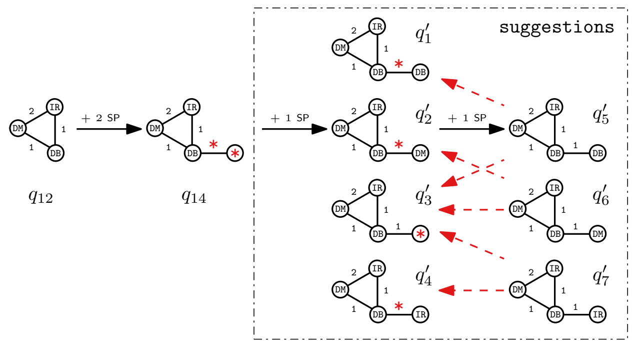


**Fig. 8** An example of suggestions in relation to SP

adopts the suggestion, and normalized by the maximum specialization value increment of all candidate suggestions.

**Definition 13** (Summarization power (summary)) Given a set of candidate suggestions $\mathcal{U}$ to an existing query $q$, the summarization power of a subset of candidate suggestions $Q' \subseteq \mathcal{U}$ w.r.t. $\mathcal{U}$ is defined as

$$\mathsf{summary}(Q') = \frac{|\mathsf{SM}(Q')|}{|\mathcal{U}|}.$$

The summarization power of a set of suggestions is defined as the number of candidate suggestions summarized by them, and normalized by the total number of candidate suggestions.

***Example 7*** We illustrate specialization power (special) and summarization power (summary) using Fig. 8. The user manually adds a wildcard edge (with a wildcard node) to query $q_{12}$ and obtains $q_{14}$. There are 7 candidate suggestions for the current query $q_{14}$, *i.e.*, $q'_1$, $q'_2$, ..., $q'_7$. According to Def. 12, $\mathsf{special}(q'_1, q_{14}) = 0.5$ and $\mathsf{special}(q'_5, q_{14}) = 1$. According to Definition 11 and 13, $\mathsf{SM}(q'_1) = \{q'_1, q'_5\}$ and $\mathsf{SM}(q'_3) = \{q'_3, q'_5, q'_6, q'_7\}$. Then, $\mathsf{summary}(\{q'_1, q'_3\})$ is $\frac{5}{7}$.

**Definition 14** (Utility of query suggestions) Given a set of query suggestions $Q' : \{q'_1, q'_2, \ldots, q'_k\}$, the specialization power of each suggestion with respect to the existing query $q$, the summarization power of $Q'$ with respect to all candidate suggestions, a user preference component $\alpha \in [0, 1]$, and scaling factors $\gamma$ and $\eta$, the utility of $Q'$ is defined as follows:

$$\mathsf{util}(Q') = \frac{\alpha}{k} \sum_{q' \in Q'} \mathsf{special}(q', q)^{\gamma} + (1 - \alpha)\mathsf{summary}(Q')^{\eta}$$

The bi-criteria ranking function combines the specialization power and summarization power of the query suggestions. $\alpha$ is a parameter to set the preference between the two criteria, and the constant denominator $k$ is for normalization. Since the values of the two criteria can be of very different ranges in practice, which makes $\alpha$ sensitive and difficult to tune, we introduce the scaling factors. The parameters $\alpha$, $\gamma$ and $\eta$ are data-specific. In order to tune the parameters, we adopt a machine learning method. However, this requires all the functions involved to be differentiable. However, the maximum function in Def. 12 is not continuous and differentiable. We adopt a differentiable approximation to the maximum function [4]. Hence, in the experiments, we can use a stochastic gradient descent algorithm to learn the parameters.

***Example 8*** Continuing with Fig. 8, we illustrate the utility of query suggestions defined in Def. 14. $\gamma$ and $\eta$ are set to 1. There are 7 candidate suggestions to the existing query $q_{14}$, *i.e.*, $q'_1$, $q'_2$, ..., $q'_7$. When GQAC only considers how much the suggestions specialize the existing query (*i.e.*, $\alpha = 1$), $q'_5$ and $q'_6$ would be the top-2 suggestions. When GQAC only considers how much the suggestions summarize other candidate suggestions (*i.e.*, $\alpha = 0$), then $q'_1$ and $q'_3$ would be the top-2 suggestions. When $\alpha$ is set to 0.5, then $q'_1$ and $q'_5$ would be the top-2 suggestions.

The ranking task is then to find the top-$k$ candidate suggestions that have the highest util value. It can be noted that the two objectives special and summary of util can be competing: in practice, the summarization power of smaller queries are often larger as more candidate suggestions are summarized by smaller ones, whereas smaller queries provide smaller specialization power. It is not surprising that the problem of determining the query suggestions with the highest util value is NP-hard.

**Definition 15** (Ranked Subgraph Query Suggestions for Large Graphs (RSQL )) Given a query $q$, a set of query suggestions $Q'$, the ranking function util, a user preference component $\alpha$, and a user-specified constraint $k$, the *ranked subgraph query suggestions* problem is to determine a subset $Q''$, $\mathsf{util}(Q'')$ is maximized, *i.e.*, $Q'' \subseteq Q'$, $|Q''| \leq k$ and there is no other $Q''' \subseteq Q'$ such that $\mathsf{util}(Q''') > \mathsf{util}(Q'')$.

**Proposition 3** *The RSQL problem is NP-hard.*

*(Proof sketch)* The maximization of this utility function is NP-hard, by a reduction from the *Set Cover* (SC) problem. Given an instance of SC problem, each subset $S^i$ of elements $\{o^i_1, \cdots, o^i_m\}$ is converted to a candidate suggestion $q'_i$ that summarizes $q'_1$, ⋯, $q'_m$; and $k$ remains the same. $\alpha$ and $\eta$ of RSQL is set to 0 and 1, respectively. Finding the query suggestion set is then to find the $i$ query suggestions, where $i$ is smaller than or equal to $k$, that cover the candidate suggestions the most. It can be trivially mapped to the

solution of Sc, that covers all elements with the smallest number of subsets $i$. □

## 5.3 Efficient Summarization Computation

This subsection presents efficient algorithms for determining summary, which enables efficient ranking for the online autocompletion. We remark that the computation of special is straightforward, given $q$, and hence, is omitted.

The computation of summary depends on SM (Defs. 12 and 13). To determine whether suggestions summarize the others, *i.e.*, the specialization orders between them, we need to compute subgraph isomorphism between each pair of online suggestions. Hence, we derive a *necessary* condition for the specialization order between candidate suggestions and index them. This can be efficiently indexed for the following two reasons. (i) Some query suggestions are similar because they are composed by adding small increments on the *same* existing query graph. (ii) The specialization order between the wildcard features (*i.e.*, the increments) is available offline.

We formalize a *necessary* condition for the specialization order between candidate suggestions. We illustrate how to efficiently i) offline compute and index all possible specialization orders and ii) online prune the false ones based on current query graph $q$.

**Proposition 4** *Given two suggestions $q_1'$ and $q_2'$ to query $q$, where $q_1'$ is formed via* compose($q$,$f_{12}$, cs$_1$, $\lambda_{11}$, $\lambda_{12}$) *and $q_2'$ is formed via* compose($q$,$f_{22}$, cs$_2$, $\lambda_{21}$, $\lambda_{22}$), *$q_2'$ specializes $q_1'$ (i.e., $q_1' \prec q_2'$) only if*

1. *the increment of $q_2'$ specializes that of $q_1'$, i.e., $\Delta_{q_1'} \prec_\lambda \Delta_{q_2'}$; and*
2. *there exists one embedding $\lambda' \in \{\lambda | \Delta_{q_1'} \prec_\lambda \Delta_{q_2'}\}$, s.t., the nodes where $q_1'$ increments at matches that of $q_2'$ via $\lambda'$.*

The proposition can be established by a simple proof by contradiction. The first condition of Prop. 4 can be computed offline and then indexed. The second condition can be used in the online autocompletion to prune false specialization orders using the current query.

### 5.3.1 Indexing Wildcard Features

We extend Feature DAG index (FDag) [36] with the support of wildcards. Due to space limitations, we highlight the main ideas of the extensions but skip the verbose index definition. An illustration of the index is shown in Fig. 9. In particular, we index the wildcard features (shown in the bottom) in a DAG, where each index node represents a feature, and an edge represents a specialization order between features. All possible subgraph isomorphism embeddings are indexed (shown in $M$ of the index edge and $\zeta$ of the indexed content). That is, all the possible ways that two well-formed features are composed have been precomputed and indexed. This avoids computing specializations of features online. The features are further indexed by their SP values.

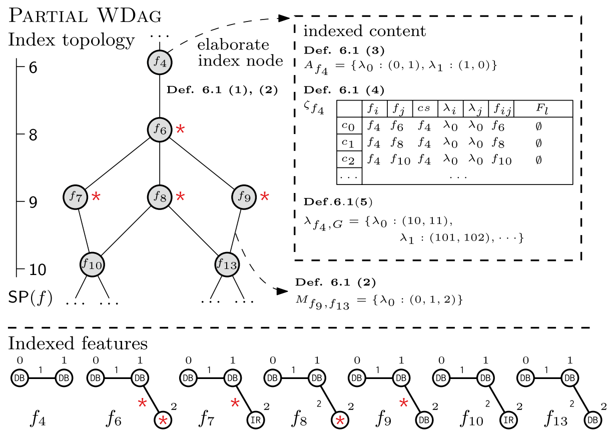

Partial WDag
Index topology
elaborate index node
indexed content
Def. 6.1 (3)
$A_{f_4} = \{\lambda_0 : (0,1), \lambda_1 : (1,0)\}$
Def. 6.1 (4)
$\zeta_{f_4}$
$f_i$ $f_j$ $cs$ $\lambda_i$ $\lambda_j$ $f_{ij}$ $F_l$
$c_0$ $f_4$ $f_6$ $f_4$ $\lambda_0$ $\lambda_0$ $f_6$ $\emptyset$
$c_1$ $f_4$ $f_8$ $f_4$ $\lambda_0$ $\lambda_0$ $f_8$ $\emptyset$
$c_2$ $f_4$ $f_{10}$ $f_4$ $\lambda_0$ $\lambda_0$ $f_{10}$ $\emptyset$
Def. 6.1 (1), (2)
Def.6.1(5)
$\lambda_{f_4,G} = \{\lambda_0 : (10,11), \lambda_1 : (101,102), \cdots\}$
Def. 6.1 (2)
$M_{f_9,f_{13}} = \{\lambda_0 : (0,1,2)\}$
6
8
9
10
SP($f$)
$f_4$ $f_6$ $f_7$ $f_8$ $f_9$ $f_{10}$ $f_{13}$
Indexed features
$f_4$ $f_6$ $f_7$ $f_8$ $f_9$ $f_{10}$ $f_{13}$

**Fig. 9** Index structure (partial) for CiteSeer

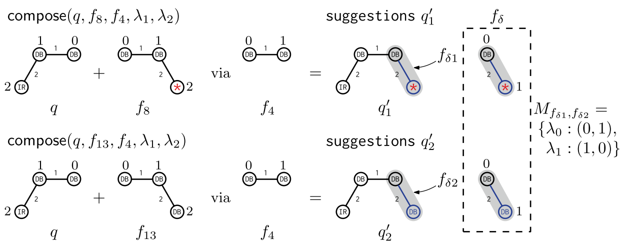

compose($q$, $f_8$, $f_4$, $\lambda_1$, $\lambda_2$)
suggestions $q_1'$
$f_\delta$
$f_{\delta 1}$
+
via
=
$q$ $f_8$ $f_4$ $q_1'$
$M_{f_{\delta 1}, f_{\delta 2}} = \{\lambda_0 : (0,1), \lambda_1 : (1,0)\}$
compose($q$, $f_{13}$, $f_4$, $\lambda_1$, $\lambda_2$)
suggestions $q_2'$
$f_{\delta 2}$
$q$ $f_{13}$ $f_4$ $q_2'$

**Fig. 10** Efficient suggestion summarization computation

***Example 9*** We illustrate the efficient specialization order computation with Fig. 10. Given two compositions, the specialization relation of the increments (*i.e.*, $f_{\delta_1} \prec f_{\delta_2}$) has been indexed. $\lambda_0 = (0,1)$, $\lambda_1 = (1,0)$ can be simply retrieved. During online autocompletion, we check the second condition of Prop. 4. We find that $\lambda_0 = (0,1)$ satisfies that the nodes where $q_1'$ increments at matches that of $q_2'$ via $\lambda_0$. Hence, the suggestion $q_2'$ specializes $q_1'$.

**Algorithm 3** Overall query autocompletion of FLAG

```
Input: a query $q$ and the set of feature embeddings $M_q$, user preference component $\alpha$, number of suggestions requested $k$ and max. increment size $\delta$
Output: the top-$k$ suggestions $Q'_k$
1: // generate candidate suggestions (Sec. 5.1)
2: Let $Q'$ be an empty set    // init. candidate suggestions
3: for all $(f, \lambda) \in M_q$ do
4:     $Q' \leftarrow Q' \cup Q'_C$    // $Q'_C$ is the possible suggestions composed by adding another feature to $f$, where $f$ is embedded in $q$ via $\lambda$, $q'_c \in Q'_C$ implies $|q'_c|$-$|q| \leq \delta$.
5: // prune empty suggestions
6: for all $q' \in Q'$ do
7:     for all composition $c$ of $q'$ do
8:         if composition $c$ does not satisfy Prop. 2 then
9:             $Q' \leftarrow Q' \setminus \{q'\}$
10: // prune false specializations (Prop. 4 of Sec. 5.3)
11: Let INDEX be the indexed suggestion specialization
12: for all $(c_i, c_j) \in$ INDEX do
13:     if $q_{c_i} \not\prec q_{c_j}$ then
14:         INDEX$\leftarrow$ INDEX$\setminus\{(c_i, c_j)\}$
15: // rank top-$k$ suggestions (Sec. 5.4)
16: Let $Q'_k$ be an empty set    // init. top-$k$ suggestions
17: for $i = 1 \ldots k$ do
18:     $q'_{max} \leftarrow$ argmax(util($Q'_k \cup \{q'\}$)), where $q' \in Q'$
19:     $Q'_k \leftarrow Q'_k \cup \{q'_{max}\}$
20:     $Q' \leftarrow Q' \setminus \{q'_{max}\}$
21: return $Q'_k$
```

## 5.4 Efficient Ranking Algorithm

Given that the ranking function util of a set of candidate suggestions $Q$ can be efficiently computed, we present a greedy ranking algorithm in Lines 16–20 of Algo 3. Greedy algorithms are typical approximation algorithm for RSQL because util is submodular. Recall that a function is submodular if the marginal gain from adding an element to a set $S$ is at least as high as the marginal gain from adding it to a superset of $S$. In particular, it satisfies: $f(S \cup \{o\}) - f(S) \geq f(T \cup \{o\}) - f(T)$, for all element $o$ and all pair of sets $S \subseteq T$. We can analyze util as follows. Firstly, $\sum$ special is linear and monotone submodular since it is a sum of non-negative numbers. Secondly, summary is monotone submodular because adding new suggestions can only summarize more candidate suggestions. Hence, util is a non-negative linear combination of the two scaled monotone submodular components. Thus, util is monotone submodular. The problem of maximizing a monotone submodular function subject to a cardinality constraint admits a $1 - 1/e$ approximation algorithm [27].

## 6 Experimental Evaluation

This section presents an experimental evaluation of FLAG. We first investigated the suggestion quality via user study and then conducted an extensive performance evaluation via simulation on popular real datasets. In particular, we studied the overall performance of FLAG, the effectiveness of the optimizations, and the effects of the parameters of FLAG.

**Table 2** Some characteristics of the datasets

| Dataset | $\|V\|$ | $\|E\|$ | $\|l(V)\|$ | $\|l(E)\|$ | $\|\bar{\deg}(V)\|$ |
|---|---|---|---|---|---|
| TWITTER | 11,316,811 | 85,331,845 | 32 | 1 | 15.1 |
| WORDNET | 73,753 | 234,024 | 28 | 1 | 6.3 |
| CITESEER | 3,312 | 4,591 | 6 | 3 | 2.8 |

## 6.1 Software and Hardware

We implemented the FLAG prototype on top of AUTOG[36]. The prototype was mainly implemented in C++, using VF2 [7] for subgraph query processing and the McGregor's algorithm [21] (with minor adaptations) for determining common subgraphs. We used GSPAN [35] for frequent subgraph mining. We conducted all the experiments on a machine with a 2.2GHz Xeon E5-2630 processor and 256GB memory, running Linux. All the indexes were built offline and loaded from the hard disk and were then made fully memory-resident for online query autocompletion.

## 6.2 Datasets

We conducted experiments on several different workload settings by employing real graph datasets with various characteristics. Table 2 reports some dataset characteristics.

1. TWITTER. [9] This dataset models the Twitter social network. It consists of ~11M vertices and ~85M edges. Each vertex represents a user and each edge represents the friendship/followership relation between two users. The original graph has no labels. We randomly added labels to the vertices. The number of distinct labels was set to 32 and the randomization follows a Gaussian distribution ($\mu$=50 and $\sigma$=3).
2. WORDNET. [10] This dataset models the lexical network of words. It consists of ~74K vertices and ~234K edges. Each vertex represents an English word and each edge represents the relationships between them, such as synonym, antonym, and meronym. The original graph has no labels. We randomly added labels to the vertices, similar to the way used in TWITTER.
3. CITESEER. [11] This dataset models publications in CiteSeer. It consists of ~3K vertices and ~4K edges. Each

[9] http://socialcomputing.asu.edu/datasets/Twitter
[10] https://networkrepository.com/wordnet-words.php
[11] https://linqs.soe.ucsc.edu/data

vertex represents a publication and each edge represents the citation relation between two publications. Each vertex is labeled with the Computer Science area (*e.g.*, `DB`, `DM`, `IR`) and each edge is labeled with the Jaccard distance between the pair of publications. The distance is computed from the word attributes of the publications and further evenly categorized into three types (small, medium, large distances).

## 6.3 Query Sets

We generated numerous sets of query graphs of different query sizes $|q|$ (the number of edges) and various frequencies in the large graph. Each query set contained 100 graphs. [12] In particular, we generated queries that yield different result set sizes (*i.e.*, $|G_q| > |G_q^{\min}|$ for all query graphs). These query sets enable us to investigate the usefulness and performance of FLAG with different user workloads. Query sets of query sizes ranged from 2 to 9.

## 6.4 Graph Sampling

Instead of running expensive frequent subgraph mining algorithms on the single large graph, we scaled down the large graph using *Random Walk* sampling [16] before frequent subgraph mining. We sampled $\min\{|V(G)|, 10^6\}$ graphs of 10 edges from the large graph.[13] In particular, we randomly selected a vertex as the starting vertex and then simulated a random walk on the graph. At each step, there is a probability 0.15 (the value commonly used in literature) we jumped to the starting vertex and continued the random walk. If we cannot meet the required sample graph size after a large number of steps (*e.g.*, $100 * |V(G|)$ or random walk has exhausted the neighbors of the starting vertex, we would select another starting vertex and restart the random walk.

## 6.5 Feature Mining

We followed AutoG using gSpan[35] to obtain a sufficient number of features (frequent subgraphs) to build the index offline.[14] In particular, we set the default minimum support value (minSup) to 0.2, 0.3, and 0.5% for Twitter, WordNet, and CiteSeer, respectively. These minimum support values

**Table 3** Some characteristics of the features of datasets

| Dataset | minSup | $\|F\|$ | $\|\text{avg}(V)\|$ | $\|\text{avg}(E)\|$ | Time(s) |
|---|---|---|---|---|---|
| Twitter | 0.2% | 1859 | 3.33 | 2.33 | 114.6 |
| WordNet | 0.3% | 1745 | 3.30 | 2.33 | 8.7 |
| CiteSeer | 0.5% | 1720 | 5.72 | 4.92 | 1.4 |

**Table 4** Index construction

| Dataset | $\|V\|$ | $\|E\|$ | time (s) | # compositions | Time(s) |
|---|---|---|---|---|---|
| Twitter | 1,859 | 12,637 | 1.0 | 244,195 | 1,711,636 |
| WordNet | 1,745 | 11,862 | 0.9 | 298,369 | 16,333 |
| CiteSeer | 1,720 | 20,842 | 2.3 | 8,481,895 | 151,847 |

are an order of magnitude smaller than those used in AutoG. We set smaller minSups because that frequent subgraphs are relatively scarce in large graphs. The maximum feature size *maxL* was set to 10 for all datasets. Some statistics of the features are summarized in Table 3.

## 6.6 Index

With frequent features mined by gSpan, we adopted the AutoG procedure (*i.e.*, Algorithm 4 of [36]) to enumerate the possible compositions of feature pairs. We discovered that the pruning technique proposed in AutoG for composition enumeration is ineffective for the employed large graphs. Their pruning technique can prune 13 and 45% of the empty compositions for the Aids and PubChem datasets. It is not surprising to find this necessary condition only prunes 0.1% of the compositions on CiteSeer since the characteristics of citation network are much different from those of chemical and biological structures.

After applying the embedding-based necessary condition for non-empty query compositions (introduced in Sect. 5.1), 41% of the compositions for the CiteSeer dataset are pruned. Table 4 briefly summarizes the characteristics of constructing an index and enumerating compositions, respectively.

## 6.7 Quality Metrics

We adopted several popular metrics to measure suggestion qualities [25, 36]. We report the number of suggestion adoptions (*i.e.*, #Auto) and the *total profit metric* (*i.e.*, tpm). Specifically, the *total profit metric* (tpm) [25, 36] quantifies the percentage of mouse clicks saved by adopting suggestions during the visual query formulation.

$$\frac{\textit{no. of clicks saved by suggestions}}{\textit{no. of clicks without suggestions}} \times 100\%.$$

[12] A query was generated following the *Random Walk* sampling (same as graph sampling). We checked that the constructed query $q$ was not generated before and with a result set size $|G_q|$ large than $|G_q^{\min}|$.

[13] This limitation is due to the gSpan binary executable.

[14] We have investigated several existing feature mining work before opting to apply gSpan to graph samples.

**Table 5** Quality metrics and their meanings

| Metric | Meaning |
|---|---|
| #AUTO | The average number of suggestions accepted in the simulation |
| SP | The total number of specializations obtained from suggestions |
| avg(SP) | The average number of specializations obtained from each accepted suggestion |
| #AUTO | The useful suggestion ratio U defined as $\frac{\textit{no. of useful suggestions}}{\textit{no. of returned suggestions}} \times 100\%$ |
| TPM | The *total profit metric (TPM)* adopted from [25], which quantifies the % of specializations saved by FLAG in the visual graph query formulation: $TPM = \frac{\textit{no. of specializations saved by FLAG}}{\textit{no. of specializations without FLAG}}$ |

In addition to #AUTO and TPM, we report the number of specializations from adopting suggestions (denoted as SP), the average number of specializations from each adoption (denoted as avg(SP)) and the useful suggestion ratio $U$ defined as $\frac{\textit{no. of useful suggestions}}{\textit{no. of returned suggestions}} \times 100\%$. Each reported number is the average of the 100 queries in each query set. Note that even when the suggestions are correct, users still need at least a mouse click to adopt them to obtain the target query. The employed quality metrics are listed in Table 5.

## 6.8 Learning Scaling Factors

We used a stochastic gradient descent algorithm to learn the default scaling factors for Definition 14. We generate 100 random simple queries from a dataset. Each initial query contains 1 edge and its target query contains 4 edges. We divide the queries into 10 groups. Each group is used to learn the parameters around 33 iterations. The learning rate is set to 0.01. The learning algorithm converges at around 300 iterations. For TWITTER dataset, the default $\gamma$ and $\eta$ are 3.8 and 7.6, respectively. For CITESEER dataset, we obtained the defaults similarly. Their values are 3.6 and 7.2, respectively. The learned $\alpha$ for TWITTER and CITESEER are 0.58 and 0.45, respectively. $\gamma$, $\eta$ and $\alpha$ values of the WORDNET dataset are 1, 1, and 0.5, respectively.

## 6.9 Suggestion Qualities of FLAG

### 6.9.1 User Study

We first conducted a user test with 10 volunteers. Each user was given 3 queries with high, medium, and low TPM values, respectively, from the simulation. We randomly shuffled these 9 queries. 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 "FLAG *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*".

Consistent with [36, 37], our result showed that the correlation coefficient between TPMs and users' points is 0.819 and the $p$-value is 0.007. Thus, TPM is a good quality indication of FLAG. The average ratings of the queries with high, medium, and low TPM values are 3.57 (between "*strongly agree*" and "*agree*"), 2.63 ("*neither agree nor disagree*") and 1.83 (between "*disagree*," and "*strongly disagree*"), respectively.

### 6.9.2 Large-Scale Simulations

We investigated the suggestion qualities via simulations under a large variety of parameter settings. $\alpha$ is set to 0.5 so that both special and summary contribute to ranking. For each target query, we started with a random edge with one node label (the other node and edge are with wildcard labels). In each step, we called FLAG. Then, we chose the useful suggestion with the largest number of specializations. If no useful suggestions were returned, we specialized the query by a random specialization operator toward the target query. Each target query set contains 100 queries.

We studied the effects of the major parameters of FLAG on CITESEER, WORDNET, and TWITTER. We report the representative simulation results in Tables 6–17. The performance characteristics presented here can be useful for users to set their default parameter values, which could be dataset-specific.

### 6.9.3 Varying the Maximum Increment Sizes ($\delta_{max}$)

Table 6 shows the quality metrics of Q5 (*i.e.*, queries of 5 edges) with various $\delta_{max}$ on CITESEER. The results show the qualities decrease as $\delta_{max}$ increases. #AUTO shows that the suggestions were used in multiple iterations of the

**Table 6** Quality metrics by varying $\delta_{max}$ (CITESEER)

| $\delta_{max}$ | #AUTO | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 4 | 4.9 | 13.1 | 60 | 2.7 | 14 |
| 8 | 5.2 | 12.6 | 54 | 2.6 | 11 |
| 12 | 5.2 | 11.7 | 47 | 2.4 | 10 |
| 16 | 4.9 | 11.4 | 48 | 2.4 | 9 |
| 20 | 3.9 | 11.8 | 58 | 4.0 | 9 |

**Table 7** Quality metrics by varying $\delta_{\max}$ (WordNet)

| $\delta_{\max}$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 4 | 5.1 | 10.6 | 40 | 2.1 | 8 |
| 8 | 4.3 | 9.2 | 36 | 2.2 | 7 |
| 12 | 3.6 | 7.7 | 30 | 2.2 | 6 |
| 16 | 3.4 | 7.6 | 31 | 2.3 | 6 |
| 20 | 3.4 | 7.6 | 31 | 2.3 | 6 |

**Table 8** Quality metrics by varying $\delta_{\max}$ (Twitter)

| $\delta_{\max}$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 4 | 5.1 | 11.3 | 44 | 2.2 | 9 |
| 8 | 3.7 | 9.4 | 40 | 2.7 | 6 |
| 12 | 3.2 | 8.7 | 39 | 2.9 | 6 |
| 16 | 3.2 | 8.7 | 39 | 2.9 | 6 |
| 20 | 3.2 | 8.7 | 39 | 2.9 | 6 |

**Table 9** Quality metrics by varying $|q|$ (CiteSeer)

| $\|q\|$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 2 | 2.6 | 5.3 | 44 | 2.1 | 8 |
| 3 | 3.8 | 8.4 | 46 | 2.3 | 9 |
| 4 | 5.2 | 11.7 | 47 | 2.4 | 10 |
| 5 | 5.8 | 14.9 | 52 | 3.0 | 10 |
| 6 | 7.0 | 18.5 | 54 | 2.9 | 10 |
| 7 | 8.1 | 21.5 | 54 | 2.9 | 10 |
| 8 | 8.8 | 24.3 | 54 | 3.0 | 10 |
| 9 | 10.8 | 27.7 | 52 | 2.8 | 10 |

**Table 10** Quality metrics by varying $|q|$ (WordNet)

| $\|q\|$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 2 | 1.9 | 4.0 | 35 | 2.2 | 6 |
| 3 | 2.7 | 6.1 | 34 | 2.4 | 6 |
| 4 | 3.6 | 7.7 | 30 | 2.2 | 6 |
| 5 | 4.3 | 9.6 | 31 | 2.3 | 6 |
| 6 | 4.4 | 10.4 | 28 | 2.5 | 5 |
| 7 | 5.2 | 12.0 | 28 | 2.4 | 5 |
| 8 | 6.0 | 13.5 | 27 | 2.3 | 5 |
| 9 | 6.3 | 14.2 | 24 | 2.3 | 5 |

**Table 11** Quality metrics by varying $|q|$ (Twitter)

| $\|q\|$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 2 | 1.4 | 3.9 | 41 | 2.9 | 5 |
| 3 | 2.6 | 7.0 | 44 | 3.1 | 6 |
| 4 | 3.2 | 8.7 | 39 | 2.9 | 6 |
| 5 | 4.4 | 11.4 | 39 | 2.7 | 6 |
| 6 | 5.1 | 12.8 | 35 | 2.6 | 5 |
| 7 | 6.3 | 15.4 | 35 | 2.6 | 5 |
| 8 | 7.2 | 17.8 | 35 | 2.6 | 5 |
| 9 | 8.2 | 19.9 | 35 | 2.5 | 5 |

**Table 12** Quality metrics by varying $k$ (CiteSeer)

| $k$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 4 | 4.9 | 10.3 | 40 | 2.3 | 19 |
| 6 | 5.0 | 11.0 | 44 | 2.4 | 14 |
| 8 | 5.2 | 11.5 | 46 | 2.4 | 12 |
| 10 | 5.2 | 11.7 | 47 | 2.4 | 10 |

**Table 13** Quality metrics by varying $k$ (WordNet)

| $k$ | #Auto | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 4 | 2.0 | 4.5 | 19 | 2.4 | 6 |
| 6 | 2.3 | 5.3 | 22 | 2.4 | 5 |
| 8 | 2.9 | 6.5 | 27 | 2.3 | 5 |
| 10 | 3.6 | 7.7 | 30 | 2.2 | 6 |

query formulation. In particular, the formulation process of each query adopted around 5.7 suggestions on average. SP shows that the number of specialization added by FLAG was around 15. TPM shows that FLAG saved roughly 53% manual specialization in query formulation. avg(SP) shows that each adoption introduced 2–3 specializations to the existing query. U shows that FLAG generally produced useful suggestions. Tables 7 and 8 show the quality metrics of Q4 with various $\delta_{\max}$ on WordNet and Twitter. The results of WordNet and Twitter share the same trends as that of CiteSeer. The values of quality metrics of WordNet and Twitter were lower than CiteSeer since the number of compositions of WordNet and Twitter was relatively few.

### 6.9.4 Varying the Target Query Sizes ($|q|$)

Tables 9, 10 and 11 show the quality metrics of various $|q|$. It is not surprising that FLAG achieved more suggestion adoptions as $|q|$ increased. The number of adoption (#Auto) and adopted specializations (SP) increased as $|q|$ increased. TPM and U of CiteSeer, WordNet, and Twitter generally retained as $|q|$ increased.

### 6.9.5 Varying the User-Specified Constant $k$

Tables 12, 13 and 14 show the suggestion quality when we varied $k$. The results show that #Auto, SP, TPM, and avg(SP) generally increased with $k$. It is not surprising because as more suggestions are returned, the higher chance some of

**Table 14** Quality metrics by varying $k$ (TWITTER)

| $k$ | #AUTO | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 4 | 2.3 | 6.1 | 28 | 2.9 | 8 |
| 6 | 2.8 | 7.4 | 33 | 2.8 | 7 |
| 8 | 3.1 | 8.2 | 36 | 2.9 | 6 |
| 10 | 3.2 | 8.7 | 39 | 2.9 | 6 |

**Table 15** Quality metrics by varying $\alpha$ (CITESEER)

| $\alpha$ | #AUTO | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 0.00 | 5.6 | 13.1 | 55 | 2.5 | 15 |
| 0.20 | 5.2 | 12.9 | 56 | 2.8 | 12 |
| 0.40 | 5.4 | 12.0 | 49 | 2.4 | 10 |
| 0.60 | 4.5 | 11.0 | 48 | 2.6 | 9 |
| 0.80 | 2.8 | 10.2 | 54 | 4.4 | 7 |
| 1.00 | 0.3 | 3.0 | 20 | 11.5 | 1 |

**Table 16** Quality metrics by varying $\alpha$ (WORDNET)

| $\alpha$ | #AUTO | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 0.00 | 5.4 | 11.2 | 42 | 2.1 | 12 |
| 0.20 | 5.0 | 10.4 | 39 | 2.1 | 8 |
| 0.40 | 4.3 | 9.0 | 34 | 2.2 | 7 |
| 0.60 | 2.6 | 6.1 | 26 | 2.5 | 4 |
| 0.80 | 1.4 | 3.6 | 16 | 2.8 | 2 |
| 1.00 | 0.2 | 1.5 | 9 | 6.6 | 1 |

them are adopted. Importantly, the useful suggestions ratio is higher when $k$ is smaller mainly because the useful suggestions of CITESEER usually rank higher than WORDNET and TWITTER.

### 6.9.6 Varying $\alpha$ of the Ranking Functions

Tables 15, 16 and 17 show the suggestion quality with various $\alpha$s. The results show that the suggestion qualities were generally good when $\alpha$ was small. The optimal for CITESEER was 0.2, that for WORDNET was around 0, and that for TWITTER was 0.0-0.4. Then, the quality decreased as the value of $\alpha$ increased. The learned $\alpha$s from Sect. 6.8 produced slightly lower TPM when compared to the optimal ones. FLAG generally produced high-quality suggestions when $\alpha$s are smaller than 0.8 for CITESEER, 0.2 for WORDNET, and 0.8 for TWITTER.

**Table 17** Quality metrics by varying $\alpha$ (TWITTER)

| $\alpha$ | #AUTO | SP | TPM | avg(SP) | U |
|---|---|---|---|---|---|
| 0.00 | 5.6 | 11.8 | 44 | 2.1 | 13 |
| 0.20 | 5.2 | 11.3 | 44 | 2.2 | 9 |
| 0.40 | 4.4 | 10.5 | 44 | 2.5 | 7 |
| 0.60 | 3.1 | 8.6 | 39 | 3.0 | 4 |
| 0.80 | 2.2 | 6.9 | 34 | 3.4 | 2 |
| 1.00 | 0.4 | 2.0 | 11 | 5.0 | 1 |

## 6.10 Efficiency of FLAG

We conducted a detailed evaluation of the online FLAG processing. We report the *Average Response Time* (ART) of FLAG under the default setting in Fig. 11.[15] For CITESEER, we obtained ARTS around 3s. For TWITTER, we obtained short ARTS as the number of compositions was relatively few. Thus, the response time of FLAG is generally very short. The rest of this section reports the average response time when we vary major parameters of FLAG, *i.e.*, $\alpha$, $k$, and $|q|$.

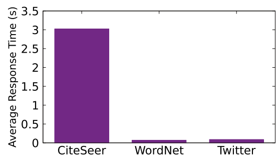


**Fig. 11** ART - default

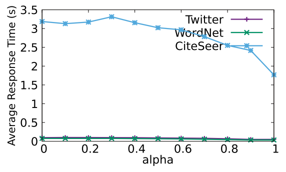


**Fig. 12** ART VS $\alpha$

### 6.10.1 Varying $\alpha$ of the Ranking Functions

We ranged $\alpha$ from 0 to 1. Figure 12 shows the effects of $\alpha$ on ARTS. The ART was always less than 3.5s. We also noticed that the ART decreased when $\alpha$ approaching 1. The higher the value of $\alpha$, the GQAC process prefers suggestions with large specialization more and small summarization, which results in shorter time for updating the summarizations of the candidate suggestions.

### 6.10.2 Varying the User-Specified Constant $k$

We varied $k$ from 10 to 50 and reported the ARTS for CITESEER and TWITTER in Fig. 13. The largest value of $k$ tested was 50, which is large enough for common visual interfaces. The results show that the ARTS increased as $k$ increased. FLAG returned suggestions within 5s when $k$ is less than 20. The

[15] We remark that query decomposition takes less than a few milliseconds, which are negligible, and hence, is not shown separately.

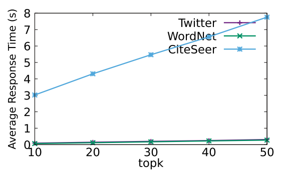


**Fig. 13** ART vs $k$

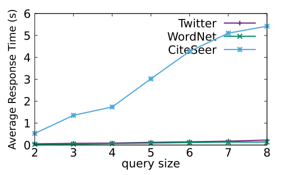


**Fig. 14** ART vs $|q|$

GQAC process may need 8s to provide suggestions when $k$ is up to 50.

### 6.10.3 Varying the Target Query Sizes ($|q|$)

Figure 14 shows the ART as the query size increased. The results show that the autocomplete process of FLAG finished within 6s for queries with up to 8 edges. The ART increased when the query size $|q|$ increased. The ART increased mainly because large queries required more time to generate more candidate suggestions and then rank them.

## 7 Related Work

Query formulation aids have recently gained increasing research attention. Firstly, recent work has proposed a variety of innovative approaches to help query formulation. For example, GestureQuery [26] proposes to use *gestures* for specifying SQL queries. *SnapToQuery* [15] guides users to explore query specification via *snapping* user's likely intended queries. [3] has proposed a data-driven approach for GUI construction. Exploratory search has been demonstrated as useful for enhancing interactions between users and search systems (*e.g.*, [20, 22, 23]). Quble [11] allows users to explore regions of a graph that contains at least a query answer. Wang et al. [32] recently propose efficient visual exploratory search in graph databases. Huang et al. [10] study canned subgraph patterns for GUI. SeeDB [31] proposes *visualization* recommendations for supporting data analysis. [18] introduces *Meaningful Query Focus* (MQF) of a given keyword search to generate XQuery. While keyword search (*e.g.*, [33]) has been proposed to query graphs, this approach does not allow users to precisely specify query structures. This paper contributes to *query autocompletion* for query formulation.

Secondly, there is existing work on query autocompletion on various query types. For instance, there is work on query autocompletion for keyword search (*e.g.*, [2, 25, 34]) and structured queries (*e.g.*, [24]). Li et al. [9] extend keyword search autocompletion to XML queries. [18] associated structures to query keywords. LotusX provides position-aware autocompletion capability for XML [19]. An autocompletion learning editor for XML provides intelligence autocompletion [1]. [12] presents a conversational mechanism that accepts incomplete SQL queries, which then matches and replaces a part (user focus) of the previously issued queries. There has been a stream of work on extending Query By Example to construct structural queries, *e.g.*, [5, 6, 14]. In contrast, this paper focuses on structural queries for graphs. Hence, we only include related work on *graphs*.

Regarding GQAC, Yi et al. [36] proposed AutoG to rank subgraph suggestions for graphs of small or modest sizes. The recent work [28, 37] introduced user focus to GQAC. In [22], Mottin et al. proposed graph query reformulation, which determines a set of reformulated queries that maximally cover the *results* of the current query. In Pienta et al. [29] and Li et al. [13], the authors demonstrated interactive methods to produce *edge* or *node* suggestions for visual graph query construction. In contrast, this paper considers flexible subgraph suggestions for large graphs.

## 8 Conclusion

We proposed FLAG that exploits the *wildcard label* notion to generate top-$k$ query suggestions to help the query formulation for large graphs. Considering that the graph features exploited by existing GQAC studies are either absent or rare in large graphs, we proposed to introduce wildcard labels for query graph and query suggestions to allow more query suggestion candidates. Candidate query suggestions are ranked by a new ranking function that considers how much the suggestion augments the existing query and how many other suggestions it summarizes. We proposed efficient algorithm for suggestion ranking. Our user study and experiments verified both the effectiveness and efficiency of FLAG.

This paper leads to a variety of interesting future work. We are extending the study of histories of users' activities [38] into the ranking. We are studying the explanations of the few cases (*e.g.*, [17]), where GQAC returned incorrect suggestions.

**Funding** Hong Kong Research Grants Council (C6030-18GF, 12201119, 12201518), Hong Kong Baptist University (IRCMS/19-20/H01), and Prof. Byron Choi.

## References

1. Abiteboul S, Amsterdamer Y, Milo T, Senellart P (2012) Auto-completion learning for xml. In *SIGMOD*, pages 669–672
2. Bast H, Weber I (2006) Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371
3. Bhowmick SS, Choi B, Dyreson CE (2016) Data-driven visual graph query interface construction and maintenance: challenges and opportunities. PVLDB 9:984–992
4. Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press, Cambridge
5. Braga D, Campi A, Ceri S (2005) XQBE (XQuery By Example): a visual interface to the standard xml query language. In *TODS*, pages 398–443
6. Comai S, Damiani E, Fraternali P (2001) Computing graphical queries over xml data. *TOIS*, pages 371–430
7. Cordella LP, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. *PAMI*, pages 1367–1372
8. Elseidy M, Abdelhamid E, Skiadopoulos S, Kalnis P (2014) Grami: frequent subgraph and pattern mining in a single large graph. PVLDB 7:517–528
9. Feng J, Li G (2012) Efficient fuzzy type-ahead search in xml data. *TKDE*, pages 882–895
10. Huang K, Chua H, Bhowmick SS, Choi B, Zhou S (2019) CATAPULT: data-driven selection of canned patterns for efficient visual graph query formulation. In *SIGMOD*, pages 900–917
11. Hung HH, Bhowmick SS, Truong BQ, Choi B, Zhou S (2013) QUBLE: blending visual subgraph query formulation with query processing on large networks. In *SIGMOD*, pages 1097–1100
12. Ioannidis YE, Viglas S (2006) Conversational querying. *Inf. Syst.*, pages 33–56
13. Jayaram N, Goyal S, Li C (2015) VIIQ: Auto-suggestion enabled visual interface for interactive graph query formulation. *PVLDB*, pages 1940–1951
14. Jayaram N, Gupta M, Khan A, Li C, Yan X, Elmasri R (2014) GQBE: querying knowledge graphs by example entity tuples. In *ICDE*, pages 1250–1253
15. Jiang L, Nandi A (2015) Snaptoquery: providing interactive feedback during exploratory query specification. PVLDB 8(11):1250–1261
16. Leskovec J, Faloutsos C (2006) Sampling from large graphs. In *KDD*
17. Li J, Cao Y, Ma S (2017) Relaxing graph pattern matching with explanations. In *CIKM*
18. Li Y, Yu C, Jagadish HV (2008) Enabling schema-free xquery with meaningful query focus. *VLDB J.*, pages 355–377
19. Lin C, Lu J, Ling TW, Cautis B (2012) LotusX: a position-aware xml graphical search system with auto-completion. In *ICDE*, pages 1265–1268
20. Marchionini G (2006) Exploratory search: from finding to understanding. *Commun. ACM*, pages 41–46
21. McGregor JJ (1982) Backtrack search algorithms and the maximal common subgraph problem. *Softw., Pract. Exper.*, pages 23–34
22. Mottin D, Bonchi F, Gullo F (2015) Graph query reformulation with diversity. In *KDD*, pages 825–834
23. Mottin D, Müller E (2017) Graph exploration: From users to large graphs. In *SIGMOD*, pages 1737–1740
24. Nandi A, Jagadish HV (2007) Assisted querying using instant-response interfaces. In *SIGMOD*, pages 1156–1158
25. Nandi A, Jagadish HV (2007) Effective phrase prediction. In *VLDB*, pages 219–230
26. Nandi A, Jiang L, Mandel M (2013) Gestural query specification. PVLDB 7(4):289–300
27. Nemhauser GL, Wolsey LA, Fisher ML (1978) An analysis of approximations for maximizing submodular set functions - i. *Math. Program.*, pages 265–294
28. Ng N, Yi P, Zhang Z, Choi B, Bhowmick SS, Xu J (2019) Fgreat: focused graph query autocompletion. In *ICDE*, pages 1956–1959
29. Pienta R, Hohman F, Tamersoy A, Endert A, Navathe SB, Tong H, Chau DH (2017) Visual graph query construction and refinement. In *SIGMOD*, pages 1587–1590
30. Sahu S, Mhedhbi A, Salihoglu S, Lin J, Özsu MT (2017) The ubiquity of large graphs and surprising challenges of graph processing. PVLDB 11:420–431
31. Vartak M, Rahman S, Madden S, Parameswaran A, Polyzotis N (2015) Seedb: efficient data-driven visualization recommendations to support visual analytics. PVLDB 8(13):2182–2193
32. Wang C, Xie M, Bhowmick SS, Choi B, Xiao X, Zhou S (2020) FERRARI: an efficient framework for visual exploratory subgraph search in graph databases. VLDB J 29(5):973–998
33. Wu Y, Yang S, Srivatsa M, Iyengar A, Yan X (2013) Summarizing answer graphs induced by keyword queries. PVLDB 6:1774–1785
34. Xiao C, Qin J, Wang W, Ishikawa Y, Tsuda K, Sadakane K (2013) Efficient error-tolerant query autocompletion. *PVLDB*, pages 373–384
35. Yan X, Han J (2002) gSpan: graph-based substructure pattern mining. In *ICDM*, pages 721–724
36. Yi P, Choi B, Bhowmick SS, Xu J (2017) Autog: a visual query autocompletion framework for graph databases. VLDB J 26(3):347–372
37. Yi P, Li J, Choi B, Bhowmick SS, Xu J (2020) Gfocus: user focus-based graph query autocompletion. *TKDE*
38. Zhang A, Goyal A, Kong W, Deng H, Dong A, Chang Y, Gunter CA, Han J (2015) adaqac: adaptive query auto-completion via implicit negative feedback. In *SIGIR*, pages 143–152